Formation DBA5

PostgreSQL Sauvegardes Avancées

17.12

Dalibo SCOP

https://dalibo.com/formations

PostgreSQL Sauvegardes Avancées

Formation DBA5

TITRE : PostgreSQL Sauvegardes Avancées
SOUS-TITRE : Formation DBA5

REVISION: 17.12
DATE: 8 janvier 2018
ISBN: 979-10-97371-04-3
COPYRIGHT: © 2005-2017 DALIBO SARL SCOP
LICENCE: Creative Commons BY-NC-SA

Le logo éléphant de PostgreSQL (”Slonik”) est une création sous copyright et
le nom ”PostgreSQL” est marque déposée par PostgreSQL Community Association
of Canada.

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, Sharon Bonan, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Virginie Jourdan, Guillaume Lelarge, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Flavie Perette, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Cédric Villemain, Thibaud Walkowiak

---

À propos de DALIBO :

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

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

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

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

Vous êtes autorisé :

- 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.

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

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de plus de 12 ans 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 !

Contents

Licence Creative Commons BY-NC-SA 2.0 FR 5

1 Licence Creative Commons CC-BY-NC-SA 10

2 Sauvegardes avancées 11
2.1 Introduction 11
2.2 Définir une politique de sauvegarde 12
2.3 Sauvegardes logiques 18
2.4 Sauvegardes physiques à froid 32
2.5 Sauvegardes physiques à chaud et PITR 35
2.6 Écriture d'un script de sauvegarde 50
2.7 Écriture d'un script de restauration 55
2.8 Conclusion 58
2.9 Travaux Pratiques 59

3 PostgreSQL : Outils de sauvegarde 70
3.1 Introduction 70
3.2 pg_back - Présentation 72
3.3 pg_basebackup - Présentation 72
3.4 Barman - Présentation générale 75
3.5 pitrery - Présentation générale 94
3.6 Autres outils de l'écosystème 110
3.7 Conclusion 113
3.8 Travaux Pratiques 114

4 PostgreSQL : Gestion d'un sinistre 128
4.1 Introduction 128
4.2 Anticiper les désastres 129
4.3 Réagir aux désastres 133
4.4 Rechercher l'origine du problème 142
4.5 Outils 151
4.6 Cas type de désastres 160
4.7 Conclusion 166
4.8 Travaux Pratiques 166

1 LICENCE CREATIVE COMMONS CC-BY-NC-SA

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

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

Cette formation (diapositives, manuels et travaux pratiques) est sous licence **CC-BY-NC-SA**.

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

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

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre).

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

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

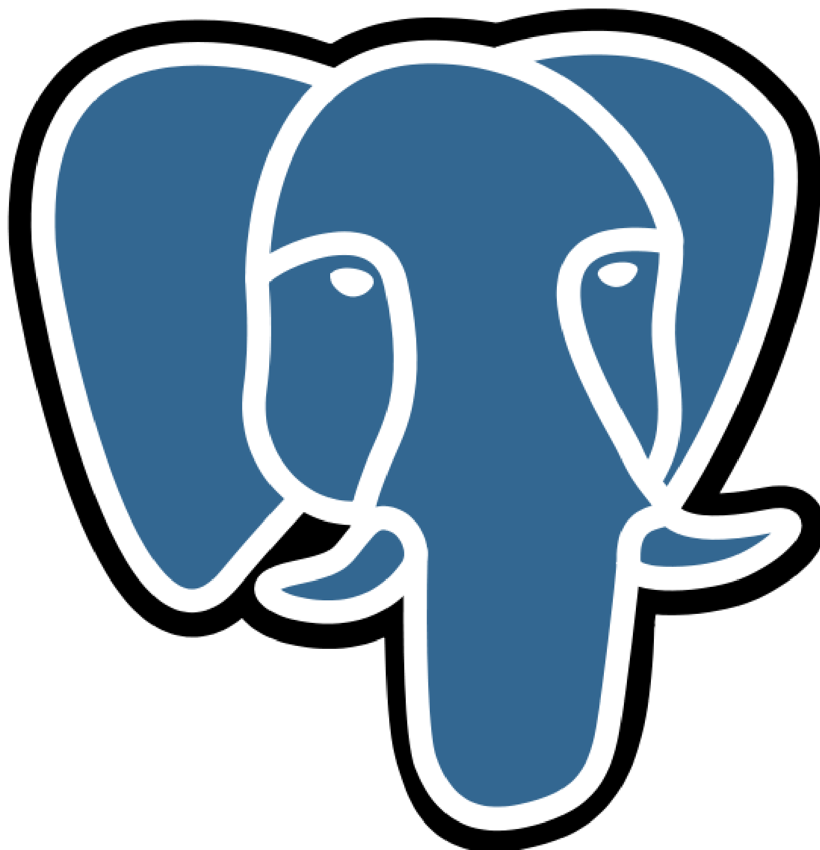
À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web.

Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre.

Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

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

2 SAUVEGARDES AVANCÉES



2.1 INTRODUCTION

- Opération essentielle de sécurisation des données
- Présenter les techniques de sauvegardes disponibles
- Créer une politique de sauvegarde

Afin de se prémunir des pertes de données, il est primordial de sauvegarder régulièrement les données. Selon les besoins de chacun, plusieurs techniques peuvent être utilisables.

Tout dépend des contraintes matérielles, de volumétrie et de qualité de service.

Après une discussion sur la mise au point d'une politique de sauvegardes, cette présentation décrit l'utilisation, les avantages et les inconvénients des outils de sauvegardes disponibles avec PostgreSQL. L'accent est enfin mis sur la production de scripts de sauvegarde et restauration sûrs et efficaces.

2.1.1 AU MENU

- Stratégies de sauvegarde et restauration
- Sauvegardes logiques et restauration
- Sauvegardes physiques et restauration
- Scripts de sauvegarde et de restauration

2.1.2 OBJECTIFS

- Choisir les méthodes de sauvegarde les mieux adaptées à son contexte
- Maîtriser la sauvegarde logique (dump/restore)
- Maîtriser la sauvegarde à chaud et le PITR

Ce module s'adresse aux administrateurs de base de données qui ont déjà eu un premier contact avec les techniques de sauvegardes de PostgreSQL, et qui souhaitent approfondir la question par la pratique.

À l'issue de ce chapitre, un administrateur disposera de tous les éléments nécessaires pour produire des scripts de sauvegarde, afin d'intégrer la sauvegarde des bases de données PostgreSQL dans son SI.

2.2 DÉFINIR UNE POLITIQUE DE SAUVEGARDE

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

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

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

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

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

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

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

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

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

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

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

2.2.1 OBJECTIFS

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

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

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

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

2.2.2 DIFFÉRENTES APPROCHES

- Sauvegarde à froid des fichiers
  - aussi appelée sauvegarde physique
- Sauvegarde à chaud en SQL
  - aussi appelée sauvegarde logique
- Sauvegarde à chaud des fichiers (PITR)

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

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

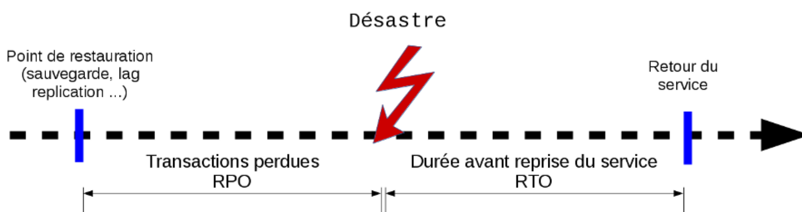
La sauvegarde logique permet de créer un fichier texte de commandes SQL ou un fichier binaire contenant le schéma et les données de la base de données.

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

Suivant les prérequis et les limitations de chaque méthode, il est fort possible qu'une seule de ces solutions soit utilisable. Par exemple, si le serveur ne peut pas être arrêté la sauvegarde à froid est exclue d'office, si la base de données est très volumineuse la sauvegarde logique devient très longue, si l'espace disque est limité et que l'instance génère beaucoup de journaux de transactions, la sauvegarde PITR sera difficile à mettre en place.

2.2.3 RTO/RPO

- `RPO` (Recovery Point Objective)
  - Perte de Données Maximale Admissible
- `RTO` (Recovery Time Objective)
  - Durée Maximale d'Interruption Admissible
- Définissent la politique de sauvegarde/restauration



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

Avec un RPO faible, la perte de données admissible est très faible, voire nulle. Il faudra alors s'orienter vers des solutions de type sauvegarde à chaud, PITR, réplication (asynchrone/synchrone).

Avec un RPO important, on s'autorise une perte de données importante. Dans ce cas, on peut utiliser des solutions de type sauvegarde logique (dump) ou sauvegarde de fichiers à froid.

Un RTO court implique une durée d'interruption courte. Le service doit vite remonter. Cela nécessite des procédures avec le moins de manipulations possible et réduisant le nombre d'acteurs. Les solutions habituelles sont la réplication et les solutions de haute disponibilité.

Quant à un RTO long, la durée de reprise du service n'est pas critique, ce qui fait qu'on peut tranquillement utiliser des solutions simple comme les sauvegardes physiques de

fichiers et les sauvegardes logiques (dump).

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

2.2.4 INDUSTRIALISATION

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

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

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

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

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

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

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

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

2.2.5 DOCUMENTATION

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

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

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

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

2.2.6 POINTS D'ATTENTION

- Externaliser les sauvegardes
- Stocker plusieurs copies
  - de préférence dans des endroits différents
- Sauvegarder les fichiers de configuration
- Tester la restauration

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

Il faut donc externaliser les sauvegardes, l'idéal étant de stocker des copies sur des machines différentes, voire elles-mêmes physiquement dans des lieux différents. Pour limiter la consommation d'espace disque dans ce cas, on peut appliquer une rétention différente, par exemple conserver uniquement la dernière sauvegarde sur la machine, les 5 dernières sur bandes, et un archivage des bandes vers un site sécurisé tous les mois.

Ensuite, la sauvegarde ne concerne pas uniquement les données. Il est également fortement conseillé de sauvegarder les fichiers de configuration du serveur et les scripts d'administration. L'idéal est de les copier avec les sauvegardes. On peut également déléguer cette tâche à une sauvegarde au niveau système, vu que ce sont de simples fichiers. Les principaux fichiers à prendre en compte sont `postgresql.conf`, `postgresql.auto.conf`, `pg_hba.conf`, `pg_ident.conf`, ainsi que `recovery.conf` pour les serveurs répliqués. Cette liste n'est en aucun cas exhaustive.

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

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

2.3 SAUVEGARDES LOGIQUES

- Extraction à chaud des données dans un fichier
- Photo des données au début de l'opération
  - quelle que soit la durée de la sauvegarde
  - restauration à cet état
- Large choix d'options pour sélectionner les données

La sauvegarde logique consiste à se connecter à une base de données pour en extraire le contenu. PostgreSQL fournit les outils pour extraire et charger la structure des données. Dans le jargon PostgreSQL, on nomme ce type d'opération de sauvegarde et le fichier résultant « dump », et la restauration associée « restore ».

Cette partie récapitule l'utilisation des outils intégrés de sauvegarde et restauration logiques, en mettant l'accent sur les critères qui permettront de décider si cette technique est adaptée.

2.3.1 OUTILS

- Dump
  - `pg_dump` extrait le contenu d'une base en texte (SQL) ou binaire
  - `pg_dumpall` extrait une instance en totalité au format texte
- Restore
  - `psql` exécute le SQL des dumps au format texte
  - `pg_restore` restaure un dump binaire dans une base

Les outils de sauvegardes sont différents de ceux de restauration. Le choix de l'outil de restauration d'un fichier se fait en fonction du format du fichier `dump` généré par la sauvegarde.

`pg_dump` permet d'extraire le contenu d'une seule base de données dans différents formats. Pour rappel, une instance PostgreSQL contient plusieurs bases de données.

`pg_dumpall` permet d'extraire le contenu d'une instance en totalité au format texte. Il s'agit des données globales (rôles, tablespaces, configuration par base de données et rôles), de la définition des bases de données et de leur contenu.

`psql` exécute les ordres SQL contenus dans les `dumps` au format texte.

`pg_restore` traite uniquement les `dumps` au format binaire, et produit le SQL qui permet de restaurer les données.

Il est important de bien comprendre que ces outils n'échappent pas au fonctionnement client-serveur de PostgreSQL. Ils « dialoguent » avec l'instance PostgreSQL uniquement en SQL, aussi bien pour le dump que la restore.

2.3.2 FORMATS

Format	Dump	Restore
plain (ou SQL)	`pg_dump -Fp` ou `pg_dumpall`	`psql`
tar	`pg_dump -Ft`	`pg_restore`
custom	`pg_dump -Fc`	`pg_restore`
directory	`pg_dump -Fd`	`pg_restore`

Un élément important est le format des données extraites. Selon l'outil de sauvegarde utilisé et les options de la commande, l'outil de restauration diffère. Le tableau indique les outils compatibles selon le format choisi.

pg_dump accepte d'enregistrer la sauvegarde suivant quatre formats :

- un fichier SQL, donc un fichier texte dont l'encodage dépend de la base ;
- un répertoire disposant du script SQL et des fichiers, compressés avec libz, contenant les données de chaque table ;
- un fichier tar intégrant tous les fichiers décrits ci-dessus mais non compressés ;
- un fichier personnalisé, sorte de fichier tar compressé avec libz.

Pour choisir le format, il faut utiliser l'option --format (ou -F) et le faire suivre par le nom ou le caractère indiquant le format sélectionné :

- plain ou p pour le fichier SQL ;
- tar ou t pour le fichier tar ;
- custom ou c pour le fichier personnalisé ;
- directory ou d pour le répertoire.

À noter que le format directory n'est disponible qu'à partir de la version 9.1.

Le fichier SQL est naturellement lisible par n'importe quel éditeur texte. Le fichier texte est divisé en plusieurs parties :

- configuration de certaines variables ;
- ajout des objets à l'exception des index, des contraintes et triggers (donc schémas, tables, vues, procédures stockées) ;
- ajout des données aux tables ;
- ajout des index, contraintes et triggers ;
- rafraichissement des vues matérialisées ;
- définition des droits d'accès aux objets.

Les index sont créés après l'ajout des données dans les tables pour des raisons de performance. Il est plus rapide de créer un index à partir des données finales que de le mettre à jour en permanence pendant l'ajout des données. Les contraintes le sont parce qu'il faut que les données soient restaurées dans leur ensemble avant de pouvoir mettre en place certaines contraintes comme les clés étrangères. Les triggers ne devant pas être déclenchés pendant la restauration, ils sont aussi restaurés à la fin. Les propriétaires sont restaurés pour chacun des objets.

Voici un exemple de sauvegarde d'une base de 2 Go pour chaque format :

```
$ time pg_dump -Fp b1 > b1.Fp.dump
real   0m33.523s
user   0m10.990s
sys 0m1.625s

$ time pg_dump -Ft b1 > b1.Ft.dump
```

```
real  0m37.478s
user  0m10.387s
sys 0m2.285s

$ time pg_dump -Fc b1 > b1.Fc.dump
real  0m41.070s
user  0m34.368s
sys 0m0.791s

$ time pg_dump -Fd -f b1.Fd.dump b1
real  0m38.085s
user  0m30.674s
sys 0m0.650s
```

La sauvegarde la plus longue est la sauvegarde au format `custom` car elle est compressée. La sauvegarde au format `directory` se trouve entre la sauvegarde au format `custom` et la sauvegarde au format `tar` car elle est aussi compressée mais sur des fichiers plus petits. En terme de taille :

```
$ du -sh b1.F?.dump
116M  b1.Fc.dump
116M  b1.Fd.dump
379M  b1.Fp.dump
379M  b1.Ft.dump
```

Le format compressé est évidemment le plus petit. Le format `plain` et le format `tar` sont les plus lourds à cause du manque de compression. Le format `tar` est même généralement un peu plus lourd que le format `plain` à cause de l'entête des fichiers `tar`.

Le format `tar` produit une véritable archive tar, comme le montre la commande `file` :

```
$ file b1.F?.dump
b1.Fc.dump: PostgreSQL custom database dump - v1.12-0
b1.Fd.dump: directory
b1.Fp.dump: UTF-8 Unicode text, with very long lines
b1.Ft.dump: tar archive
```

Il s'agit d'un format POSIX de tar, et il en partage les limites. Les fichiers intégrés ne peuvent pas dépasser une taille totale de 8 Go, autrement dit il n'est pas possible, avec une sauvegarde au format `tar`, de stocker une table dont la représentation physique au niveau de l'archive tar fait plus de 8 Go. Voici l'erreur obtenue :

```
$ pg_dump -Ft tar > tar.Ft.dump
```

17.12

```
pg_dump: [tar archiver] archive member too large for tar format
```

Cependant, depuis la version 9.5, il est possible de le faire car la plupart des implémentations modernes de tar supportent une extension le permettant. De ce fait, PostgreSQL utilise le format étendu si nécessaire, plutôt que d'échouer.

2.3.3 LIMITES

- Restaurations partielles très difficiles avec le format `plain` (SQL)
- Limite inhérente au format tar POSIX pour les versions antérieures à la 9.5
- Parallélisation du dump seulement possible avec le format `directory` (9.3+)
- Sauvegarde de la définition des objets globaux seulement possible avec `pg_dumpall`
- Pas de support des formats binaires par `pg_dumpall`

=> il faut combiner `pg_dump` et `pg_dumpall` pour avoir la sauvegarde la plus flexible.

Il convient de bien appréhender les limites de chaque outil de dump et des formats.

Tout d'abord, le format `tar` est à éviter. Il n'apporte aucune plus-value par rapport au format `custom` et ajoute la limitation de la taille d'un fichier de l'archive qui ne doit pas faire plus de 8 Go pour les versions antérieures à la 9.5.

Ensuite, même si c'est le plus portable et le seul disponible avec `pg_dumpall`, le format `plain` rend les restaurations partielles difficiles car il faut extraire manuellement le SQL d'un fichier texte souvent très volumineux. On privilégie donc les formats `custom` et `directory` pour plus de flexibilité à la restauration.

Le format `directory` ne doit pas être négligé. Il permet d'utiliser la fonctionnalité de dump en parallèle de `pg_dump`, disponible à partir de PostgreSQL 9.3.

Enfin, l'outil `pg_dumpall`, initialement prévu pour les montées de versions majeures, permet de sauvegarder les objets globaux d'une instance : la définition des rôles et des tablespaces. Ainsi, pour avoir la sauvegarde la plus complète possible d'une instance, il faut combiner `pg_dumpall`, pour obtenir la définition des objets globaux, avec `pg_dump` utilisant le format `custom` ou `directory` pour sauvegarder les bases de données une par une.

2.3.4 AVANTAGES

- Simple et rapide à mettre en œuvre

- Sans interruption de service
- Indépendante de la version de PostgreSQL
- Granularité de sélection à l'objet
- Ne conserve pas la fragmentation des tables et des index

La sauvegarde logique ne nécessite aucune configuration particulière de l'instance hormis l'autorisation de la connexion du client effectuant le dump ou la restore. L'opération de sauvegarde se fait sans interruption de service. Par contre, le service doit être disponible, ce qui n'est pas un problème dans la majorité des cas.

Elle est indépendante de la version du serveur PostgreSQL, source et cible. Le fait que seuls les ordres SQL nécessaires à la création des objets à l'identique permet de s'abstraire du format stockage sur le serveur. De ce fait, la fragmentation des tables et des index disparaît à la restauration.

L'utilisation du dump/restore est d'ailleurs la méthode officielle de montée de version majeure. Même s'il existe d'autres méthodes de migration de version majeure, le dump/restore est le moyen le plus sûr parce que le plus éprouvé.

Un dump ne contenant que les données utiles, sa taille est généralement beaucoup plus faible que la base de données source, sans parler de la compression qui peut encore réduire l'occupation sur disque. Par exemple, seuls les ordres DDL permettant de créer les index sont stockés, leur contenu n'est pas dans le dump, ils sont alors créés de zéro au moment de la restore.

Enfin, les outils de dump/restore, surtout lorsqu'on utilise les format `custom` ou `directory`, permettent au moment du dump comme à celui du restore de sélectionner très finement les objets sur lesquels on travaille.

2.3.5 INCONVÉNIENTS

- Durée d'exécution dépendante des données et de l'activité
- Efficace pour des volumétries inférieures à 200 Go
- Restauration à l'instant du démarrage de l'export uniquement
- Impose d'utiliser plusieurs outils pour sauvegarder une instance complète
- Nécessite de recalculer les statistiques de l'optimiseur, les `FSM` et `VM` des tables à la restauration

L'un des principaux inconvénients du dump/restore concerne la durée d'exécution des opérations. Elle est proportionnelle à la taille de la base de données pour un dump complet, et à la taille des objets choisis pour un dump partiel.

En conséquence, il est généralement nécessaire de réduire le niveau de compression pour les formats `custom` et `directory` afin de gagner du temps. Avec des disques mécaniques en RAID 10, il est généralement nécessaire d'utiliser d'autres méthodes de sauvegarde lorsque la volumétrie dépasse 200 Go.

Le second inconvénient majeur de la sauvegarde logique est l'absence de granularité temporelle. Une « photo » des données est prise au démarrage du dump et on ne peut restaurer qu'à cet instant, quelle que soit la durée d'exécution de l'opération. Il faut cependant se rappeler que cela rend possible la cohérence du contenu du dump d'un point de vue transactionnel.

Comme les objets et leur contenu sont effectivement recréés à partir d'ordres SQL lors de la restauration, on perd la fragmentation des tables et index, mais on perd aussi les statistiques de l'optimiseur, ainsi que certaines méta-données des tables (structures `FSM` et `VM`). Il est donc nécessaire de lancer au moins l'opération de maintenance `VACUUM ANALYZE` sur les objets restaurés.

2.3.6 OPTIONS DE CONNEXION

- `-h` / `$PGHOST` / socket Unix
- `-p` / `$PGPORT` / 5432
- `-U` / `$PGUSER` / utilisateur du système
- `-d` / `$PGDATABASE` / utilisateur de connexion
- Gestion des mots de passe
  - pas d'option en ligne de commande
  - `$PGPASSWORD`
  - `.pgpass`

Les commandes `pg_dump`, `pg_dumpall` et `pg_restore` se connectent au serveur PostgreSQL comme n'importe quel autre outil (psql, pgAdmin, etc.). Ils disposent donc des options habituelles pour se connecter :

- `-h` ou `--host` pour indiquer l'alias ou l'adresse IP du serveur ;
- `-p` ou `--port` pour préciser le numéro de port ;
- `-U` ou `--username` pour spécifier l'utilisateur ;
- `-d` ou `--dbname` pour spécifier la base de données de connexion ;
- `-W` ne permet pas de saisir le mot de passe en ligne de commande. Il force seulement l'outil à demander un mot de passe (en interactif donc).

À partir de la version 10, il est possible d'indiquer plusieurs hôtes et ports. L'hôte sélectionné est le premier qui répond au paquet de démarrage. Si l'authentication ne passe

pas, la connexion sera en erreur. Il est aussi possible de préciser si la connexion doit se faire sur un serveur en lecture/écriture ou en lecture seule. Par exemple on effectuera une sauvegarde depuis le premier serveur disponible ainsi :

```
pg_dumpall -h esclave,maitre -p 5432,5433 -U postgres -f sauvegarde.sql
```

Si la connexion nécessite un mot de passe, ce dernier sera réclamé lors de la connexion. Il faut donc faire attention avec `pg_dumpall` qui va se connecter à chaque base de données, une par une. Dans tous les cas, il est préférable d'utiliser un fichier `.pgpass` qui indique les mots de passe de connexion. Ce fichier est créé à la racine du répertoire personnel de l'utilisateur qui effectue la sauvegarde. Il contient les informations suivantes :

```
hote:port:base:utilisateur:mot de passe
```

Ce fichier est sécurisé dans le sens où seul l'utilisateur doit avoir le droit de lire et écrire ce fichier. L'outil vérifiera cela avant d'accepter d'utiliser les informations qui s'y trouvent.

2.3.7 IMPACT DES PRIVILÈGES

- Les outils se comportent comme des clients pour PostgreSQL
- Préférer un rôle super-utilisateur autorisé dans `pg_hba.conf`
- Dans le cas contraire
  - la connexion à la/aux base(s) de données doit être autorisée
  - le rôle doit pouvoir lire le contenu de tous les objets à exporter

Même si ce n'est pas obligatoire, il est recommandé d'utiliser un rôle de connexion disposant des droits de super-utilisateur pour les opérations de `dump` et `restore`.

En effet, pour le `dump`, il faut pouvoir :

- se connecter à la base de données : autorisation dans `pg_hba.conf`, être propriétaire ou avoir le privilège `CONNECT` ;
- voir le contenu des différents schémas : être propriétaire ou avoir le privilège `USAGE` sur le schéma ;
- lire le contenu des tables : être propriétaire ou avoir le privilège `SELECT` sur la table.

Pour la `restore`, il faut pouvoir :

- se connecter à la base de données : autorisation dans `pg_hba.conf`, être propriétaire ou avoir le privilège `CONNECT` ;
- optionnellement, pouvoir créer la base de données cible et pouvoir s'y connecter (option `-C` de `pg_restore`)

- pouvoir créer des schémas : être propriétaire de la base de données ou avoir le privilège `CREATE` sur celle-ci ;
- pouvoir créer des objets dans les schémas : être propriétaire du schéma ou avoir le privilège `CREATE` sur celui-ci ;
- pouvoir écrire dans les tablespaces cibles : être propriétaire du tablespace ou avoir le privilège `CREATE` sur celui-ci ;
- avoir la capacité de donner ou retirer des privilèges : faire partie des rôles bénéficiant d'`ACL` dans le `dump`.

On remarque que les prérequis en matière de privilèges sont assez conséquents pour la restore. C'est pourquoi il n'est parfois possible de ne restaurer qu'avec un super-utilisateur.

2.3.8 PG_DUMP - OPTIONS

- Base de données : `-d` ou en fin de commande
- Format : `-F` (`p`, `t`, `c`, `d`)
- Fichier de sortie : `-f` chemin, sortie standard sinon
- Sélection : `-a`, `-s`, `-n`, `-N`, `-t`, `-T`, `-O`, `-x`, `--section`
- Compression : `-Z 0-9`
- Parallélisme (format `directory`, 9.3+) : `-j` jobs

L'ensemble des options disponibles pour pg_dump est disponible dans son aide en ligne, disponible avec l'option `--help` :

```
pg_dump --help
```

Les options de `pg_dump` sont très riches. Il est possible en les combinant de faire des extractions de données très fines. Il y a cependant des points d'attention :

- la sélection d'une table avec `-t` ignore les options `-n` et `-N`
- pour sélectionner une table dans un schéma précis, il faut la préfixer du nom du schéma : `-t schema.table`
- la sélection d'une table dépend du `search_path` du rôle de connexion

En dehors de cette subtilité sur les schémas, les options se combinent bien pour affiner une sélection. Il est possible d'indiquer plusieurs fois les options `-n`, `-t`, `-N` ou `-T`, pour sélectionner ou exclure plusieurs objets.

2.3.9 PG_DUMPALL - OPTIONS

- Limiter le dump aux données globales (rôles et tablespaces) : `-g`
- Limiter le dump aux rôles : `-r`
- Limiter le dump aux tablespaces : `-t`
- Sélection : `-a`, `-s`, `-O`, `-x`

L'ensemble des options disponibles pour pg_dumpall est disponible dans son aide en ligne, disponible avec l'option `--help` :

```
pg_dumpall --help
```

Les options de sélection de pg_dumpall sont plus limitées que celles de pg_dump. Cependant, ce sont généralement les options relatives aux objets globaux qui sont intéressantes :

- `-r` pour ne garder que la définition des rôles ;
- `-t` pour ne garder que la définition des tablespaces ;
- `-g` qui combine `-r` et `-t`.

Dans le cas des rôles, leur définition est sauvegardée en deux ordres SQL pour palier les erreurs sur les rôles existants (typiquement "postgres", le nom le plus fréquemment utilisé comme super-utilisateur par défaut) :

```
CREATE ROLE postgres;
ALTER ROLE postgres WITH SUPERUSER INHERIT CREATEROLE CREATEDB LOGIN REPLICATION
                         PASSWORD 'md503391226bda56b94d01b4948952511c1';
```

On remarque également que le mot de passe est sauvegardé sous forme de hash si c'est le cas dans l'instance.

2.3.10 PSQL

- Client standard capable d'exécuter du SQL au format texte (`plain`)
- `-f` permet de spécifier l'emplacement du fichier dump
- `-1` permet d'exécuter la restauration en une transaction
  - `-v ON_ERROR_ROLLBACK=ON` : faire un rollback en cas d'erreur et continuer la restauration
  - `-v ON_ERROR_STOP=ON` : arrêter l'exécution à la première erreur (rollback complet)

psql étant le client standard de PostgreSQL, le dump au format `plain` se trouve être un script SQL qui peut également contenir des commandes psql, comme `\connect` pour se

connecter à une base de données (ce que fait pg_dumpall pour changer de base de données).

On bénéficie alors de toutes les options de psql, les plus utiles étant celles relatives au contrôle de l'aspect transactionnel de l'exécution.

Par défaut, psql ne crée pas de transaction explicite pour la restauration. Il traite le dump comme n'importe quel script SQL, chaque ordre est donc lancé en autocommit, soit une transaction par ordre.

Utiliser l'option `-1` (« moins un ») permet de lancer la restore dans une transaction. On retrouve alors le côté atomique : l'opération ne doit produire aucune erreur pour être validée. Par défaut, psql continue même en cas d'erreur, ce qui fait que les erreurs s'accumulent : après une erreur, tout nouvel ordre dans la transaction sort en erreur parce que la transaction est elle-même en erreur.

Des options permettent de contrôler le comportement en cas d'erreur :

- `-v ON_ERROR_ROLLBACK=ON` : psql crée un savepoint avant chaque ordre du script, et peut donc annuler un ordre en erreur et continuer la restauration.
- `-v ON_ERROR_STOP=ON` : psql annule la transaction et s'arrête. Cela permet d'identifier l'ordre en erreur immédiatement.

2.3.11 PG_RESTORE - OPTIONS

- **Attention** : `-f` indique un fichier de sortie
  - Le dump est indiqué en fin de ligne de commande
  - Entrée standard sinon
- Format : `-F` (`t`, `c`, `d`), détecté automatiquement
- Sélection : `-a`, `-I`, `-n`, `-O`, `-P`, `-s`, `-t`, `-T`, `-x`, `--section`
- Transaction : `-1`

L'ensemble des options disponibles pour pg_restore est disponible dans son aide en ligne, disponible avec l'option `--help` :

```
pg_restore --help
```

pg_restore ne fonctionne qu'avec les dump aux formats `tar`, `custom` et `directory`.

Les options de sélection de `pg_restore` sont similaires à celles de `pg_dump`, avec en plus la possibilité de restaurer une fonction ou un trigger précis. Dans le cas d'une fonction, il faut fournir le prototype complet : nom et type des arguments dans l'ordre sous la forme `nom(arguments)`.

Comme pour psql, il est possible de réaliser l'opération dans une seule transaction. Cependant, on ne peut pas contrôler le comportement de `pg_restore` en cas d'erreur : l'exécution stoppe immédiatement en cas d'erreur.

Enfin, l'option `-f` de pg_restore est **un faux ami**. Comme pour `pg_dump`, `-f` indique le fichier de sortie, ce qui signifie dans le cas de `pg_restore` les messages affichés lors de l'opération, **et non pas le fichier** `dump`. Le fichier dump doit être spécifié comme dernier élément de la ligne de commande.

2.3.12 PG_RESTORE - BASE DE DONNÉES

- `-d` indique la base de données de connexion
- Avec `-C` (créer la base de données cible)
  - `pg_restore` se connecte (`-d`) et exécute `CREATE DATABASE`
  - `pg_restore` se connecte à la nouvelle base et exécute le SQL
- Sans `-C`
  - pg_restore se connecte (`-d`) et exécute le SQL
- Sans `-d`
  - `pg_restore` affiche le SQL (permet de débugger)

Avec `pg_restore`, il est indispensable de fournir le nom de la base de données de connexion avec l'option `-d`. Dans le cas contraire, le SQL transmis au serveur est affiché sur la sortie standard, ce qui est très pratique pour valider les options d'une restauration partielle.

Comme l'option `-C` permet créer la base de données cible, cela peut provoquer une confusion avec l'option `-d`. Si `-C` est spécifié, l'option `-d` doit indiquer une base de données existante afin que pg_restore se connecte pour exécuter l'ordre `CREATE DATABASE`. Après cela, il se connecte à la base nouvellement créée pour exécuter les ordres SQL de restauration. Pour vérifier cela, on peut lancer la commande sans l'option `-d`, en observant le code SQL renvoyé on remarque un `\connect` :

```
$ pg_restore -C b1.dump
--
-- PostgreSQL database dump
--

SET statement_timeout = 0;
SET lock_timeout = 0;
SET client_encoding = 'UTF8';
```

```
17.12

SET standard_conforming_strings = on;
SET check_function_bodies = false;
SET client_min_messages = warning;

--
-- Name: b1; Type: DATABASE; Schema: -; Owner: postgres
--

CREATE DATABASE b1 WITH TEMPLATE = template0 ENCODING = 'UTF8'
                        LC_COLLATE = 'en_US.UTF-8' LC_CTYPE = 'en_US.UTF-8';


ALTER DATABASE b1 OWNER TO postgres;

\connect b1

SET statement_timeout = 0;
-- Suite du dump…
```

Il n'est par contre pas possible de restaurer dans une base de données ayant un nom différent de la base de données d'origine avec l'option `-C`.

2.3.13 PG_RESTORE - TABLE DES MATIÈRES

- Pour les sélections plus complexes
  - obtenir la table des matières avec `-l`
  - choisir les éléments à restaurer
  - fournir la liste avec `-L liste.txt`
- Les dépendances doivent être respectées

Quand il faut restaurer beaucoup d'objets, cela devient difficile d'utiliser les options de ligne de commande. `pg_restore` fournit un moyen avancé pour sélectionner les objets.

L'option `-l` (`--list`) permet de connaître la liste des actions que réalisera pg_ restore avec un dump particulier. Par exemple :

```
$ pg_restore -l b1.dump
;
; Archive created at 2017-08-28 15:39:27 CEST
;     dbname: b1
```


DALIBO

```
;     TOC Entries: 15
;     Compression: -1
;     Dump Version: 1.12-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 10beta3
;     Dumped by pg_dump version: 10beta3
;
;
; Selected TOC Entries:
;
3116; 1262 32872 DATABASE - b1 postgres
3; 2615 2200 SCHEMA - public postgres
3117; 0 0 COMMENT - SCHEMA public postgres
1; 3079 13232 EXTENSION - plpgsql
3118; 0 0 COMMENT - EXTENSION plpgsql
207; 1255 32888 FUNCTION public f1() postgres
196; 1259 32873 TABLE public t1 postgres
197; 1259 32879 TABLE public t2 postgres
198; 1259 32889 VIEW public v1 postgres
3110; 0 32873 TABLE DATA public t1 postgres
3111; 0 32879 TABLE DATA public t2 postgres
2987; 2606 32886 CONSTRAINT public t2 t2_pkey postgres
2985; 1259 32887 INDEX public t1_c1_idx postgres
```

Toutes les lignes qui commencent avec un point-virgule sont des commentaires. Le reste indique les objets à créer : un schéma public, le langage `plpgsql`, la procédure stockée `f1`, les tables `t1` et `t2`, la vue `v1`, la clé primaire sur `t2` et l'index sur `t1`. Il indique aussi les données à restaurer avec des lignes du type « TABLE DATA ». Donc, dans cette sauvegarde, il y a les données pour les tables `t1` et `t2`.

Il est possible de stocker cette information dans un fichier, de modifier le fichier pour qu'il ne contienne que les objets que l'on souhaite restaurer, et de demander à pg_restore, avec l'option `-L` (`--use-list`), de ne prendre en compte que les actions contenues dans le fichier. Voici un exemple complet :

```
$ pg_restore -l b1.dump > liste_actions

$ cat liste_actions | \
  grep -v "f1" | \
```

```
  grep -v "TABLE DATA public t2" | \
  grep -v "INDEX public t1_c1_idx" \
  > liste_actions_modifiee

$ createdb b1_new

$ pg_restore -L liste_actions_modifiee -d b1_new -v b1.dump
pg_restore: connecting to database for restore
pg_restore: creating SCHEMA "public"
pg_restore: creating COMMENT "SCHEMA public"
pg_restore: creating EXTENSION "plpgsql"
pg_restore: creating COMMENT "EXTENSION plpgsql"
pg_restore: creating TABLE "public.t1"
pg_restore: creating TABLE "public.t2"
pg_restore: creating VIEW "public.v1"
pg_restore: processing data for table "public.t1"
pg_restore: creating CONSTRAINT "public.t2 t2_pkey"
```

L'option `-v` de pg_restore permet de visualiser sa progression dans la restauration. On remarque bien que la procédure stockée `f1` ne fait pas partie des objets restaurés, tout comme l'index sur `t1` et les données de la table `t2`.

Enfin, il est à la charge de l'utilisateur de fournir une liste cohérente en terme de dépendances. Par exemple, sélectionner seulement l'entrée `TABLE DATA` alors que la table n'existe pas dans la base de données cible provoquera une erreur.

2.4 SAUVEGARDES PHYSIQUES À FROID

- Une sauvegarde à froid impose l'arrêt de l'instance
- Outils système à utiliser pour sauvegarder et restaurer
- L'instance complète doit être sauvegardée :
  - PGDATA
  - tous les tablespaces
  - journaux de transactions (fichiers WAL)
  - fichiers de configuration

Il est possible de sauvegarder une instance PostgreSQL avec n'importe quel outil de sauvegarde de fichiers, tout simplement parce que PostgreSQL stocke les données dans de simples fichiers.

L'impératif pour ce type de sauvegarde est d'être réalisé à froid, c'est-à-dire **avec l'instance arrêtée**.

Pour réaliser une sauvegarde complète, il faut prendre en compte :

- le répertoire principal de données (`$PGDATA`) ;
- le répertoire pg_wal, s'il se situe en dehors de `$PDATA` ;
- l'ensemble des tablespaces ;
- les fichiers de configuration, s'ils se situent en dehors de `$PGDATA`.

Les éléments précédents forment un tout indissociable, il faut tout sauvegarder et par conséquent tout restaurer.

2.4.1 OUTILS

- Outils au niveau système de fichiers
- Non spécifiques à PostgreSQL
- Pour réduire la durée d'interruption de service :
  - rsync à chaud, puis rsync à froid
  - snapshots des systèmes de fichiers

Les outils utilisables pour une sauvegarde à froid sont tous ceux qui permettent de sauvegarder une arborescence en conservant les liens symboliques. Cela va de `cp` à des solutions de sauvegardes centralisées comme Bacula, en passant par `rsync`. Il ne s'agit pas d'outils spécifiques à PostgreSQL.

Comme la sauvegarde doit être effectuée avec l'instance arrêtée, la durée de l'arrêt est dépendante du volume de données à sauvegarder. On peut optimiser les choses en réduisant le temps d'interruption avec l'utilisation de snapshots au niveau système de fichier ou avec `rsync`.

Pour utiliser des snapshots, il faut soit disposer d'un SAN offrant cette possibilité ou bien utiliser la fonctionnalité du LVM. Dans ce cas, la procédure est la suivante :

- arrêt de l'instance PostgreSQL ;
- création des snapshots de l'ensemble des systèmes de fichiers ;
- démarrage de l'instance PostgreSQL ;
- sauvegarde des fichiers à partir des snapshots ;
- destruction des snapshots.

Si on n'a pas la possibilité d'utiliser des snapshots, on peut utiliser rsync de cette manière :

- rsync de l'ensemble des fichiers de l'instance PostgreSQL alors qu'elle est démarrée ;
- arrêt de l'instance PostgreSQL ;
- rsync de l'ensemble des fichiers de l'instance pour ne transférer que les différences ;
- démarrage de l'instance PostgreSQL.

2.4.2 AVANTAGES

- Simple et rapide
- De nombreux outils disponibles
- Efficace pour les fortes volumétries

Cette méthode de sauvegarde est facile à mettre en place, rapide à la sauvegarde et à la restauration, en comparaison des autres techniques.

En effet, il suffit de remettre les fichiers de la sauvegarde en place et de lancer PostgreSQL, les index sont conservés et aucune configuration supplémentaire n'est nécessaire.

2.4.3 INCONVÉNIENTS

- Arrêt de la production
- Sauvegarde de l'instance complète à un instant t
- Conservation de la fragmentation
- Impossible de changer d'architecture ou de version

L'inconvénient majeur de cette méthode est la nécessité d'arrêter l'instance. Ce n'est pas forcément possible selon le contexte.

Ensuite, il n'y a aucune granularité sur les éléments présents dans la sauvegarde, on est obligé de tout sauvegarder et de tout restaurer. On conserve donc la fragmentation des tables et index, ce qui prend de l'espace supplémentaire.

Enfin, les fichiers d'une instance PostgreSQL sont liés à la version majeure du moteur de base de données et à l'architecture du processeur (`32/64 bits`, `endianness`).

Ces inconvénients n'existent pas avec la sauvegarde logique, et l'interruption de service n'est pas nécessaire avec le PITR.

2.5 SAUVEGARDES PHYSIQUES À CHAUD ET PITR

- Sauvegarde en deux parties :
  - les fichiers de données
  - les journaux de transactions archivés
- PostgreSQL archive les journaux de transactions
- L'administrateur réalise une copie des fichiers de données
- *Point In Time Recovery* : on applique les transactions archivées jusqu'à un point donné

La sauvegarde PITR est découpée en deux parties :

- la copie des fichiers de données (sans oublier les tablespaces)
- l'archivage des journaux de transactions

La copie des fichiers de données de l'instance est réalisée par l'administrateur selon la procédure de *basebackup* (tout à fait automatisable). L'archivage des journaux de transactions est effectué automatiquement par l'instance, l'administrateur le contrôle dans la configuration de PostgreSQL. Les fichiers sont alors archivés selon l'activité en écriture sur l'instance.

La restauration *Point In Time Recovery* consiste à appliquer les données des journaux de transactions sur des fichiers de données plus anciens afin d'atteindre un point précis dans le temps, matérialisé par le commit d'une transaction. Cela permet par exemple de restaurer les données immédiatement avant une transaction responsable d'une erreur métier dans les données, en arrêtant la restauration à la transaction précédente. On configure alors le mode recovery de PostgreSQL pour piloter le rejeu des transactions.

2.5.1 LE MODE RECOVERY

- PostgreSQL écrit les données deux fois :
  - d'abord dans les journaux de transactions (fichiers WAL)
  - puis dans les fichiers de données de manière asynchrone
- En cas d'arrêt brutal, les journaux de transactions sont rejoués sur les fichiers de données
  - il s'agit du mode `recovery`
  - le rejeu des transactions permet de retourner à l'état cohérent
- Le contrôle du mode `recovery` permet le PITR

PostgreSQL sauvegarde les données modifiées par les transactions en écriture dans ses journaux de transactions. Ensuite, les données des journaux de transactions sont écrites

dans les fichiers de données, de manière asynchrone. Ce fonctionnement permet d'avoir de meilleures performances en écriture que s'il fallait écrire directement dans les fichiers de données.

Grâce aux journaux de transactions, PostgreSQL peut rejouer des transactions sur des fichiers de données. Ils servent à réparer les fichiers de données en cas d'arrêt brutal de l'instance (dont le cache contient probablement des données modifiées et non synchronisées sur disque à ce moment là).

En disposant des fichiers de données à un instant `t` et des journaux de transactions contenant les écritures ultérieures, on peut aller à un point `t + n` en appliquant `n` transactions. Il s'agit des éléments de la sauvegarde PITR.

La capacité des journaux de transactions à « réparer » des fichiers de données partiellement écrits, permet de réaliser la copie de ceux-ci à chaud, c'est-à-dire sans interruption de service. La restauration d'une sauvegarde de ce type se déroule donc en trois phases :

- restauration des fichiers de données copiés alors qu'ils étaient modifiés, donc dans un état incohérent.
- retour à l'état cohérent, correspondant au rejeu des transactions jusqu'à la fin du *basebackup*.
- rejeu des transactions jusqu'au point dans le temps configuré ou épuisement des archives disponibles.

2.5.2 ARCHIVAGE DES JOURNAUX DE TRANSACTION

- Choisir le répertoire d'archivage
- Configurer PostgreSQL, dans `postgresql.conf` :
  - `wal_level` : `replica` ou `logical`
  - `archive_mode` : `on` ou `always`
  - `archive_command` : texte de la commande
  - `archive_timeout` : temps en secondes
- Redémarrer l'instance si `wal_level` et `archive_mode` ont changé, recharger sinon

Comme PostgreSQL gère l'archivage des journaux de transactions lui-même, tout se fait dans la configuration, en modifiant le fichier `postgresql.conf`.

Le fonctionnement de l'archivage est simple : lorsqu'on l'active, le processus `archiver`, dédié à cette tâche, exécute la commande d'archivage configurée dès qu'un segment de journal de transactions est rempli. Les journaux de transactions sont archivés dans l'ordre,

ils constituent une chaîne qui ne peut être brisée, c'est pourquoi l'`archiver process` n'abandonne jamais l'archivage d'un fichier s'il est en erreur.

Pour configurer l'archivage, il faut modifier le paramètre `wal_level` (9.0+), qui indique le niveau d'information présent dans les journaux de transactions, à une valeur parmi `replica` ou `logical` (9.4+). La valeur `logical` inclut `replica`. Les versions antérieures à la 9.6 avaient deux valeurs intermédiaires à la place de `replica` : `archive` et `hot_standby`. Dans le cas de la sauvegarde PITR, ces deux valeurs sont bonnes.

Ensuite, il faut activer l'archivage en plaçant `archive_mode` à `on`. Depuis la version 9.5, la valeur `always` est aussi possible pour permettre un archivage à partir des serveurs secondaires.

La commande exécutée par l'*archiver process* est configurée avec le paramètre `archive_command`. Il est nécessaire de déterminer où PostgreSQL doit placer les fichiers archivés. Le répertoire de destination se configure par la commande d'archivage. Cette commande est exécutée par le serveur PostgreSQL en utilisant `/bin/sh`. À chaque exécution, les motifs `%f` et `%p` sont respectivement remplacés par le nom du segment de journal de transaction à archiver, et le chemin complet de ce fichier. Par exemple, avec la commande `cp` :

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

Pour le fichier `000000010000000E0000001F`, la commande sera transformée en :

```
cp pg_wal/000000010000000E0000001F \
   /var/lib/pgsql/archives/000000010000000E0000001F
```

Il est tout à fait possible de copier le fichier sur une autre machine, en utilisant `scp` ou `rsync`. La commande peut également être un script.

Dans le choix de l'outil système pour archiver, **il est primordial de ne jamais supprimer le fichier source**. En effet, on manipule directement les fichiers de `pg_wal`. La gestion des fichiers de ce répertoire (ajout, purge, rotation) est faite par PostgreSQL, la commande d'archivage ne doit pas interférer.

Afin de déterminer si la commande s'est correctement exécutée, le code retour est vérifié : un code retour égal à 0 indique le succès, un code retour différent de 0 indique l'échec. En cas d'échec, l'`archiver process` tente indéfiniment la commande, par des cycles trois tentatives suivies d'une pause d'une minute.

Le paramètre `archive_timeout` permet de forcer l'archivage au bout d'un certain délai (sa valeur en secondes) si l'activité en écriture de l'instance est trop faible. Ainsi, en cas de faible activité, on peut configurer le temps maximum de perte de données avec ce paramètre. En contrepartie, la consommation d'espace disque est plus élevée car il y a

plus de chance d'archiver des journaux de transactions partiellement remplis, leur taille étant fixée à 16 Mo quel que soit leur contenu.

Enfin, la modification des paramètres `wal_level` et `archive_mode` nécessite un redémarrage de l'instance, les autres un simple rechargement de la configuration.

2.5.3 SAUVEGARDE À CHAUD / BASEBACKUP

- L'archivage doit fonctionner sans erreur
- Se connecter et exécuter `SELECT pg_start_backup('label', true);`
- Copier l'ensemble des fichiers de l'instance :
  - fichiers de données ($PGDATA) et de configuration
  - tablespaces
- Mais ignorer :
  - fichier postmaster.pid
  - répertoires log, pg_wal, pg_replslot
- Se connecter et exécuter `SELECT pg_stop_backup();`

La sauvegarde à chaud ou *basebackup* nécessite un archivage opérationnel. Dans cette étape, on copie les données alors que l'instance s'exécute, ce qui signifie que la copie est incohérente : le seul moyen de s'en sortir lors de la restauration est d'appliquer les journaux de transactions qui permettront de retrouver la cohérence des données. C'est pourquoi l'archivage de journaux de transactions doit être en place et fonctionner sans erreur lors de l'exécution de la procédure de *basebackup* - faute de quoi la sauvegarde ne peut être considérée comme valide.

La première étape consiste à placer l'instance en « mode backup », en exécutant la fonction `pg_start_backup()`. Cette fonction prend en premier argument une chaîne de caractères libre, qui permet de nommer le backup. Le second paramètre, optionnel, est un booléen ; à la valeur `true`, un checkpoint est forcé, à la valeur `false`, la fonction attend la fin de l'exécution du prochain checkpoint. Un checkpoint est un point sûr dans les journaux de transactions : toutes les données le précédant ont obligatoirement été écrites dans les fichiers de données. Lors de la restauration, ce point servira comme référence au démarrage du `recovery`, permettant de repartir de ce point pour appliquer les journaux archivés. L'adresse de ce checkpoint, attendu ou forcé par `pg_start_backup()` est sauvegardée dans le fichier `$PGDATA/backup_label`. Un troisième paramètre, optionnel lui-aussi, apparaît avec la version 9.6. Il permet d'indiquer si on autorise les sauvegardes concurrentes. Ce n'est pas le cas par défaut, et nous conseillons de l'autoriser qu'en cas de besoin réel. Dans le cas où les sauvegardes concurrentes sont autorisées, le

fichier `$PGDATA/backup_label` ne sera pas créé. Son contenu sera renvoyé par la fonction d'arrêt du mode backup.

Ensuite, il faut copier l'ensemble des fichiers de l'instance, à l'exception de `pg_wal`. En effet, puisque la restauration utilisera sur les journaux de transactions archivés, le contenu de ce répertoire n'est pas nécessaire. D'autres répertoires ne sont pas nécessaires. En dehors de `pg_wal` et de `log`, il est conseillé de sauvegarder tous les autres répertoires de `$PGDATA`. En plus du répertoire `$PGDATA`, il ne faut pas oublier les tablespaces et les fichiers de configuration s'ils se trouvent en dehors de `$PGDATA`.

Enfin, il ne reste plus qu'à exécuter `pg_stop_backup()` pour indiquer à PostgreSQL que la sauvegarde est terminée. Elle n'accepte aucun argument avant la version 9.6. Un argument optionnel à partir de la 9.6 permet d'indiquer si la sauvegarde était concurrente. La fonction permet de s'assurer que tous les journaux de transactions créés depuis l'exécution de `pg_start_backup()` ont bien été archivés, et elle ne rendra pas la main tant que ce n'est pas le cas. Cela permet d'être sûr que l'instance pourra retourner à l'état cohérent au moment de la restauration. Le fichier `$PGDATA/backup_label` est alors archivé sous le nom du segment de journal de transaction contenant l'adresse du checkpoint du début de la sauvegarde et suffixé par l'adresse du début du backup et `.backup`.

Attention, dans le cas d'une sauvegarde concurrente, les fonctions `pg_start_backup()` et `pg_stop_backup()` doivent être exécutées au sein de la même session.

2.5.4 RESTAURATION PITR - FICHIERS DE DONNÉES

- Restaurer les fichiers et répertoires suivants :
  - $PGDATA
  - les tablespaces (mettre à jour `$PGDATA/pg_tblspc` si nécessaire)
  - fichiers de configuration
- Ne **pas** restaurer les fichiers suivants :
  - fichiers WAL de l'instance, i.e. `$PGDATA/pg_wal` (on restaurera les WAL archivés)
  - fichiers `postmaster.pid` et `postmaster.opts`
  - traces si elles sont incluses

La première étape de restauration est de copier les fichiers de l'instance sauvegardés vers le PGDATA cible. Les fichiers de données contenus dans des répertoires différents du PGDATA (comme des éventuels tablespaces) doivent être également restaurés, dans des répertoires dont le chemin est identique à celui d'origine. Il est possible de modifier le chemin des tablespaces :

- il faut mettre à jour les liens symboliques contenus dans le répertoire `$PGDATA/pg_tblspc` pour qu'ils pointent sur les répertoires des tablespaces restaurés ;
- pour les versions inférieures à la 9.2, il faut également mettre à jour la table `pg_catalog.pg_tablespaces` à la fin de la restauration.

La méthode de restauration doit être adaptée à la méthode utilisée pour copier les fichiers lors de la sauvegarde. Si la sauvegarde est effectuée avec `tar` suivi d'une compression, il faudra également utiliser `tar` et l'outil approprié pour décompresser l'archive. Si des outils de snapshot spécifiques ont été utilisés pour la sauvegarde (système de fichiers, virtualisation, baie), alors la restauration devra faire appel à ces mêmes outils.

Il convient de faire attention aux droits et au propriétaire des répertoires dans lesquels les données sont restaurées, ainsi que sur les fichiers eux-mêmes. Notamment, le répertoire PGDATA de l'instance doit avoir comme droits 700, et appartenir à l'utilisateur système démarrant l'instance (généralement `postgres`).

À la fin de cette étape, l'instance est restaurée mais est dans un état non cohérent : il reste à restaurer les fichiers WAL archivés qui sont **indispensables** pour que l'instance retrouve un état cohérent au démarrage.

Les fichiers WAL qui étaient en cours d'écriture au moment de la sauvegarde (répertoire `pg_wal`) ne sont pas nécessaires à la restauration, ils ne peuvent pas être utilisés pour retrouver un état cohérent. Ce sont les fichiers WAL **archivés** qui seront utilisés. Par sécurité, il est préférable d'éviter de restaurer ces fichiers, voire de les exclure de la sauvegarde elle-même.

Les fichiers `postmaster.pid` et `postmaster.opts` ne devraient pas non plus être restaurés. En effet, ces fichiers permettent d'indiquer le PID d'un processus PostgreSQL actif et les options utilisées lors du démarrage. La présence de ces fichiers n'a pas d'intérêt, et pourrait empêcher le démarrage de l'instance.

2.5.5 RESTAURATION PITR - RECOVERY.CONF

- PostgreSQL restaure lui-même les WAL archivés au démarrage
- L'instance doit pouvoir accéder aux fichiers WAL archivés
- Créer le fichier `$PGDATA/recovery.conf` :
  - `restore_command`
  - recovery target settings : `recovery_target_*` parameters
  - Standby server settings

L'étape suivante consiste à restaurer les fichiers WAL nécessaires à la restauration de l'instance. Au minimum, tous les fichiers WAL archivés entre le début et la fin de la sauvegarde **doivent** être présents. La séquence de ces fichiers peut être trouvée dans le fichier `.backup` généré (et archivé) à la fin de la sauvegarde dans le répertoire des WAL.

Exemple de contenu de ce fichier :

```
START WAL LOCATION: 582/F4212134 (file 0000000100000582000000F4)
STOP WAL LOCATION: 583/9B0752D8 (file 00000001000005830000009B)
CHECKPOINT LOCATION: 582/F4212168
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2015-07-13 02:04:30 CEST
LABEL: sauvegarde_pitr
STOP TIME: 2015-07-13 02:57:31 CEST
```

On peut donc déduire du contenu de ce fichier que tous les fichiers WAL compris entre `0000000100000582000000F4` et `00000001000005830000009B` sont nécessaires pour restaurer la sauvegarde. Les fichiers antérieurs au WAL `0000000100000582000000F4` ne sont pas nécessaires. Les fichiers postérieurs au WAL `00000001000005830000009B` sont nécessaires si l'on désire restaurer également les transactions effectuées au delà de la fin de ce fichier, donc après la fin de la sauvegarde. En aucun cas un fichier ne doit manquer dans la séquence des WAL archivés. Si c'était le cas, la restauration ne pourrait pas se poursuivre au delà du dernier fichier en séquence - et si le fichier manquant fait partie des fichiers archivés pendant la sauvegarde, alors l'état cohérent ne pourra pas être atteint, et la restauration sera impossible.

D'autres informations importantes sont contenues dans ce fichier, notamment « START TIME » et « STOP TIME ». Ces deux valeurs indiquent l'instant de démarrage (fin du `pg_start_backup()`) et de fin (fin du `pg_stop_backup()`) de la sauvegarde. Comme indiqué précédemment, une restauration de type PITR ne permet à l'instance d'atteindre un état cohérent que lorsque tous les WAL archivés pendant la sauvegarde ont été rejoués. Cela signifie que la sauvegarde donnée ici en exemple peut restaurer les données au plus tôt telles qu'elles étaient au moment du « STOP TIME » (`2015-07-13 02:57:31 CEST`). Si l'on veut restaurer les données à un instant antérieur, même une seconde avant, il faudra utiliser une sauvegarde antérieure, **et** disposer de tous les WAL archivés depuis le début de cette sauvegarde jusqu'au WAL incluant l'instant où l'on désire restaurer.

Pour que l'instance puisse procéder au `recovery`, il est nécessaire de créer un fichier `recovery.conf`, qui doit impérativement se trouver dans le répertoire `PGDATA` de l'instance (et ce, même si les autres fichiers de configuration sont déportés dans un autre répertoire, comme c'est le cas notamment pour les installations par paquet sur Debian).

Le paramètre `restore_command` est indispensable à la restauration. Ce paramètre permet d'indiquer la commande (ou le script) à utiliser pour rejouer les fichiers WAL archivés. C'est un peu l'inverse du paramètre `archive_command` de l'instance : la commande indiquée va être utilisée par PostgreSQL lors du `recovery` pour copier les fichiers WAL depuis le répertoire contenant les WAL archivés vers le répertoire hébergeant les WAL de l'instance. Tout comme pour le paramètre `archive_command`, les chaînes `%f` et `%p` sont également disponibles, et indiquent respectivement le nom du WAL (généralement préfixé du chemin du répertoire d'archivage) et le chemin complet vers lequel le WAL sera copié.

Plusieurs paramètres permettent de contrôler le point d'arrêt de la restauration :

- `recovery_target` n'autorise à ce jour qu'une seule valeur, 'immediate', et indique à l'instance d'arrêter le `recovery` sitôt qu'un point de cohérence est trouvé, ce qui permet donc de restaurer les données telles qu'elles étaient exactement à la fin de la sauvegarde ;
- `recovery_target_name` permet d'indiquer à l'instance d'arrêter le `recovery` lorsque le rejeu atteint un point de restauration nommé, créé sur l'instance en activité (donc avant sa restauration) à l'aide de la fonction `pg_create_restore_point()` ;
- `recovery_target_time` permet d'indiquer à l'instance d'arrêter le `recovery` lorsque le rejeu atteint un timestamp précis ;
- `recovery_target_xid` permet d'indiquer à l'instance d'arrêter le `recovery` lorsque le rejeu atteint un numéro de transaction précis ;
- `recovery_target_lsn` permet d'indiquer à l'instance d'arrêter le `recovery` lorsque le rejeu atteint un LSN précis ;
- `recovery_target_inclusive` est utilisé en conjonction avec `recovery_target_time` et `recovery_target_xid`, et permet d'indiquer si une transaction correspondant exactement au point d'arrêt demandé doit être inclue (`true`, valeur par défaut) ou non (`false`) ;
- `recovery_target_timeline` est utilisé dans le cas de plusieurs restaurations successives, pour spécifier un numéro de *timeline* précis à suivre pendant la restauration (par défaut, la restauration se fait en restant sur la *timeline* qui était active au moment de la sauvegarde) - une valeur particulière, `latest`, est utilisable pour spécifier de suivre la plus récente *timeline*, mais ce cas est surtout utile pour la réplication.

D'autres paramètres peuvent également être utilisés.

Avant la version 9.6, le paramètre `pause_at_recovery_target` n'était pris en compte que si le paramètre `hot_standby` était activé dans le fichier `postgresql.conf` de l'instance et qu'un paramètre `recovery_target*` était activé dans le fichier `recovery.conf`. Si c'est le

cas, et que pause_at_recovery_target est à true (c'est la valeur par défaut), alors au lieu de démarrer normalement, l'instance sera passée en pause sitôt après la fin du recovery, empêchant ainsi toute transaction d'effectuer des modifications de données. Ce mode permet d'effectuer des lectures sur l'instance afin de s'assurer que la cible de restauration est bien celle souhaitée. Si ce n'est pas le cas, il est possible d'arrêter l'instance et de modifier la cible de restauration.

À partir de la version 9.5, ce paramètre a été remplacé par recovery_target_action. Trois options sont possibles :

- pause : même comportement que pause_at_recovery_target à true. Si le paramètre hot_standby n'est pas activé, le comportement sera le même que shutdown.
- promote : L'instance est promue et accepte des requêtes en écriture. On passe à la *timeline* suivante. C'est le même comportement si une cible de restauration est spécifiée et que pause_at_recovery_target est à false.
- shutdown : L'instance est stoppée à la fin du rejeu.

Il faut supprimer le fichier recovery.conf ou modifier l'option pour qu'elle démarre.

Ce paramétrage est généralement utilisé lors d'une restauration à un point dans le temps pour vérifier que le point atteint correspond bien à l'état dans lequel on souhaite restaurer les données. Pour sortir l'instance du mode pause et autoriser de nouveau les écritures, il faut se connecter à l'instance en tant que super utilisateur et exécuter la fonction pg_wal_replay_resume().

Le paramètre recovery_end_command permet de spécifier une commande ou un script qui sera exécutée une fois le recovery terminé, juste avant le démarrage de l'instance.

Le paramètre archive_cleanup_command permet de spécifier une commande ou un script destiné à supprimer les fichiers WAL archivés sitôt qu'ils ne sont plus nécessaires. Celui-ci est habituellement utilisé en combinaison avec le module pg_archivecleanup dans la configuration des instances secondaires en réplication. De plus, il n'est généralement pas utilisé dans le cadre d'une restauration, il est préférable de conserver les WAL archivés disponibles pour pouvoir recommencer la restauration facilement si cela s'avérait nécessaire.

Les autres paramètres utilisables dans ce fichier de configuration concernent la mise en réplication de l'instance, et ne sont donc pas utiles dans le cas d'une restauration.

Exemple de fichier recovery.conf :

```
restore_command = 'rsync -av /mnt/server/archivedir/%f "%p"'
recovery_target_time = '2015-07-27 15:12:00'
```

17.12

Cette configuration aura l'effet suivant sur la phase de `recovery` : * PostgreSQL utilisera la commande spécifiée dans `restore_command` (`rsync`) pour récupérer les fichiers WAL à rejouer ; * lorsque le `recovery` atteindra la première transaction correspondant au moment spécifié par `recovery_target_time`, il arrêtera le `recovery` ; * si l'état atteint est cohérent (ie si on a bien rejoué les WAL archivés au delà du point marquant la fin de la sauvegarde), alors l'instance sera démarrée et ouverte en écriture (ce qui provoquera une incrémentation du numéro de *timeline*).

2.5.6 RESTAURATION PITR : DIFFÉRENTES TIMELINES

- En fin de `recovery`, la *timeline* change
  - l'historique des données prend une autre voie
  - le nom des WAL change pour éviter d'écraser des archives suivant le point d'arrêt
  - l'aiguillage est inscrit dans un fichier `.history`, archivé
- Permet de faire plusieurs restaurations PITR à partir du même *basebackup*
- `recovery_target_timeline` permet de choisir la *timeline* à suivre

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/
000000010000000000000007
```

```
000000010000000000000008
000000010000000000000009
00000001000000000000000A
00000001000000000000000B
00000001000000000000000C
00000001000000000000000D
00000001000000000000000E
00000001000000000000000F
000000010000000000000010
000000010000000000000011
000000020000000000000009
00000002000000000000000A
00000002000000000000000B
00000002000000000000000C
00000002.history
```

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 :

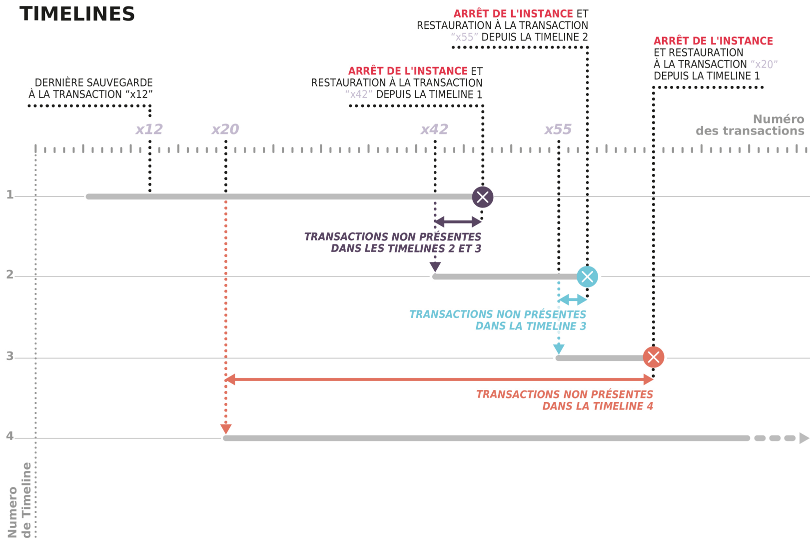
```
$ cat 00000003.history
1 0/9765A80 before 2015-10-20 16:59:30.103317+02


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` dans le fichier `recovery.conf`. Par défaut, la restauration se fait dans la même `timeline` que le *base backup*. Pour choisir une autre `timeline`, il faut donner le numéro hexadécimal de la `timeline` cible comme valeur du paramètre `recovery_target_timeline`. On peut aussi indiquer `'latest'` pour que PostgreSQL détermine la `timeline` la plus récente en cherchant les fichiers `history`. Il prend alors le premier ID de `timeline` disponible. Attention, pour restaurer dans une `timeline` précise, il faut que le fichier `history` correspondant soit présent dans les archives, sous peine d'erreur.

17.12

En sélectionnant la `timeline` cible, on peut alors effectuer plusieurs restaurations successives à partir du même *base backup*.

2.5.7 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 configure le `recovery.conf` 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 le `recovery.conf` 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 `recovery.conf` pour restaurer jusqu'à la transaction `x20`.

Lors du `recovery`, l'instance va :

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

Comme la création des deux `timelines` précédentes est archivée dans les fichiers

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

2.5.8 ADAPTER LA CONFIGURATION

- `postgresql.conf`
  - `port`
  - `listen_addresses`
  - `data_directory`
  - …
- `pg_hba.conf`

Dans certains cas, il peut être nécessaire de modifier le paramétrage de l'instance après la restauration pour permettre à l'instance de démarrer. C'est par exemple le cas si l'on désire restaurer l'instance avec un numéro de port différent de celui d'origine (paramètre `port`), ou avec une interface d' écoute différente (paramètre `listen_addresses`).

Dans certaines configurations (notamment sous Debian), le chemin du PGDATA de l'instance est spécifié en dur dans le fichier de configuration. Dans ce cas, si l'instance est restaurée sur un chemin différent, il faudra également modifier les paramètres mentionnant ce chemin, comme `data_directory`.

Enfin, si l'instance a été restaurée sur une machine hébergée dans un réseau différent, il peut être nécessaire d'adapter le contenu du fichier `pg_hba.conf` pour permettre aux connexions applicatives de s'établir.

2.5.9 AVANTAGES

- Sauvegarde sans interruption de service
- Restauration à la transaction près
- Efficace pour les fortes volumétries

La technique de sauvegarde à chaud a l'énorme avantage de ne pas forcer l'arrêt de l'instance. Cela rend possible la sauvegarde de l'instance avec des contraintes de disponibilités maximales.

La puissance de cette technique tient aussi du fait qu'il est possible de restaurer les données à la transaction près. Il s'agit de la meilleure granularité possible pour une base de données qui respecte les propriétés ACID.

Enfin, le PITR ne souffre pas de limitations dues au volume de données à sauvegarder ou de temps de sauvegarde. Un *base backup* peut prendre des jours, à partir du moment où on garde le précédent, il est possible de restaurer même en cas d'incident durant la sauvegarde. Les archives des fichiers WAL pourront être appliquées sur la sauvegarde précédente, avec la même granularité.

2.5.10 INCONVÉNIENTS

- Restauration de l'instance complète
- Conservation de la fragmentation
- Impossible de changer d'architecture ou de version
- Point dans le temps souvent difficile à trouver

Comme la sauvegarde des fichiers à froid, le PITR sauvegarde et restaure l'instance complète. Dans une instance contenant plusieurs bases de données pour des applications différentes, une restauration ramène toutes les bases de données au point dans le temps configuré. Comme la technique convient aux fortes volumétries, on la réserve souvent aux grosses instances (volumétrie supérieure à plusieurs centaines de gigaoctets) qui ne gèrent qu'une application à la fois.

On a aussi les limitations inhérentes à la sauvegarde des fichiers de l'instance : on sauvegarde la fragmentation des fichiers de données des tables et des index. Aussi, la sauvegarde est liée au format de stockage binaire, il est donc impossible de changer de version majeure de PostgreSQL (un des deux premiers chiffres de la version qui change) et d'architecture processeur.

Enfin, le point dans le temps idéal est souvent difficile à déterminer. On n'a généralement pas une information précise sur la date et heure d'un incident, sachant que le point d'arrêt est configurable à la microseconde près. Comme on ne peut pas « désappliquer » les données des journaux de transactions, il faut recommencer la restauration si on a dépassé le point dans le temps de l'incident.

2.6 ÉCRITURE D'UN SCRIPT DE SAUVEGARDE

- Un script de sauvegarde doit :
  - être testé
  - être documenté
  - gérer la sauvegarde dans son intégralité
  - gérer les cas d'erreur

La sauvegarde déclenchée par un script devrait être complète et ne pas dépendre d'autres actions pour permettre une restauration. Par exemple, les sauvegardes devraient systématiquement prendre en considération tous les fichiers de configuration de l'instance, même si ceux-ci se trouvent dans d'autres répertoires (`/etc`, …). De la même façon, un script déclenchant une sauvegarde logique (export de données) doit prendre soin d'exporter également la définition des objets globaux afin que cette information ne soit pas perdue au moment de restaurer.

2.6.1 POINTS D'ATTENTION

- Le script de sauvegarde doit au minimum :
  - être commenté
  - être versionné
  - renvoyer un code d'erreur différent de zéro si un problème est survenu
  - permettre de tracer les différentes étapes de la sauvegarde
  - annuler la sauvegarde en cours en cas d'erreur
  - générer des sauvegardes valides (il convient donc de les tester)

Les commentaires sont souvent rares ou inexistants dans les scripts d'exploitation, ce qui peut amener à des problèmes de maintenance et de robustesse lors de futures évolutions du script. Le script devrait également être versionné, de préférence à l'aide d'un outil de gestion de version comme `git` ou `svn`.

Il est très important que le script renvoie un code d'erreur différent de zéro si un problème a été détecté pendant l'exécution du script. Par exemple, dans le cas d'une sauvegarde logique, si on commence par sauvegarder les objets globaux à l'aide de `pg_dumpall -g`, il faut faire attention à bien récupérer le code retour de cette étape afin que le script puisse indiquer à la fin qu'au moins une partie de la sauvegarde est manifestement invalide.

Il est également très important de tracer autant d'informations que possibles, au minimum vers les sorties standard / d'erreur, et que ces sorties soient redirigées vers des fichiers lors des exécutions planifiées (on peut bien sûr utiliser des solutions plus poussées, comme

rsyslog). Ces messages de traces devraient toujours contenir au moins un timestamp et suffisamment d' éléments pour déterminer le succès ou l'échec des différentes étapes importantes du script, ainsi que les éventuels messages d'erreurs associés.

En cas d'erreur survenant lors de l'exécution du script, celui-ci doit s'assurer que la sauvegarde est annulée proprement, par exemple que la fonction `pg_stop_backup()` est bien exécutée même si la sauvegarde elle-même a échoué. Il peut également être opportun de déclencher un événement (comme un envoi de mail) depuis le script pour signaler l'erreur.

2.6.2 DOCUMENTER LE SCRIPT

- La documentation devrait :
  - détailler l'architecture de sauvegarde
  - expliquer le fonctionnement du script (options, …)
  - ses dépendances (logiciels, points de montage, …)
  - être relue et validée par toute l'équipe
  - permettre de trouver la documentation associée à la restauration

La documentation du script devrait être aussi détaillée que possible. L'architecture de sauvegarde doit être expliquée dans son intégralité, et de préférence accompagnée de schémas tenus à jour.

Les différentes options du script, ainsi que son fonctionnement, les choix d'implémentation (sauvegarde logique, physique, commandes utilisées…), ses dépendances (comme par exemple un montage NFS), doivent toutes être explicitées et tenues à jour.

Évidemment, l'emplacement de la documentation expliquant les méthodes de restauration des sauvegardes en question devrait être indiquée dans la documentation des sauvegardes, à moins que la procédure de restauration ne fasse directement partie de cette même documentation.

2.6.3 SPÉCIFICITÉS D'UN SCRIPT DE SAUVEGARDE LOGIQUE

- `pg_dumpall -g` : définition des rôles et des tablespaces
- `pg_dump` : contenu de chaque base de données
- Restent la définition des configurations des bases et les ACL…
- Attention aux fichiers de configuration

Pour réaliser un script de sauvegarde logique complet et surtout qui permet une restauration sélective aisée, il faut combiner l'utilisation de `pg_dumpall` pour obtenir la définition des objets globaux et celle de `pg_dump` pour extraire le schéma et les données dans un format exploitable par `pg_restore`. On pourrait utiliser simplement `pg_dumpall`, mais le format `plain` n'est pas facile à manipuler si on souhaite ne restaurer qu'une table d'une certaine base de donnée. `pg_restore` offre cette souplesse avec les formats binaires (`tar`, `directory` et `custom`), on doit alors utiliser `pg_dump`.

Pour obtenir la liste des bases de données, il suffit de requêter la table `pg_database` avec `psql`, on utilisera l'option `-c` pour lui passer la requête en ligne de commande.

`pg_dumpall` avec l'option `-g` permet d'extraire la définition des rôles et des tablespaces. Cependant, les ACL et paramètres de configuration placés sur les bases de données et les paramètres de configurations placés sur des rôles dans des bases de données ne sont pas sauvegardés.

On peut obtenir les ACL sur les bases de données avec la méta-commande `\l` de `psql`. La configuration par base de données et rôle est disponible en utilisant la méta-commande `\drds` de `psql`.

2.6.4 SPÉCIFICITÉS D'UN SCRIPT DE SAUVEGARDE PITR - ARCHIVAGE DES WALS

- Prérequis : configurer l'archivage des WAL
- La commande d'archivage peut être un script
  - dans ce cas, attention à bien tester les codes de retour !
- Utiliser `pg_start_backup()` et `pg_stop_backup()`
  - même en cas d'utilisation d'un mécanisme de snapshot (virtualisation, baie…)

Pour qu'un script de sauvegarde basé sur la méthode PITR fonctionne, sa première dépendance est que l'archivage des journaux de transactions soit bien configuré au niveau de l'instance à sauvegarder, et que cet archivage soit fonctionnel au moment de l'exécution de la sauvegarde.

La commande d'archivage configurée via le paramètre `archive_command` peut être plus complexe qu'une unique commande. On peut par exemple vouloir archiver les WAL vers plus d'une seule destination si l'on archive pour la sauvegarde PITR ET pour une ou plusieurs instances en réplication *Log shipping*. Dans ce cas, il est préférable d'écrire un script d'archivage, et d'appeler ce script dans l'"archive_command" en lui passant en paramètres les informations nécessaires (au minimum, le chemin complet vers le WAL à archiver, à savoir `%p`). Il est extrêmement important que ce script valide le code retour de chaque étape effectuée, afin de détecter si l'un des archivages a échoué, et remonter un

code de retour différent de zéro à PostgreSQL dans ce cas. En effet, si le script renvoie un retour de zéro en dépit de l'échec d'une partie de l'archivage, l'instance considérera que l'archivage a réussi, et la séquence des WAL archivés sera rompue.

Pour gagner de l'espace de stockage, la commande d'archivage peut éventuellement inclure la compression du WAL archivé. Il faut faire bien attention dans ce cas à ce que la procédure / script de restauration soit adapté en conséquence, notamment pour intégrer la décompression des WAL archivés au fil de leur rejeu dans la configuration de la `restore_command`.

2.6.5 SPÉCIFICITÉS D'UN SCRIPT DE SAUVEGARDE PITR - LA COPIE

- Laisser le temps à `pg_start_backup()` de terminer avant de démarrer la sauvegarde
- Ne pas oublier de fichiers lors de la copie (tablespaces, …)
- Laisser le temps à `pg_stop_backup()` de terminer avant de déclarer la sauvegarde valide
- Penser à sauvegarder et référencer les WAL archivés référencés dans le `.backup`

Une fois que l'on s'est assuré que la commande d'archivage est fonctionnelle, le script de sauvegarde lui-même peut être réalisé. Celui-ci doit gérer tous les aspects vus précédemment dans la présentation de la méthode de sauvegarde PITR.

Pour commencer, le script doit se connecter à l'instance à sauvegarder et exécuter la fonction `pg_start_backup()`. Tant que cette commande n'a pas rendu la main avec un code de retour de zéro, la copie des fichiers ne peut pas être démarrée, faute de quoi la sauvegarde sera invalide. Pour rappel, par défaut, cette fonction attend l'exécution du prochain *checkpoint* sur l'instance - si le script est exécuté sur une période de très faible activité, cela peut prendre du temps. Il peut alors être utile de forcer l'exécution immédiate d'un checkpoint à l'aide du second paramètre de la fonction, par exemple :

```
SELECT pg_start_backup('daily_backup', true) ;
```

Une fois que la fonction est terminée, la copie des fichiers peut commencer. Là encore, tester le code de retour de la commande est important - l'échec de la copie d'un seul fichier compromet la totalité de la sauvegarde. Attention toutefois, certaines commandes de copie de fichiers (comme `tar par exemple`) envoient des messages d'erreur et peuvent sortir avec un code d'erreur si des fichiers sont modifiés pendant la sauvegarde. Ces erreurs doivent être traitées comme normales par le script. En effet, l'instance étant encore ouverte en écriture, la modification (voire la disparition) de fichiers de l'instance est normale.

Attention à ne pas oublier de fichiers lors de la copie ! Parmi les oublis les plus fréquents, on trouve les fichiers de configuration (notamment sur Debian, où ils sont relocalisés par défaut dans `/etc`) et surtout les fichiers des tablespaces.

Lorsque la copie des fichiers est terminée, le script doit de nouveau se connecter à l'instance et exécuter la fonction `pg_stop_backup()`. Tant que cette fonction n'a pas renvoyé un code de retour de zéro, la sauvegarde n'est pas valide ! Il est possible que cette commande mette du temps à rendre la main du fait d'un grand nombre de fichiers WAL à archiver. Si l'archivage est défaillant, la commande ne se terminera jamais, mais enverra toutes les minutes des messages de ce type :

```
WARNING:  pg_stop_backup still waiting for all required WAL segments to be
          archived (60 seconds elapsed)
HINT:  Check that your archive_command is executing properly.  pg_stop_backup
       can be canceled safely, but the database backup will not be usable
       without all the WAL segments.
```

Une fois que PostgreSQL a reçu le signal de fin de sauvegarde et a terminé d'archiver les WAL, le script devrait vérifier que tous les WAL compris entre le premier et le dernier mentionnés dans le fichier `.backup` sont bien tous présents dans le répertoire de destination. Une fois cela fait, la sauvegarde est concrètement terminée.

2.6.6 SPÉCIFICITÉS D'UN SCRIPT DE SAUVEGARDE PITR - RÉTENTION

- Gérer une période de rétention pour les sauvegardes ET les WAL archivés
- attention à bien conserver tous les fichiers WAL archivés nécessaires à la restauration

La dernière étape à gérer est celle de la rétention des sauvegardes. Cela peut être potentiellement problématique à implémenter car les sauvegardes sont séparées en deux composantes :

- les fichiers de données copiés ;
- les WAL archivés.

Il est important lorsque l'application de la politique de rétention supprime une sauvegarde, que seuls les WAL archivés qui servent à cette sauvegarde soient supprimés. Pour cela, le script doit se baser sur le contenu du fichier `.backup` généré à chaque sauvegarde, et qui indique le premier et le dernier fichier de la séquence des WAL indispensables à la restauration d'une sauvegarde.

2.7 ÉCRITURE D'UN SCRIPT DE RESTAURATION

- Un script de restauration doit :
  - être testé
  - être documenté
  - permettre la restauration intégrale de l'instance
  - détecter et signaler les erreurs rencontrées

La restauration est habituellement une opération destinée à être exécutée manuellement, ce qui ne signifie pas qu'il faut négliger l'écriture d'un script. L'un des premiers objectifs des sauvegardes est d'offrir la capacité à restaurer les données en cas d'incident majeur. Il est donc extrêmement important d'avoir écrit le script et la procédure de restauration en gardant en tête que cette opération sera certainement effectuée dans des conditions difficiles (incident en pleine nuit, production arrêtée, …). Les actions doivent être aussi simples et claires que possible, pour ne laisser aucune marge d'erreur.

2.7.1 POINTS D'ATTENTION

- Le script de restauration doit au minimum :
  - être commenté
  - être versionné
  - faire un récapitulatif des actions effectuées et demander validation
  - renvoyer un code d'erreur différent de zéro si un problème est survenu
  - permettre de tracer les différentes étapes de la restauration
  - remettre l'instance dans un état complètement fonctionnel

Les commentaires sont souvent rares ou inexistants dans les scripts d'exploitation, ce qui peut amener à des problèmes de maintenance et de robustesse lors de futures évolutions du script. Le script devrait également être versionné, de préférence à l'aide d'un outil de gestion de version comme `git` ou `svn`.

Une restauration est par nature une opération destructive ! Il ne faudrait pas qu'une personne lance un tel script sur une instance de production fonctionnelle… Il est donc préférable que, par défaut, le script effectue un récapitulatif des actions **avant** de les effectuer, et demande une validation explicite.

Une fois l'exécution décidée, il est très important que le script renvoie un code d'erreur différent de zéro si un problème a été détecté pendant l'exécution du script. Par exemple, dans le cas de la restauration d'une sauvegarde logique, on aura souvent plusieurs centaines d'objets qui seront restaurés, et l'absence d'un seul d'entre eux devrait être

considérée comme critique. Par ailleurs, ce type de restauration s'effectue souvent en plusieurs étapes (restauration des objets globaux, rôles et tablespaces, puis restauration des données elles-mêmes), il convient de s'assurer que chaque étape s'est correctement déroulée.

Il est également très important de tracer autant d'informations que possible, au minimum vers les sorties standard / d'erreur, et que ces sorties soient redirigées vers des fichiers lors d'une restauration (on peut bien sûr utiliser des solutions plus poussées, comme rsyslog). Ces messages de traces devraient toujours contenir au moins un timestamp et suffisamment d'éléments pour déterminer le succès ou l'échec des différentes étapes importantes du script, ainsi que les éventuels messages d'erreurs associés.

En cas d'erreur survenant lors de l'exécution du script, c'est l'administrateur chargé d'effectuer la restauration qui devra décider s'il faut continuer la procédure ou l'interrompre pour effectuer des actions correctrices. Il faut donc que les messages d'erreur soient facilement identifiables par la personne effectuant la restauration, et qu'ils contiennent des informations aussi détaillées que possible sur le contexte de l'erreur et les causes possibles.

2.7.2 DOCUMENTER LE SCRIPT

- La documentation devrait
  - donner succintement et clairement l'usage du script
  - expliquer dans le détail le fonctionnement du script (commandes utilisées, …)
  - détailler les impacts (données écrasées dans un répertoire, …)
  - ses dépendances (logiciels, points de montage, …)
  - être relue et validée par toute l'équipe
  - être facile à trouver dans une version à jour (ne pas l'imprimer !)

L'usage d'un script de restauration ne devrait pas nécessiter de réflexion - c'est la décision de restaurer ou non qui mérite une réflexion. Le script doit être en lui-même aussi simple et clair que possible, et sa documentation doit suivre le même principe. Il ne faut pas que l'administrateur d'astreinte doive parcourir dix pages de documentation pour réussir à se faire une idée des options à utiliser !

Les différentes options du script, ainsi que son fonctionnement, les choix d'implémentation (restauration logique, physique, commandes utilisées, emplacement des sauvegardes, …), ses dépendances (comme par exemple un montage NFS), doivent toutes être explicitées et tenues à jour. La documentation devrait contenir des exemples d'utilisation avec différents cas de figures prévus. Il ne faut pas hésiter à tenir à jour cette documen-

tation fréquemment, à la faire relire et dérouler par toutes les personnes de l'équipe régulièrement.

La documentation doit également expliquer clairement les impacts d'une restauration. Au moment de lancer le script, l'administrateur doit être capable de savoir précisément dire quelle sera la perte de données consécutive à la manipulation.

Cette documentation devrait être stockée à un emplacement connu par l'équipe chargée d'intervenir en cas d'incident. Ne pas imprimer une telle documentation ! C'est le meilleur moyen pour qu'une mauvaise version de la procédure soit utilisée…

2.7.3 SPÉCIFICITÉS D'UN SCRIPT DE RESTAURATION LOGIQUE

- Laisser `pg_restore` détecter le format des dump binaires
- N'automatiser que les restaurations récurrentes
- Demander des confirmations de façon interactive pour les suppressions de données
- Utiliser les modes verbeux des commandes pour tracer l'exécution

Les besoins de restauration, surtout partielles, sont souvent nombreux et un script ne peut répondre à tous sans apporter une complexité souvent génératrice de bugs. C'est pourquoi, il vaut mieux n'automatiser que les cas de restaurations fréquemment rencontrés. Pour le reste, il vaut mieux utiliser directement `pg_restore`, vu qu'il est plutôt versatile.

Comme une restauration de données peut remplacer des données existantes, il est fortement conseillé de rendre le script interactif en demandant la confirmation de l'utilisateur avant la suppression des données.

Enfin, il convient de tracer un maximum d'opérations en activant les modes verbeux des commandes :

- pour `psql`, il s'agit des paramètres `--echo-queries`, voire `--echo-all`
- pour `pg_restore`, il s'agit de `--verbose`

2.7.4 SPÉCIFICITÉS D'UN SCRIPT DE RESTAURATION PITR

- Utiliser la date et heure de fin pour choisir le *basebackup* à utiliser
  - le point d'arrêt dans le temps doit être après le point de cohérence
  - on peut utiliser le `backup_label` archivé pour obtenir le `STOP TIME` du backup
- Restaurer le répertoire de données et les tablespaces

- Configurer au moins `restore_command` dans `$PGDATA/recovery.conf`
- Laisser à l'utilisateur le soin de démarrer l'instance

Le script de restauration doit être adapté à la méthode de sauvegarde choisie. Si la sauvegarde est effectuée avec `tar` suivi d'une compression, le script de restauration devra également utiliser `tar` et l'outil approprié pour décompresser l'archive. Si des outils de snapshot spécifiques ont été utilisés pour la sauvegarde (système de fichiers, virtualisation, SAN), alors la restauration devra faire appel à ces mêmes outils.

L'objectif d'un tel script est de pouvoir ramener l'instance à un point dans le temps le plus rapidement possible, c'est pourquoi il faut choisir le *basebackup* qui soit à la fois le plus proche et **dont la fin précède ce point dans le temps**. Comme la sauvegarde a été faite à chaud, les fichiers de données ne sont pas cohérents, la première phase de la restauration consiste donc a retourner à l'état cohérent. On ne peut donc pas choisir de point d'arrêt avant la date et heure du point de cohérence, qui correspond à peu près à la date d'exécution de `pg_stop_backup()`.

Pour trouver le meilleur *base backup* lorsqu'on en a plusieurs, successifs, à disposition, il est possible d'obtenir la date de fin du *basebackup* en récupérant le `backup_label` archivé (fichier terminant par `.backup`).

Une fois les fichiers copiés depuis le *base backup*, il est nécessaire de créer le fichier de configuration `$PGDATA/recovery.conf` et d'y configurer au minimum le paramètre `restore_command`. Ce paramètre doit contenir la commande pour copier un fichier WAL archivé et le fournir à PostgreSQL :

- On peut utiliser une commande ou un script ;
- Il ne faut pas oublier de décompresser le fichier si nécessaire ;
- Il faut toujours copier le fichier, jamais supprimer le fichier source, sinon le backup deviendrait inutilisable faute d'archives.

Enfin, on recommande de ne pas démarrer automatiquement l'instance en fin de script. Il est préférable qu'un opérateur s'en charge afin qu'il puisse vérifier, voire affiner la configuration au préalable. De plus, le point d'arrêt étant souvent difficile à trouver, on a tendance à recommencer la procédure de restauration pour s'en approcher au mieux.

2.8 CONCLUSION

- Les techniques de sauvegarde de PostgreSQL sont :
  - complémentaires
  - automatisables

- La maîtrise de ces techniques est indispensable pour assurer un service fiable.

2.9 TRAVAUX PRATIQUES

2.9.1 ÉNONCÉS

Il est nécessaire d'installer une instance PostgreSQL, utiliser pour cela la version la plus récente disponible. Voir : http://www.postgresql.org/download/

Préparation

- En tant qu'utilisateur `postgres`, créer un sous-répertoire dans le répertoire `home` pour héberger le tablespace de test :

```
mkdir ~postgres/testtbs/
```

- Démarrer l'instance, s'y connecter et créer une base de données nommée `testdb`, un tablespace, un rôle et une table de suivi pour nos tests :

```
CREATE DATABASE testdb ;
CREATE ROLE testuser WITH LOGIN ;
CREATE TABLESPACE testtbs LOCATION '/var/lib/pgsql/testtbs/' ;

\password testuser

\c testdb

CREATE TABLE suivi (id serial, datetime timestamptz, timeline text)
       TABLESPACE testtbs;
ALTER TABLE suivi OWNER TO testuser ;

\q
```

En fonction du système utilisé (CentOS, Debian…) l'emplacement du répertoire `home` de l'utilisateur `postgres` peut varier : adapter en conséquence la commande de création du tablespace.

- Créer également le script `gen_activity.sh` suivant :

```
#!/bin/bash

db=testdb
logfile=$(dirname $0)/gen_activity.log

while $(true); do
    if ! psql -c "select 1" >/dev/null 2>&1 ; then
```

```
        echo "$(date) : L'instance est arretee " >> $logfile
        echo "====================================" >> $logfile
        sleep 5
        continue
    fi
    echo "$(date) : Insertion d'une ligne dans la table de suivi :" >> $logfile
    psql -c "select pg_switch_wal()" $db
    psql -c "insert into suivi (datetime, timeline) values (
        now(),
        trim(leading '0' from substr(
            pg_walfile_name(
                pg_current_wal_lsn()
            ), 1, 8))) RETURNING *;" $db >> $logfile
    echo "====================================" >> $logfile
    sleep 5
done
```

Ce script génère des écritures dans la table de suivi, et s'exécute ainsi :

```
./gen_activity.sh
```

Il fonctionnera en arrière-plan, insérant des données à intervalle régulier afin que l'on puisse visualiser l'impact en terme de perte de données des différentes méthodes.

À chaque insertion, il écrit dans le fichier de log `gen_activity.log` la dernière ligne insérée dans la table de suivi.

Sauvegardes logiques

Assurez-vous que le script `gen_activity.sh` est bien actif et a déjà inséré des données.

Sauvegarde logique :

- Prenez une sauvegarde logique complète des objets importants de l'instance.

Restauration complète :

- Laisser le script `gen_activity.sh` tourner encore quelques minutes, puis l'arrêter.
- Arrêter et supprimer l'instance, en créer une nouvelle et restaurer son contenu.
- S'assurer que tout le contenu de l'instance est bien restauré :

```
SELECT 'DB OK' FROM pg_database WHERE datname = 'testdb' ;
```

```
SELECT 'USER OK' FROM pg_roles WHERE rolname = 'testuser' ;
```

- Visualiser la perte de données induite par la restauration en comparant les dernières lignes présentes dans la table de suivi et les dernières lignes insérées par le script `gen_activity.sh` avant l'arrêt de l'instance.
- Relancer le script `gen_activity.sh`.

Restauration partielle :

- Restaurer :
  - uniquement la table `suivi`
  - dans une nouvelle base de données `testdb2`
  - dans le PGDATA de l'instance
  - avec un nouveau propriétaire `testuser2`
- Note : Il sera certainement nécessaire pour cette étape d'adapter la gestion de l'authentification dans le fichier `pg_hba.conf` pour permettre la connexion avec l'utilisateur `testuser2`, par exemple en y ajoutant la ligne suivante :

```
local           all           testuser2               md5
```

- Tester une insertion de données à l'aide de la commande suivante :

```
INSERT INTO suivi (datetime, timeline) VALUES (
        now(),
        trim(leading '0' from substr(
            pg_walfile_name(
                pg_current_wal_lsn()
            ), 1, 8)));
```

- Que constate-t-on ? Comment corriger cette erreur ?

PITR

Assurez-vous que le script `gen_activity.sh` est bien actif et a déjà inséré des données.

Sauvegarde PITR :

- Réaliser une sauvegarde PITR :
  - Configurer l'archivage des journaux de transaction
  - Créer un basebackup de toute l'instance (attention aux tablespaces et aux fichiers de configuration)

Restauration à un point dans le temps :

- Stopper le script et relever la ligne la plus récente de la table `suivi`, en regardant dans le fichier `gen_activity.log` ou en exécutant la requête suivante :

```
SELECT * FROM suivi ORDER BY 2 DESC LIMIT 1;
```

- Arrêter et supprimer l'instance et restaurer à partir du backup, 30 secondes avant la date de la ligne relevée précédemment. Vérifier le contenu de la table `suivi`.

Suivi de timeline :

- Relancer le script pendant quelques minutes.

- Arrêter le script et relever l'avant-dernière ligne de la table `suivi` et le numéro de transaction associé, en exécutant la requête suivante :

```
SELECT xmin AS xid,* FROM suivi ORDER BY 3 DESC LIMIT 1 OFFSET 1;
```

- Arrêter l'instance.
- Restaurer les données à l'identifiant de transaction relevé à l'étape précédente en excluant cette transaction.

Contrôle du mode recovery :

- Faire en sorte que le serveur soit accessible en lecture seule durant le mode recovery.
- Relancer le script pour générer de l'activité pendant quelques minutes.
- Réaliser un nouveau basebackup de l'instance.
- Laisser le script insérer des données pendant quelques minutes, puis l'arrêter et relever la ligne la plus récente de la table `suivi`.
- Arrêter l'instance et restaurer à partir du dernier backup, 30 secondes avant la date de la ligne relevée précédemment. Faire en sorte que le serveur ne quitte pas le mode recovery.
- Avancer de 3 transactions et terminer le mode recovery.

Point de cohérence :

- Déterminer un point dans le temps entre la date de début et celle de fin du dernier basebackup.
- Restaurer à ce point.

2.9.2 SOLUTIONS

Sauvegardes logiques

Les ordres suivants sont à exécuter en tant qu'utilisateur postgres, à part les ordres `sudo`.

Prendre une sauvegarde logique complète des objets importants de l'instance :

- Création du répertoire de sauvegardes :

```
mkdir -p ~postgres/10/backups/dumps/
```

- Sauvegarde des objets globaux :

```
pg_dumpall -g -f ~postgres/10/backups/dumps/globals.sql
```

- Sauvegarde du contenu de la base `testdb` :

```
pg_dump -Fc -f ~postgres/10/backups/dumps/testdb.dmp testdb
```

- Sauvegarde des fichiers de configuration (note : sur Debian, ils se trouvent dans `/etc` et non dans le `PGDATA`) :

```
cp -p $PGDATA/*.conf ~postgres/10/backups/dumps/
```

Restauration complète :

- Laisser le script `gen_activity.sh` tourner encore quelques minutes, puis l'arrêter.
- Arrêter et supprimer l'instance : utiliser la commande appropriée au système utilisé
  - par exemple sur CentOS 6 :

```
sudo service postgresql-10 stop
rm -rf ~postgres/10/data/*
rm -rf ~postgres/testtbs/*
```

Créer une nouvelle instance - utiliser la commande appropriée au système utilisé :

```
* par exemple sur CentOS 6 :
```

```
sudo service postgresql-10 initdb
```

- Restaurer les fichiers de configuration (note : sur Debian, ils se trouvent dans `/etc` et non dans le `PGDATA`) :

```
cp -p ~postgres/10/backups/dumps/*.conf $PGDATA/
```

- Démarrer l'instance - utiliser la commande appropriée au système utilisé
  - par exemple sur CentOS 6 :

```
sudo service postgresql-10 start
```

- Restaurer les objets globaux :

```
psql -f ~postgres/10/backups/dumps/globals.sql
```

- Restaurer du contenu de la base `testdb` en recréant la base de données :

```
pg_restore -d template1 -C ~postgres/10/backups/dumps/testdb.dmp
```

- S'assurer que tout le contenu de l'instance est bien restauré :

```
SELECT 'DB OK' FROM pg_database WHERE datname = 'testdb' ;
```

```
SELECT 'USER OK' FROM pg_roles WHERE rolname = 'testuser' ;
```

- Visualiser la perte de données induite par la restauration en comparant les dernières lignes présentes dans la table de suivi et les dernières lignes insérées par le script `gen_activity.sh` avant l'arrêt de l'instance :

```
SELECT * FROM suivi ORDER BY 2 DESC LIMIT 1;
```

- Relancer le script `gen_activity.sh`.

Restauration partielle :

17.12

- Créer la nouvelle base de données et le nouveau rôle :

```
CREATE DATABASE testdb2 ;
CREATE ROLE testuser2 WITH LOGIN ;
\password testuser2
```

- Restaurer uniquement la table `suivi` dans la nouvelle base de données `testdb2`, dans le `PGDATA` de l'instance, avec comme nouveau propriétaire `testuser2` :

```
pg_restore -v -O --no-tablespace -t suivi -d testdb2 -U testuser2 \
           ~postgres/10/backups/dumps/testdb.dmp
```

- Tester une insertion de données dans cette table à l'aide de la commande suivante :

```
INSERT INTO suivi (datetime, timeline) VALUES (
        now(),
        trim(leading '0' from substr(
            pg_walfile_name(
                pg_current_wal_lsn()
            ), 1, 8)));
```

- Que constate-t-on ?

L'insertion échoue avec l'erreur suivante :

```
ERROR:  null value in column "id" violates not-null constraint
```

- La raison est que la séquence associée à la colonne `id` n'a pas été restaurée.
- Supprimer la table restaurée dans la base de données `testdb2` :

```
\c testdb2
DROP TABLE suivi;
```

Pour effectuer une restauration partielle de la table et de la séquence associée, il faut utiliser les options `-l` et `-L` de `pg_restore` :

- Exporter la TOC de la sauvegarde dans un fichier :

```
pg_restore -v -l ~postgres/10/backups/dumps/testdb.dmp \
           > ~postgres/10/backups/dumps/toc.txt
```

- Éditer le fichier `toc.txt` pour commenter toutes les lignes sauf celles correspondant à la table `suivi` et à la séquence `suivi_id_seq`.
- Restaurer en utilisant ce fichier comme TOC :

```
pg_restore -v -L ~postgres/10/backups/dumps/toc.txt -O --no-tablespace \
           -d testdb2 -U testuser2 ~postgres/10/backups/dumps/testdb.dmp
```

- Tester de nouveau l'insertion pour s'assurer du bon fonctionnement de la restauration.

PITR

Réaliser une sauvegarde PITR

- Créer un répertoire pour les sauvegardes et pour les WAL archivés :

```
mkdir -p ~postgres/10/backups/bkp_pitr/ ~postgres/10/backups/archived_wals/
```

- Configurer l'archivage des journaux de transaction - modifier les paramètres suivants dans le fichier `postgresql.conf`:

```
wal_level = replica
archive_mode = on
archive_command = 'rsync -a %p /var/lib/pgsql/10/backups/archived_wals/%f'
```

- Redémarrer l'instance - utiliser la commande appropriée au système utilisé
- par exemple sur CentOS 6 :

```
sudo service postgresql-10 restart
```

- Tester que l'archivage fonctionne :

```
SELECT pg_switch_wal();
```

- Créer un basebackup de toute l'instance (attention aux tablespaces et aux fichiers de configuration):

```
cd ~
psql -c "SELECT pg_start_backup('basebackup1', true);"
tar --exclude="./10/data/pg_wal/*" -zcvf \
    ./10/backups/bkp_pitr/basebackup1_pgdata.tgz ./10/data/
tar -zcvf ./10/backups/bkp_pitr/basebackup1_testtbs.tgz ./testtbs/
psql -c "SELECT pg_stop_backup();"
```

Ces commandes, adaptées à une installation CentOS 6 par défaut, sont à adapter en fonction de l'installation choisie et du système utilisé. Notamment, l'emplacement du répertoire `PGDATA` peut varier, ainsi que le chemin du répertoire home de l'utilisateur `postgres` et l'emplacement des fichiers de configuration.

Restauration à un point dans le temps :

- Stopper le script et relever la ligne la plus récente de la table `suivi`, en regardant dans le fichier `gen_activity.log` ou en exécutant la requête suivante :

```
SELECT * FROM suivi ORDER BY 2 DESC LIMIT 1;
```

- Arrêter et supprimer l'instance : utiliser la commande appropriée au système utilisé ; par exemple sur CentOS 6 :

```
sudo service postgresql-10 stop
rm -rf ~postgres/10/data/*
rm -rf ~postgres/testtbs/*
```

- Restaurer à partir du backup :

```
cd ~
tar -zxvf ./10/backups/bkp_pitr/basebackup1_pgdata.tgz
tar -zxvf ./10/backups/bkp_pitr/basebackup1_testtbs.tgz
```

- Créer le fichier `recovery.conf` dans le répertoire PGDATA pour indiquer à PostgreSQL de s'arrêter 30 secondes avant l'insertion de la dernière ligne relevée précédemment :

```
restore_command = 'rsync -a /var/lib/pgsql/10/backups/archived_wals/%f %p'
recovery_target_time = '<timestamp relevé moins 30 secondes>'
```

- Démarrer l'instance - utiliser la commande appropriée au système utilisé ; par exemple sur CentOS 6 :

```
sudo service postgresql-10 start
```

- Vérifier le contenu de la table `suivi` :

```
SELECT * FROM suivi ORDER BY 2 DESC LIMIT 1;
```

Suivi de timeline :

- Relancer le script pendant quelques minutes.
- Arrêter le script et relever l'avant dernière ligne de la table `suivi` et le numéro de transaction associé, en exécutant la requête suivante :

```
SELECT xmin AS xid,* FROM suivi ORDER BY 3 DESC LIMIT 1 OFFSET 1;
```

- Arrêter et supprimer l'instance : utiliser la commande appropriée au système utilisé ; par exemple sur CentOS 6 :

```
sudo service postgresql-10 stop
rm -rf ~postgres/10/data/*
rm -rf ~postgres/testtbs/*
```

- Restaurer à partir du backup :

```
cd ~
tar -zxvf ./10/backups/bkp_pitr/basebackup1_pgdata.tgz
tar -zxvf ./10/backups/bkp_pitr/basebackup1_testtbs.tgz
```

- Créer le fichier `recovery.conf` dans le répertoire PGDATA pour indiquer à PostgreSQL de s'arrêter à l'identifiant de transaction relevé à l'étape précédente en excluant cette transaction - il faudra également lui indiquer de suivre la timeline 2 :

```
restore_command = 'rsync -a /var/lib/pgsql/10/backups/archived_wals/%f %p'
recovery_target_xid = '<XID relevé>'
recovery_target_inclusive = false
```

```
recovery_target_timeline = 2
```

- Démarrer l'instance - utiliser la commande appropriée au système utilisé ; par exemple sur CentOS 6 :

```
sudo service postgresql-10 start
```

- Vérifier le contenu de la table `suivi` :

```
SELECT * FROM suivi ORDER BY 2 DESC LIMIT 1;
```

Contrôle du mode recovery :

- Faire en sorte que le serveur soit accessible en lecture seule durant le mode recover : l'instance PostgreSQL restaurée doit avoir les paramètres suivants activés dans le fichier `postgresql.conf`:

```
wal_level = 'replica'
hot_standby = 'on'
```

- Appliquer ces modifications en redémarrant l'instance ; par exemple sur CentOS 6 :

```
sudo service postgresql-10 restart
```

- Relancer le script `gen_activity.sh`
- Créer un nouveau basebackup de toute l'instance (attention aux tablespaces et aux fichiers de configuration) :

```
cd ~
psql -c "SELECT pg_start_backup('basebackup2', true);"
tar --exclude="./10/data/pg_wal/*" -zcvf \
    ./10/backups/bkp_pitr/basebackup2_pgdata.tgz ./10/data/
tar -zcvf ./10/backups/bkp_pitr/basebackup2_testtbs.tgz ./testtbs/
psql -c "SELECT pg_stop_backup();"
```

- Laisser le script insérer des données pendant quelques minutes, puis l'arrêter et relever la ligne la plus récente de la table `suivi`.
- Arrêter et supprimer l'instance : utiliser la commande appropriée au système utilisé ; par exemple sur CentOS 6 :

```
sudo service postgresql-10 stop
rm -rf ~postgres/10/data/*
rm -rf ~postgres/testtbs/*
```

- Restaurer à partir du dernier backup :

```
cd ~
tar -zxvf ./10/backups/bkp_pitr/basebackup2_pgdata.tgz
tar -zxvf ./10/backups/bkp_pitr/basebackup2_testtbs.tgz
```

- Créer le fichier `recovery.conf` dans le répertoire PGDATA pour indiquer à PostgreSQL de s'arrêter 30 secondes avant l'insertion de la dernière ligne relevée précédemment, et de mettre le rejeu en pause :

```
restore_command = 'rsync -a /var/lib/pgsql/10/backups/archived_wals/%f %p'
recovery_target_time = '<timestamp relevé moins 30 secondes>'
recovery_target_timeline = 3
recovery_target_action = 'pause'
```

- Démarrer l'instance - utiliser la commande appropriée au système utilisé ; par exemple sur CentOS 6 :

```
sudo service postgresql-10 start
```

- Vérifier le contenu de la table `suivi`, en récupérant le numéro de transaction :

```
SELECT xmin as xid, * FROM suivi ORDER BY 2 DESC LIMIT 1;
```

- Arrêter de nouveau l'instance : utiliser la commande appropriée au système utilisé
  - par exemple sur CentOS 6 :

```
sudo service postgresql-10 stop
```

- Modifier la configuration dans le `recovery.conf` pour avancer de 3 transactions par rapport au numéro de transaction repéré :

```
restore_command = 'rsync -a /var/lib/pgsql/10/backups/archived_wals/%f %p'
recovery_target_xid = '<XID relevé plus 3>'
recovery_target_timeline = 3
recovery_target_action = 'pause'
```

- Démarrer l'instance - utiliser la commande appropriée au système utilisé ; par exemple sur CentOS 6 :

```
sudo service postgresql-10 start
```

- Vérifier le contenu de la table `suivi`, en récupérant le numéro de transaction :

```
SELECT xmin as xid, * FROM suivi ORDER BY 2 DESC LIMIT 1;
```

- Sortir l'instance du mode pause :

```
SELECT pg_wal_replay_resume();
```

Dans les versions antérieures à PostgreSQL 9.5, le paramètre indiquant la mise en pause de l'instance en fin de recovery se nomme `pause_at_recovery_target`.

Point de cohérence :

- Déterminer un point dans le temps entre la date de début et celle de fin du dernier basebackup

  - Les instants de démarrage et de fin d'un basebackup sont disponibles dans un fichier spécial `backup_label`, disponible dans le répertoire des WAL archivés avec l'extension `.backup`

- Arrêter et supprimer l'instance : utiliser la commande appropriée au système utilisé ; par exemple sur CentOS 6 :

```
sudo service postgresql-10 stop
rm -rf ~postgres/10/data/*
rm -rf ~postgres/testtbs/*
```

- Restaurer à partir d'un backup antérieur au dernier backup :

```
cd ~
tar -zxvf ./10/backups/bkp_pitr/basebackup1_pgdata.tgz
tar -zxvf ./10/backups/bkp_pitr/basebackup1_testtbs.tgz
```

- Créer le fichier `recovery.conf` dans le répertoire PGDATA pour indiquer à PostgreSQL de s'arrêter à l'instant repéré précédemment :

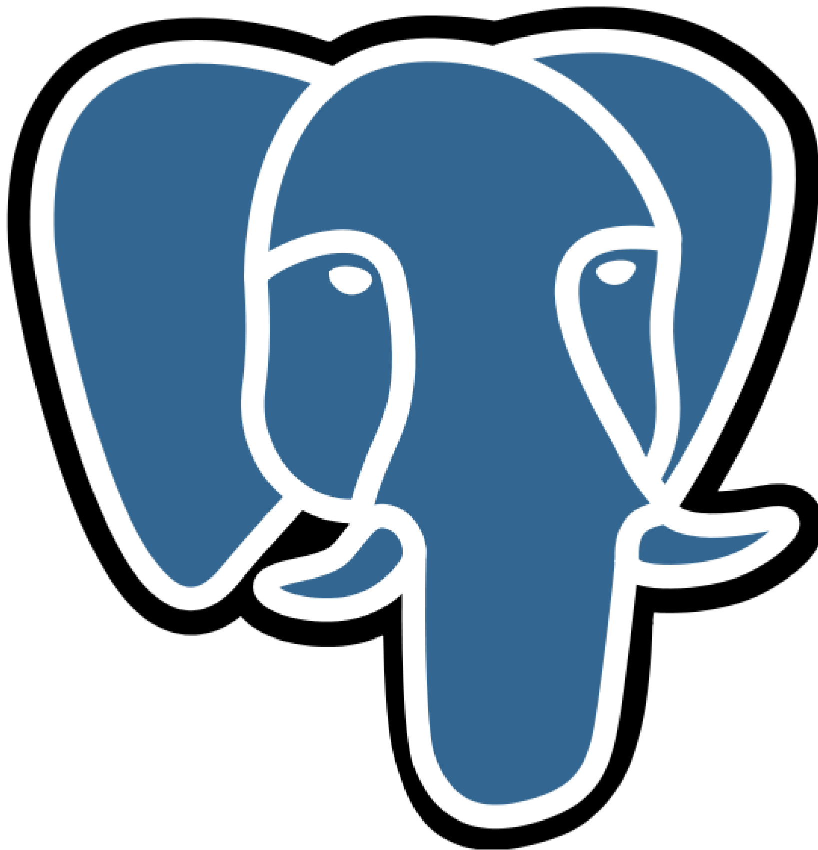
```
restore_command = 'rsync -a /var/lib/pgsql/10/backups/archived_wals/%f %p'
recovery_target_time =
            '<timestamp relevé entre le début et la fin du dernier backup>'
recovery_target_timeline = 3
```

- Démarrer l'instance - utiliser la commande appropriée au système utilisé ; par exemple sur CentOS 6 :

```
sudo service postgresql-10 start
```

- Vérifier le contenu de la table `suivi`.

3 POSTGRESQL : OUTILS DE SAUVEGARDE



3.1 INTRODUCTION

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

Nous avons vu le fonctionnement interne du mécanisme de sauvegarde physique. Celui-ci étant en place nativement dans le moteur PostgreSQL depuis de nombreuses versions,

sa robustesse n'est plus à prouver. Cependant, son industrialisation reste fastidieuse. Des outils tiers existent et vont permettre de faciliter la gestion des sauvegardes, de leur mise en place jusqu'à la restauration. Dans ce module nous allons voir en détail certains de ces outils et étudier les critères qui vont nous permettre de choisir la meilleure solution selon notre contexte.

3.1.1 AU MENU

- Présentation:
  - pg_back
  - pg_basebackup
  - Barman
  - pitrery
- Comment choisir ?

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

3.1.2 DÉFINITION DU BESOIN - CONTEXTE

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

Où les sauvegardes doivent-elles être stockées ?

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

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

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

3.2 PG_BACK - PRÉSENTATION

- Type de sauvegardes: **logiques** (pg_dump)
- Langage: **bash**
- Licence: **BSD** (libre)
- Type de stockage: **local**
- Planification: **crontab**
- OS: **Unix/Linux**
- Compression: **gzip**
- Versions compatibles: **toutes**
- Rétention: **durée** en jour

pg_back[1] est un outil écrit en bash par Nicolas Thauvin de Dalibo, également auteur de pitrery.

Il s'agit d'un script assez complet permettant de gérer simplement des sauvegardes logiques (`pg_dump`, `pg_dumpall`), y compris au niveau de la rétention.

L'outil se veut très simple et facile à adapter, il peut donc être facilement modifié pour mettre en place une stratégie de sauvegarde logique plus complexe.

L'outil ne propose pas d'options pour restaurer les données. Il faut pour cela utiliser les outils interne de PostgreSQL (`pg_restore`, `psql`).

3.3 PG_BASEBACKUP - PRÉSENTATION

- Outil intégré à PostgreSQL
- Prévu pour créer une instance secondaire
- Permet les sauvegardes PITR

pg_basebackup[2] est une application cliente intégrée à PostgreSQL, au même titre que pg_dump ou pg_ dumpall.

[1] https://github.com/orgrim/pg_back
[2] http://www.postgresql.org/docs/current/static/app-pgbasebackup.html

Elle est conçue pour permettre l'initialisation d'une instance secondaire en réplication. Cette opération consiste en une opération de sauvegarde PITR. pg_basebackup peut donc être utilisé pour faciliter l'opération de sauvegarde d'un instance.

3.3.1 PG_BASEBACKUP - FORMATS DE SAUVEGARDE

- `plain`
  - arborescence identique à l'instance sauvegardée
- `tar`
  - archive, permet la compression

Le format par défaut de la sauvegarde est `plain`, ce qui signifie que les fichiers seront créés tels quels dans le répertoire de destination (ou les répertoires en cas de tablespaces).

L'option `-F tar` active le format `tar`. pg_basebackup génère alors une archive `base.tar` pour le PGDATA de l'instance, puis une archive `<oid>.tar` par tablespace. Ce format permet l'activation de la compression `gzip` à l'aide de l'option `-z`. Le niveau de compression peut également être spécifié avec l'option `-Z <niveau>`.

3.3.2 PG_BASEBACKUP - AVANTAGES

- Sauvegarde possible à partir d'un secondaire
- Transfert des WAL pendant la sauvegarde
- Débit configurable
- Relocalisation des tablespaces
- Écriture d'un `recovery.conf`
  - prévu pour la réplication

L'un des avantages de pg_basebackup est qu'il permet nativement de réaliser une sauvegarde à partir d'une instance secondaire.

Autre avantage, avec pg_basebackup, il est possible de récupérer les fichiers WAL nécessaires à la restauration de la sauvegarde sans passer par la commande d'archivage.

Pour cela, il faut à l'exécution de la commande utiliser l'option `-x` (ou `-X fetch`) pour récupérer les WAL générés pendant la sauvegarde une fois celle-ci terminée, à condition qu'ils n'aient pas été recyclés entre-temps (ce qui peut nécessiter la configuration d'un slot de réplication, ou d'une configuration élevée du paramètre `wal_keep_segments` avant la 9.4).

Il est également possible de récupérer les WAL non pas en fin de sauvegarde mais en streaming pendant celle-ci, à l'aide de l'option `-X stream` - cela nécessite néanmoins l'utilisation d'un *wal sender* supplémentaire, le paramètre `max_wal_senders` doit donc être adapté en conséquence.

Il convient néanmoins de noter que si l'archivage des WAL n'est pas également actif par ailleurs, la sauvegarde effectuée ne sera utilisée que pour restaurer l'instance telle qu'elle était au moment de la fin de la sauvegarde - il ne sera pas possible de réaliser une restauration de type *Point In Time Recovery*.

Le débit de la sauvegarde est configurable avec l'option `-r` pour éviter d'impacter excessivement l'instance. Il convient de noter que si les fichiers WAL sont transférés en streaming (`-X stream`), ce flux ne sera pas affecté par la restriction de débit.

Il est également possible de relocaliser les éventuels tablespaces dans des répertoires distincts avec l'option `-T <ancien chemin>=<nouveau chemin>`, et de relocaliser le répertoire des fichiers WAL avec l'option `--waldir=<nouveau chemin>`.

L'option `-R` permet de demander à pg_basebackup de générer un fichier `recovery.conf` dans le répertoire PGDATA de la sauvegarde. Néanmoins, ce fichier ne contiendra que les paramètres nécessaires à l'activation de la réplication *streaming* (`standby_mode` et `primary_conninfo`), il n'est donc pas prévu en l'état pour effectuer une restauration.

3.3.3 PG_BASEBACKUP - LIMITATIONS

- Configuration de type réplication nécessaire
- Pas de configuration de l'archivage
- Pas d'association WAL archivés / sauvegarde
- Pas de gestion de la restauration
- Pas de politique de rétention

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

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

Si la sauvegarde est effectuée à partir d'une instance secondaire, les aspects suivants doivent être pris en compte :

- l'instance secondaire doit être configurée en `hot_standby` ;
- `max_wal_senders` doit avoir une valeur supérieure à `0` pour permettre à pg_basebackup de se connecter ;
- une attention particulère doit être apportée au fait que tous les fichiers WAL nécessaires à la restauration ont bien été archivés ;
- l'écriture complète des pages dans les WAL (paramètre `full_page_writes`) doit être activé.

L'archivage des fichiers WAL doit être configurée indépendamment à l'aide des commandes systèmes traditionnelles (`rsync`...).

Par ailleurs, la gestion des sauvegardes elles-mêmes n'est pas prévue dans l'outil. Notamment, pg_basebackup n'effectue pas de lien entre les WAL archivés et les sauvegardes effectuées, ne garde pas en mémoire les sauvegardes effectuées, ne permet pas de gérer de rétention.

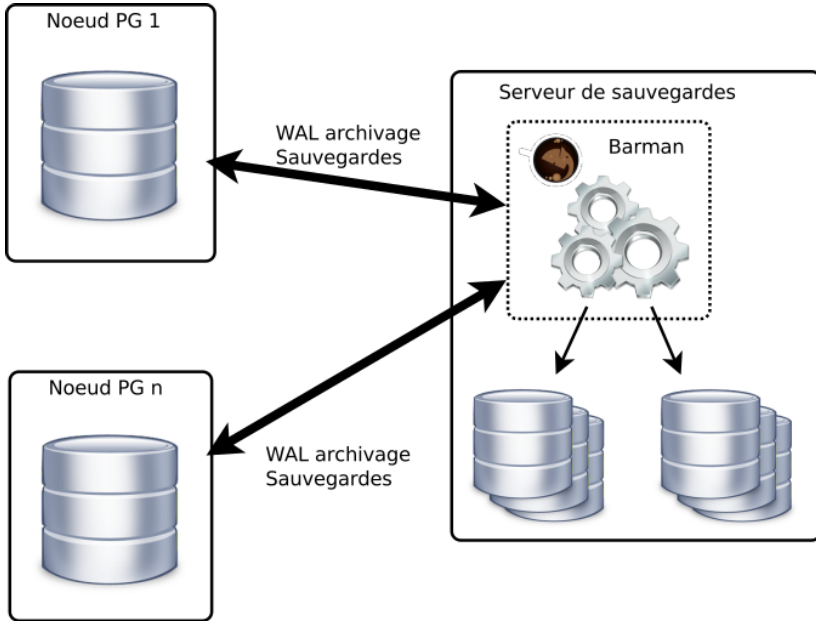
L'outil ne permet pas non plus d'effectuer directement de restauration. Il faut donc copier manuellement les fichiers à restaurer (par exemple avec `rsync`, ou `tar` si ce format a été sélectionné) vers le répertoire de destination de la restauration.

3.4 BARMAN - PRÉSENTATION GÉNÉRALE

- 2ndQuadrant IT
- Langage: **python** 2.6/2.7
- OS: **Unix/Linux**
- Versions compatibles: **>= 8.3**
- License: **GPL3** (libre)
- Type d'interface: **CLI** (ligne de commande)

`barman` est un outil développé avec le langage python, compatible avec les versions 2.6/2.7, compatible uniquement avec les environnements Linux/Unix. Il est principalement maintenu par la société 2ndQuadrant Italia et distribué sous license GPL3.

3.4.1 BARMAN - DIAGRAMME



3.4.2 BARMAN - SAUVEGARDES

- Type de sauvegardes: **physiques/PITR** (à chaud)
- Type de stockage: **local** ou **pull** (rsync/ssh)
- Planification: **crontab**
- Méthode: **pg_start_backup() / rsync / pg_stop_backup()**
- Incrémentales: **rsync + hardlink**
- Compression des WAL

Barman gère uniquement des sauvegardes physiques.

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

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

3.4.3 BARMAN - SAUVEGARDES (SUITE)

- Limitation du débit réseau lors des transferts
- Compression des données lors des transferts via le réseau
- Sauvegardes concurrentes via l'extension `pgespresso`
  - sans l'extension à partir de PostgreSQL 9.6
- Hook pre/post sauvegarde
- Hook pre/post archivage WAL
- Compression WAL : `gzip`, `bzip2`, `pigz`, `pbzip2`, etc..
- Pas de compression des données (sauf WAL)

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

Il peut être couplé à l'utilisation de l'extension `pgespresso`, permettant ainsi d'effectuer plusieurs sauvegardes en parallèle. La fonctionnalité proposée par cette extension a été intégrée à PostgreSQL en version 9.6, elle n'est donc plus nécessaire, et Barman a été modifié pour gérer ce cas.

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

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

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

3.4.4 BARMAN - POLITIQUE DE RÉTENTION

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

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

17.12

3.4.5 BARMAN - RESTAURATION

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

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

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

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

3.4.6 BARMAN - INSTALLATION

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

Barman est disponible sur le dépôt communautaire maintenu par la communauté PostgreSQL pour les systèmes d'exploitation disposant des gestionnaires de paquet APT (Debian, Ubuntu)[3] ou YUM (RHEL, CentOS, Fedora)[4] .

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

3.4.7 BARMAN - UTILISATION

```
usage: barman [-h] [-v] [-c CONFIG] [-q] [-d] [-f {console}]
              {cron,list-server,show-server,status,check,diagnose,
```

[3] http://apt.postgresql.org/pub/repos/apt/
[4] http://yum.postgresql.org/

```
                backup,list-backup,how-backup,list-files,recover,
                delete,rebuild-xlogdb}
[...]
optional arguments:
  -h, --help         show this help message and exit
  -v, --version      show program's version number and exit
  -c CONFIG, --config CONFIG
                     uses a configuration file (...)
  -q, --quiet        be quiet (default: False)
  -d, --debug        debug output (default: False)
  -f {console}, --format {console}
                     output format (default: console)
```

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

L'usage de ces différentes commandes sera détaillé ultérieurement.

L'option `-c` (ou `--config`) permet d'indiquer l'emplacement du fichier de configuration. L'option `-q` (ou `--quiet`) désactive l'envoi de messages sur la sortie standard. L'option `-f` (ou `--format`) n'offre pour le moment pas d'autre choix que `console`.

3.4.8 BARMAN - CONFIGURATION

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

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

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

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

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

3.4.9 BARMAN - CONFIGURATION UTILISATEUR

- Utilisateur système `barman`

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

3.4.10 BARMAN - CONFIGURATION SSH

- Utilisateur `postgres` pour les serveurs PostgreSQL
- Utilisateur `barman` pour le serveur de sauvegardes
- Générer les clefs SSH (RSA) des utilisateurs système `postgres` (serveurs PG) et `barman` (serveur barman)
- Échanger les clefs SSH public entre les serveurs PostgreSQL et le serveur de sauvegarde
- Établir manuellement une première connexion SSH entre chaque machine

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

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

3.4.11 BARMAN - CONFIGURATION POSTGRESQL

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

```
wal_level = 'replica'
archive_mode = on
archive_command = 'rsync -a %p barman@bkpsrv:<INCOMING_WALS_DIR>/%f'
```

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

DALIBO

Il est à noter que le paramètre `archive_mode` peut prendre la valeur `always` à partir de la version 9.5 pour permettre un archivage à partir des serveurs secondaires.

Attention à bien mettre le bon répertoire `<INCOMING_WALS_DIR>` pour l'emplacement de destination des WAL archivés. Il s'agit du répertoire indiqué par la commande suivante :

```
barman show-server <instance>
```

3.4.12 BARMAN - CONFIGURATION GLOBALE

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

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

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

3.4.13 BARMAN - CONFIGURATION SAUVEGARDES

- Configuration globale des options de sauvegarde

```
compression = gzip
reuse_backup = link
immediate_checkpoint = false
basebackup_retry_times = 0
basebackup_retry_sleep = 30
backup_options = exclusive_backup
```

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

- **reuse_backup** : paramétrage de la sauvegarde incrémentale (défaut : `off`)

  - `link` : déduplication de fichier basée sur des liens matériels (*hardlink*), économisant ainsi du temps et de l'espace disque ;
  - `copy` : copie locale des fichiers si identiques par rapport à la dernière sauvegarde, économisant ainsi du temps de transfert ;
  - `off` : sauvegarde complète.

- **immediate_checkpoint** : force la création immédiate d'un checkpoint impliquant une augmentation des écritures, le but étant de débuter la sauvegarde le plus rapidement possible (defaut : `off`) ;

- **basebackup_retry_times** : nombre de tentative d'écriture d'un fichier - utile pour relancer la copie d'un fichier en cas d'échec sans compromettre le déroulement global de la sauvegarde ;

- **basebackup_retry_sleep** : spécifié en secondes, il s'agit ici de l'intervalle de temps entre deux tentatives de copie d'un fichier en cas d'échec ;

- **backup_options** : `exclusive_backup` ou `concurrent_backup`, offre la possibilité d'activer la création de deux ou plusieurs sauvegardes de manière concurrente - nécessite l'installation de l'extension **pgespresso** avant la version 9.6 (défaut : `exclusive_backup`).

3.4.14 BARMAN - CONFIGURATION RÉSEAU

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

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

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

3.4.15 BARMAN - CONFIGURATION RÉTENTION

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

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

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

3.4.16 BARMAN - CONFIGURATION DES HOOKS

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

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

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

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

3.4.17 BARMAN - CONFIGURATION PAR INSTANCE

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

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

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

3.4.18 BARMAN - EXEMPLE CONFIGURATION PAR INSTANCE

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

```
[pgsrv]
description = "PostgreSQL Instance pgsrv"
ssh_command = ssh postgres@pgsrv
conninfo = host=pgsrv user=postgres dbname=postgres
backup_directory =
basebackup_directory =
wals_directory =
incoming_wals_directory =
```

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

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

- **description** : chaîne de caractère servant de descriptif de l'instance ;
- **ssh_command** : commande shell utilisée pour établir la connexion `ssh` vers le serveur hébergeant l'instance à sauvegarder ;
- **conninfo** : chaîne de connexion PostgreSQL ;
- **backup_directory** : définit l'emplacement du dossier parent contenant les sauvegardes et les journaux de transactions (défaut : `<barman_home>/<nom_section_instance>`) ;

DALIBO

- **basebackup_directory** : emplacement du dossier contenant les sauvegardes (défaut : <barman_home>/<nom_section_instance>/base) ;
- **wals_directory** : emplacement du dossier contenant les journaux de transactions archivés (défaut : <barman_home>/<nom_section_instance>/wals) ;
- **incoming_wals_directory** : emplacement du dossier contenant les journaux de transactions en attente d'archivage (défaut : <barman_home>/<nom_section_instance>/incoming)

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

3.4.19 BARMAN - VÉRIFICATION DE LA CONFIGURATION

- La commande `show-server` montre la configuration

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

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

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

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

C'est particulièrement utile lors d'un premier paramétrage, notamment pour récupérer la valeur assignée au paramètre `incoming_wals_directory` afin de configurer correctement la valeur du paramètre `archive_command` de l'instance à sauvegarder.

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

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

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

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

```
-bash-4.2$ barman show-server pgsrv
Server pgsrv:
      active: True
      archive_command: rsync %p barman@bkpsrv:/var/lib/barman/pgsrv/incoming/%f
      archive_mode: on
      archived_count: 88
```

```
backup_directory: /var/lib/barman/pgsrv
backup_options: BackupOptions(['exclusive_backup'])
bandwidth_limit: None
basebackup_retry_sleep: 30
basebackup_retry_times: 0
basebackups_directory: /var/lib/barman/pgsrv/base
compression: None
config_file: /var/lib/pgsql/9.4/data/postgresql.conf
conninfo: host=pgsrv user=postgres
copy_method: rsync
current_archived_wals_per_second: 0.00250231660903
current_xlog: 0000000A0000000100000063
custom_compression_filter: None
custom_decompression_filter: None
data_directory: /var/lib/pgsql/9.4/data
description: PostgreSQL Instance pgsrv
disabled: False
failed_count: 16
hba_file: /var/lib/pgsql/9.4/data/pg_hba.conf
ident_file: /var/lib/pgsql/9.4/data/pg_ident.conf
immediate_checkpoint: False
incoming_wals_directory: /var/lib/barman/pgsrv/incoming
is_archiving: True
last_archived_time: 2015-10-29 19:31:25.152625+01:00
last_archived_wal: 0000000A0000000100000062
last_backup_maximum_age: None
last_failed_time: 2015-10-29 19:30:58.923143+01:00
last_failed_wal: 0000000A0000000100000060
minimum_redundancy: 1
network_compression: False
pgespresso_installed: False
post_archive_retry_script: None
post_archive_script: None
post_backup_retry_script: None
post_backup_script: None
pre_archive_retry_script: None
pre_archive_script: None
pre_backup_retry_script: None
pre_backup_script: None
```

```
        recovery_options: RecoveryOptions([])
        retention_policy: REDUNDANCY 2
        retention_policy_mode: auto
        reuse_backup: None
        server_txt_version: 9.4.5
        ssh_command: ssh postgres@pgsrv
        stats_reset: 2015-10-29 09:46:00.034650+01:00
        tablespace_bandwidth_limit: None
        wal_level: hot_standby
        wal_retention_policy: MAIN
        wals_directory: /var/lib/barman/pgsrv/wals
```

Exemple de sortie de la commande `check` :

```
-bash-4.2$ barman check pgsrv
Server pgsrv:
        PostgreSQL: OK
        archive_mode: OK
        wal_level: OK
        archive_command: OK
        continuous archiving: OK
        directories: OK
        retention policy settings: OK
        backup maximum age: OK (no last_backup_maximum_age provided)
        compression settings: OK
        minimum redundancy requirements: OK (have 1 backups, expected at least 1)
        ssh: OK (PostgreSQL server)
        not in recovery: OK
```

3.4.20 BARMAN - STATUT

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

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

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

17.12

Les informations renvoyées sont, entre autres :

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

Exemple de sortie de la commande :

```
-bash-4.2$ barman status pgsrv
Server pgsrv:
      Description: PostgreSQL Instance pgsrv
      Active: True
      Disabled: False
      PostgreSQL version: 9.4.5
      pgespresso extension: Not available
      PostgreSQL Data directory: /var/lib/pgsql/9.4/data
      PostgreSQL 'archive_command' setting:
            rsync %p barman@bkpsrv:/var/lib/barman/pgsrv/incoming/%f
      Last archived WAL: 0000000A0000000100000062, at Thu Oct 29 19:31:25 2015
      Current WAL segment: 0000000A0000000100000063
      Failures of WAL archiver: 16
            (0000000A0000000100000060 at Thu Oct 29 19:30:58 2015)
      Server WAL archiving rate: 8.98/hour
      Retention policies: enforced
            (mode: auto, retention: REDUNDANCY 2, WAL retention: MAIN)
      No. of available backups: 1
      First available backup: 20151029T192840
      Last available backup: 20151029T192840
      Minimum redundancy requirements: satisfied (1/1)
```

3.4.21 BARMAN - DIAGNOSTIQUER

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

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

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

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

3.4.22 BARMAN - NOUVELLE SAUVEGARDE

- Pour déclencher une nouvelle sauvegarde

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

- Le détail de sauvegarde effectuée est affiché en sortie

La commande `backup` lance immédiatement une nouvelle sauvegarde, pour une seule instance si un identifiant est passé en argument, ou pour toutes les instances configurées si le mot clé `all` est utilisé.

Exemple de sortie de la commande :

```
-bash-4.2$ barman backup pgsrv
Starting backup for server pgsrv in /var/lib/barman/pgsrv/base/20151029T193737
Backup start at xlog location: 1/64000028 (0000000A0000000100000064, 00000028)
Copying files.
Copy done.
Asking PostgreSQL server to finalize the backup.
Backup size: 203.0 MiB
Backup end at xlog location: 1/64000128 (0000000A0000000100000064, 00000128)
Backup completed
Processing xlog segments for pgsrv
        0000000A0000000100000063
        0000000A0000000100000064
```

17.12

```
0000000A0000000100000064.00000028.backup
```

3.4.23 BARMAN - LISTER LES SAUVEGARDES

- Pour lister les sauvegardes existantes

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

- Affiche notamment la taille de la sauvegarde et des WAL associés

Liste les sauvegardes du catalogue, soit par instance, soit toutes si le mot clé `all` est passé en argument.

Exemple de sortie de la commande :

```
-bash-4.2$ barman list-backup pgsrv
pgsrv 20151029T193737 - Thu Oct 29 19:37:45 2015 - Size: 203.0 MiB -
                        WAL Size: 0 B (tablespaces: tmptbs:/tmp/tmptbs)
pgsrv 20151029T192840 - Thu Oct 29 19:28:48 2015 - Size: 203.0 MiB -
                        WAL Size: 48.0 MiB (tablespaces: tmptbs:/tmp/tmptbs)
```

3.4.24 BARMAN - DÉTAIL D'UNE SAUVEGARDE

- `show-backup` affiche le détail d'une sauvegarde (taille...)

```
$ sudo -u barman barman show-backup <instance> <ID-sauvegarde>
```

- `list-files` affiche le détail des fichiers d'une sauvegarde

```
$ sudo -u barman barman list-files <instance> <ID-sauvegarde>
```

La commande `show-backup` affiche toutes les informations relatives à une sauvegarde en particulier, comme l'espace disque occupé, le nombre de journaux de transactions associés, etc.

La commande `list-files` permet quant à elle d'afficher la liste complète des fichiers contenus dans la sauvegarde.

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

```
-bash-4.2$ barman show-backup pgsrv 20151029T192840
Backup 20151029T192840:
  Server Name            : pgsrv
  Status                 : DONE
```


DALIBO

```
PostgreSQL Version      : 90405
PGDATA directory        : /var/lib/pgsql/9.4/data
Tablespaces:
  tmptbs: /tmp/tmptbs (oid: 24583)

Base backup information:
  Disk usage            : 203.0 MiB (219.0 MiB with WALs)
  Incremental size      : 203.0 MiB (-0.00%)
  Timeline              : 10
  Begin WAL             : 0000000A0000000100000061
  End WAL               : 0000000A0000000100000061
  WAL number            : 1
  Begin time            : 2015-10-29 19:28:40.409406+01:00
  End time              : 2015-10-29 19:28:48.044205+01:00
  Begin Offset          : 40
  End Offset            : 240
  Begin XLOG            : 1/61000028
  End XLOG              : 1/610000F0

WAL information:
  No of files           : 3
  Disk usage            : 48.0 MiB
  WAL rate              : 37.67/hour
  Last available        : 0000000A0000000100000064

Catalog information:
  Retention Policy      : VALID
  Previous Backup       : - (this is the oldest base backup)
  Next Backup           : 20151029T193737
```

3.4.25 BARMAN - SUPPRESSION D'UNE SAUVEGARDE

- Pour supprimer manuellement une sauvegarde

```
$ sudo -u barman barman delete <instance> <ID-sauvegarde>
```

- Renvoie une erreur si la redondance minimale ne le permet pas

La suppression d'une sauvegarde nécessite de spécifier l'instance ciblée et l 'identifiant de la sauvegarde à supprimer.

Cet identifiant peut être trouvé en utilisant la commande Barman `list-backup`.

Si le nombre de sauvegardes (après suppression) ne devait pas respecter le seuil défini par la directive `minimum_redundancy`, la suppression ne sera alors pas possible.

3.4.26 BARMAN - TÂCHES DE MAINTENANCE

- La commande Barman `cron` déclenche la maintenance
  - récupération des WAL archivés
  - compression
  - politique de rétention
- Exemple

```
$ sudo -u barman barman cron
```

- À planifier pour une exécution régulière
  - par exemple avec la `crontab` Linux

La commande `cron` permet d'exécuter les tâches de maintenance qui doivent être exécutées périodiquement, telles que l'archivage des journaux de transactions (déplacement du dossier `incoming_wals/` vers `wals/`), ou la compression.

L'application de la politique de rétention est également faite dans ce cadre.

L'exécution de cette commande doit donc être planifiée via un cronjob, par exemple toutes les minutes.

3.4.27 BARMAN - RESTAURATION

- Copie/transfert de la sauvegarde
- Copie/transfert des journaux de transactions
- Génération du fichier `recovery.conf`
- Copie/transfert des fichiers de configuration

Le processus de restauration géré par Barman reste classique, mais nécessite tout de même quelques points d'attention.

En particulier, les fichiers de configuration sauvegardés sont restaurés dans le dossier `$PGDATA`, or ce n'est potentiellement pas le bon emplacement selon le type d'installation / configuration de l'instance. Dans une installation basée sur les paquets Debian/Ubuntu par exemple, les fichiers de configuration se trouvent dans

`/etc/postgresql/<version>/<instance>` et non dans le répertoire PGDATA. Il convient donc de penser à les supprimer du PGDATA s'ils n'ont rien à y faire avant de démarrer l'instance.

De même, la directive de configuration `archive_command` est passée à `false` par Barman. Une fois l'instance démarrée et fonctionnelle, il convient de modifier la valeur de ce paramètre pour réactiver l'archivage des journaux de transactions.

3.4.28 BARMAN - OPTIONS DE RESTAURATION

- Locale ou à distance
- Cibles : timeline, date, ID de transaction ou point de restauration
- Déplacement des tablespaces

Au niveau de la restauration, Barman offre la possibilité de restaurer soit en local (sur le serveur où se trouvent les sauvegardes), soit à distance.

Le cas le plus commun est une restauration à distance, car les sauvegardes sont généralement centralisées sur le serveur de sauvegarde d'où Barman est exécuté.

Pour la restauration à distance, Barman s'appuie sur la couche SSH pour le transfert des données.

Barman supporte différents types de cibles dans le temps pour la restauration :

- **timeline** : via l'option `--target-tli`, lorsqu'une divergence de timeline a eu lieu, il est possible de restaurer et rejouer toutes les transactions d'une timeline particulière ;
- **date** : via l'option `--target-time` au format **YYYY-MM-DD HH:MM:SS.mmm**, spécifie une date limite précise dans le temps au delà de laquelle la procédure de restauration arrête de rejouer les transactions ;
- **identifiant de transaction** : via l'option `--target-xid`, restauration jusqu'à une transaction précise ;
- **point de restauration** : via l'option `--target-name`, restauration jusqu'à un point de restauration créé préalablement sur l'instance via l'appel à la fonction `pg_create_restore_point(nom)`.

Barman permet également de relocaliser un tablespace lors de la restauration.

Ceci est utile lorsque l'on souhaite restaurer une sauvegarde sur un serveur différent, ne disposant pas des même points de montage des volumes que l'instance originelle.

3.4.29 BARMAN - EXEMPLE DE RESTAURATION À DISTANCE

- Exemple d'une restauration
  - déclenchée depuis le serveur Barman
  - avec un point dans le temps spécifié

```
$ sudo -u barman barman recover                      \
    --remote-ssh-command "ssh postgres@pgsrv"        \
    --target-time "2015-09-02 14:15:00"              \
    pgsrv 20150902T095027 /var/lib/pgsql/9.4/main
```

Dans cet exemple, nous souhaitons effectuer une restauration à distance via l'option `--remote-ssh-command`, prenant en argument `"ssh postgres@pgsrv"` correspondant à la commande SSH pour se connecter au serveur à restaurer.

L'option `--target-time` définit ici le point de restauration dans le temps comme étant la date "2015-09-02 14:15:00".

Les trois derniers arguments sont :

- l'identifiant de l'instance dans le fichier de configuration de Barman : `pgsrv` ;
- l'identifiant de la sauvegarde cible : `20150902T095027` ;
- et enfin le dossier PGDATA de l'instance à restaurer.

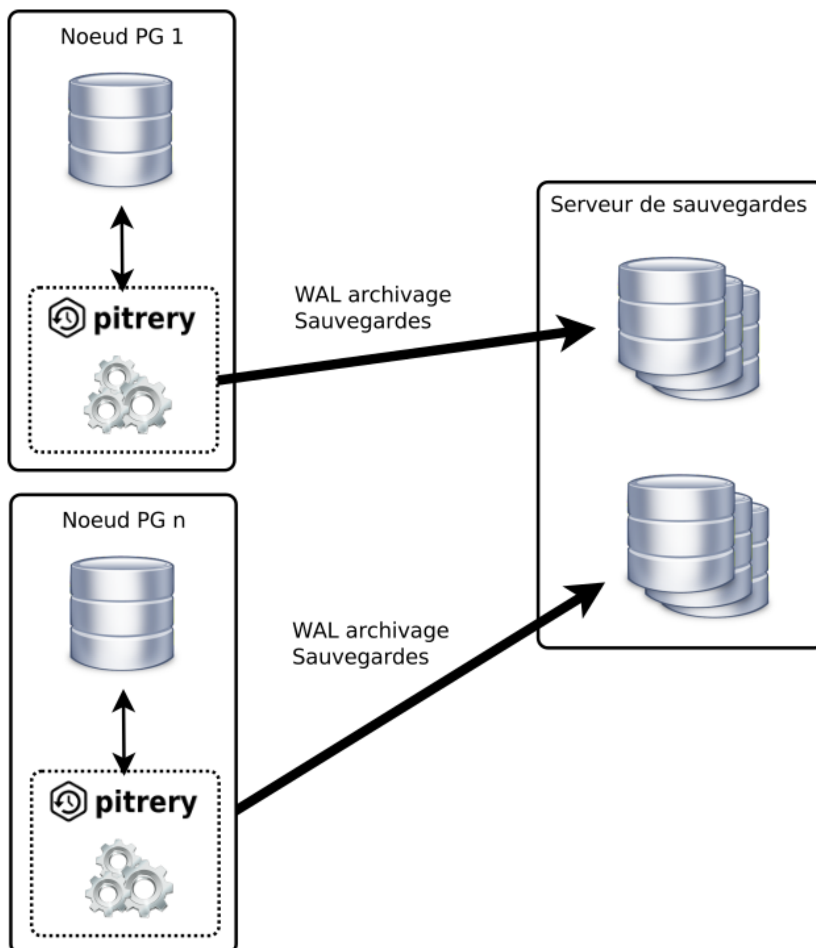
3.5 PITRERY - PRÉSENTATION GÉNÉRALE

- R&D Dalibo
- Langage : **bash**
- OS : **Unix/Linux**
- Versions compatibles : **>= 8.2**
- License : **BSD** (libre)
- Type d'interface : **CLI** (ligne de commande)

pitrery est un outil de gestion de sauvegarde physique et restauration PITR, écrit en bash, issu du labo R&D de Dalibo.

Il est compatible avec tous les environnements Unix/Linux disposant de l'interpréteur de shell `bash`, et supporte toutes les version de PostgreSQL depuis la 8.2.

3.5.1 PITRERY - DIAGRAMME



3.5.2 PITRERY - SAUVEGARDES

- Type de sauvegarde : **physiques/PITR**
- Type de stockage : **locale** ou **distant** (push) via `rsync/ssh`
- Planification : **crontab**

- Méthodes : **pg_start_backup() / rsync ou tar / pg_stop_backup()**
- Compression : **gzip**, **bzip2**, **pigz**, **pbzip2**, etc.
- Compression des WAL
- Scripts pre/post sauvegarde (hooks)
- Déduplication de fichier (`rsync` + lien matériel)

pitrery permet de gérer exclusivement des sauvegardes physiques.

Il s'installe sur le même serveur que l'instance à sauvegarder, et est capable de stocker les sauvegardes sur un point de montage local ou de les pousser au travers d'une connexion SSH vers un serveur de sauvegarde tiers.

La méthode de sauvegarde interne utilisée se base sur les appels de fonction SQL `pg_start_backup()` et `pg_stop_backup()`, les données étant effectivement copiées entre ces deux appels.

L'archivage des journaux de transactions doit être activé, et peut se faire en appelant un script interne à `pitrery`.

Les sauvegardes et les journaux de transactions peuvent être compressés, avec possibilité de spécifier l'outil de compression à utiliser.

pitrery propose également deux hooks (points d'ancrage) dans le code, offrant ainsi la possibilité d'exécuter des scripts externes avant ou après l'exécution d'une nouvelle sauvegarde.

Concernant le stockage des sauvegardes, il est possible d'activer la déduplication des fichiers entre deux sauvegardes.

Ceci est possible en utilisant `rsync` comme format de sauvegarde. Dans ce cas, pitrery va tirer parti de la capacité de `rsync` à effectuer une sauvegarde différentielle combinée avec la création de liens matériels (hardlink), ce qui permet selon les cas d'économiser de l'espace disque et du temps de sauvegarde de manière conséquente.

La compression de la sauvegarde est également possible si l'on a choisi `tar` comme format de sauvegarde.

3.5.3 PITRERY - POLITIQUE DE RÉTENTION

- **Durée** (en jours)
- **Nombre** de sauvegardes

La politique de rétention des sauvegardes se définit en nombre de jours ou de sauvegardes.

La combinaison des deux paramétrages est possible.

Dans ce cas, pitrery déclenchera la suppression d'une sauvegarde **si et seulement si** les deux rétentions sont dépassées.

Ainsi, si la durée spécifiée est de sept jours, et le nombre de sauvegardes est de trois, alors, au moment de supprimer une sauvegarde antérieure à sept jours, pitrery vérifiera qu'il reste bien au moins trois sauvegardes disponibles.

3.5.4 PITRERY - RESTAURATION

- À partir des données locales ou distantes
- Point dans le temps : date

Comme dans le cadre de la sauvegarde, lors de la restauration, pitrery est capable de transférer les données depuis un serveur tiers.

Il est également possible de spécifier un point dans le temps pour effectuer une restauration de type PITR.

3.5.5 PITRERY - INSTALLATION

- Téléchargement depuis le site du projet
  - installation depuis les sources (tarball)
  - paquets RPM et DEB disponibles

L'installation est très simple.

Pour installer depuis les sources, il faut d'abord télécharger la dernière version stable depuis le site du projet[5] , désarchiver le fichier, puis installer `pitrery` via la commande `make install`, à exécuter avec les droits root.

Par défaut, les scripts pitrery sont installés dans le dossier `/usr/local/bin`, et le fichier de configuration se trouve dans le dossier `/usr/local/etc/pitrery`.

Exemple détaillé :

```
wget https://github.com/dalibo/pitrery/archive/v1.10.tar.gz
tar xvzf v1.10.tar.gz
cd pitrery-1.10
sudo make install
```

[5] https://dalibo.github.io/pitrery/downloads.html

17.12

Des paquets RPM et DEB sont également disponibles dans la page de téléchargement.

3.5.6 PITRERY - UTILISATION

```
usage: pitrery [options] action [args]
options:
    -c file       Path to the configuration file
    -n            Show the command instead of executing it
    -V            Display the version and exit
    -?            Print help

actions:
    list
    backup
    restore
    purge
```

L'option `-c` permet de définir l'emplacement du fichier de configuration. Par défaut, celui-ci se trouve ici sous `/usr/local/etc/pitrery/pitr.conf`.

Le script `pitrery` étant un wrapper appelant d'autres scripts en fonction de l'action demandée, l'option `-n` permet d'afficher sur la sortie standard, sans l'exécuter, l'appel au script devant effectuer l'action.

L'option `-V` affiche la version de l'outil et se termine.

L'option `-?` permet d'afficher le message d'aide, et peut être combinée avec une commande pour afficher l'aide spécifique à la commande (par exemple, `pitrery restore -?`).

Les actions possibles sont les suivantes :

- **list** pour lister le contenu du catalogue de sauvegarde ;
- **backup** pour effectuer une sauvegarde ;
- **restore** pour effectuer une restauration ;
- **purge** pour supprimer les sauvegardes et journaux de transaction ne satisfaisant plus la politique de rétention.

3.5.7 PITRERY - CONFIGURATION POSTGRESQL

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

```
archive_mode = on
wal_level = replica
archive_command = '/usr/local/bin/archive_xlog %p'
```

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

Il est à noter que le paramètre `archive_mode` peut prendre la valeur `always` à partir de la version 9.5 pour permettre un archivage à partir des serveurs secondaires.

pitrery fournit un script permettant de simplifier la configuration de la commande d'archivage. Pour l'utiliser, il faut configurer le paramètre `archive_command` pour qu'il appelle le script `archive_xlog`, suivi de l'argument `%p` qui correspond au dossier contenant les journaux de transactions de l'instance.

3.5.8 PITRERY - CONFIGURATION SSH

- Stockage des sauvegardes sur un serveur distant
- Générer les clefs SSH (RSA) des utilisateurs système `postgres` sur chacun des serveurs (nœud PG et serveur de sauvegarde)
- Échanger les clefs SSH publiques entre les serveurs PostgreSQL et le serveur de sauvegarde
- Établir manuellement une première connexion SSH entre chaque serveur

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

Afin d'automatiser ces transferts reposant sur le protocole SSH, il est impératif d'autoriser l'authentification SSH par clé et d'échanger les clés publiques des différents serveurs.

3.5.9 PITRERY - FICHIER DE CONFIGURATION

- Emplacement par défaut: `/usr/local/etc/pitrery/pitr.conf`
- Possibilité de spécifier un autre emplacement

Il est possible de spécifier un fichier de configuration spécifique à utiliser à l'appel d'un script.

L'option à utiliser varie en fonction du script utilisé:

- `-c <fichier>` pour le script principal `pitrery` ;
- `-C <fichier>` pour les autres scripts.

3.5.10 PITRERY - CONFIGURATION CONNEXION POSTGRESQL

- Options de connexion à l'instance

```
PGPSQL="psql"
PGUSER="postgres"
PGPORT=5432
PGHOST="/var/run/postgresql"
PGDATABASE="postgres"
```

Ces paramètres sont utilisés par pitrery pour établir une connexion à l'instance afin d'exécuter les fonctions `pg_start_backup()` et `pg_stop_backup()` utilisées pour signaler à PostgreSQL qu'une sauvegarde PITR démarre et se termine.

- **PGPSQL** : chemin vers le binaire du client `psql` ;
- **PGUSER** : superutilisateur PostgreSQL de connexion ;
- **PGPORT** : numéro de port d'écoute PostgreSQL ;
- **PGHOST** : hôte PostgreSQL (dossier contenant le socket UNIX de connexion ou adresse IP ou alias) ;
- **PGDATABASE** : nom de la base de données de connexion.

3.5.11 PITRERY - LOCALISATION

- Configurer les emplacements
- Pour l'instance à sauvegarder

```
PGDATA="/var/lib/pgsql/9.4/data"
PGXLOG=
```

- Pour la destination des sauvegardes

```
BACKUP_DIR="/var/lib/pitrery"
```

- Pour la destination des WAL archivés

```
ARCHIVE_DIR="$BACKUP_DIR/archived_xlog"
```

- **PGDATA** : répertoire contenant les données de l'instance PostgreSQL à sauvegarder ;
- **PGXLOG** : répertoire contenant les journaux de transactions de PostgreSQL, à laisser vide s'il se trouve à la racine du `PGDATA` ;
- **BACKUP_DIR** : répertoire destiné à contenir l'arborescence des sauvegardes PITR ;
- **ARCHIVE_DIR** : répertoire destiné à contenir les journaux de transactions archivés.

Il est possible d'utiliser des paramètres initialisés précédemment comme variables lors de l'initialisation des paramètres suivants.

3.5.12 PITRERY - CONFIGURATION DU MODE DE SAUVEGARDE

- Paramètre de configuration `STORAGE`
- Deux modes possibles :
  - `tar` : sauvegarde complète, éventuellement compressée
  - `rsync` : sauvegarde complète ou différentielle, pas de compression

pitrery permet de spécifier deux modes de sauvegarde.

Le premier, `tar`, réalise la sauvegarde sous la forme d'une archive `tar` du répertoire `PGDATA`, éventuellement compressée, plus autant d'archives `tar` qu'il y a de tablespaces dans l'instance. Ce mode ne permet que les sauvegardes complètes.

Le second, `rsync`, indique à pitrery de copier les fichiers de l'instance à l'aide de la commande système idoine. Les fichiers dans le répertoire de la sauvegarde seront donc exactement les mêmes que ceux de l'instance :

- le contenu du répertoire `PGDATA` de l'instance est dans un sous-répertoire `pgdata` ;
- le contenu des différents tablespaces est dans un sous-répertoire `tblspc/<nom du tblspc>`.

Ce mode ne permet pas la compression, il consommera donc par défaut beaucoup plus de place que la sauvegarde `tar`. En revanche, ce mode permet la réalisation de sauvegarde différentielles, en réalisant des liens matériels (hardlink) des fichiers depuis une précédente sauvegarde et en utilisant la fonctionnalité de `rsync` basée sur les checksums pour

ne sauvegarder que les fichiers ayant été modifiés. Si l'instance a eu peu de modifications depuis la précédente sauvegarde, ce mode peut donc faire économiser du temps de sauvegarde et de l'espace disque.

3.5.13 PITRERY - CONFIGURATION DE LA RÉTENTION

- Configuration de la rétention en nombre de sauvegardes

```
PURGE_KEEP_COUNT=3
```

- Ou en jours

```
PURGE_OLDER_THAN=7
```

- Les deux paramètres peuvent être combinés

Le paramétrage de la rétention peut s'effectuer à deux niveaux :

- **PURGE_KEEP_COUNT** : nombre minimal de sauvegarde à conserver ;
- **PURGE_OLDER_THAN** : âge minimal des sauvegardes à conserver, exprimé en jours.

Si les deux paramètres sont utilisés de concert, une sauvegarde ne sera supprimée que si elle ne répond à aucune des deux rétentions.

Ainsi, avec la configuration faite ici, une sauvegarde datant de plus de sept jours ne sera supprimée que s'il reste effectivement au moins trois autres sauvegardes.

3.5.14 PITRERY - CONFIGURATION DE L'ARCHIVAGE

- L'archivage peut être configuré pour :
  - archiver en local (montage NFS…)
  - archiver vers un serveur distant en SSH
- Exemple

```
ARCHIVE_LOCAL="no"
ARCHIVE_HOST=bkpsrv
ARCHIVE_USER=pitrery
ARCHIVE_COMPRESS="yes"
```

- Utilisé par la commande `archive_xlog`

Ce paramétrage est utilisé lors de l'appel au script `archive_xlog` par l'instance PostgreSQL :

- **ARCHIVE_LOCAL** : `yes` ou `no`, définit si les journaux sont archivés localement ou sur un serveur de sauvegarde distant (rsync/ssh) ;
- **ARCHIVE_HOST** : hôte stockant les journaux de transactions archivés, si `ARCHIVE_LOCAL = no` ;
- **ARCHIVE_USER** : utilisateur SSH de connexion vers le serveur distant pour le transfert des WAL, si `ARCHIVE_LOCAL = no` ;
- **ARCHIVE_COMPRESS** : compression des journaux de transactions lors de l'archivage.

3.5.15 PITRERY - CONFIGURATION DE LA COMPRESSION

- Configuration de la compression
  - des WAL archivés par la commande `archive_xlog`
  - des sauvegardes effectuées au format `tar`
- Exemple

```
COMPRESS_BIN=
COMPRESS_SUFFIX=
UNCOMPRESS_BIN=
BACKUP_COMPRESS_BIN=
BACKUP_COMPRESS_SUFFIX=
BACKUP_UNCOMPRESS_BIN=
```

Les paramètres indiqués ici permettent de spécifier des commandes spécifiques pour compresser / décompresser les journaux de transactions archivés et les sauvegardes :

- **COMPRESS_BIN** : chemin vers le binaire (+ ses options) utilisé pour la compression des fichiers WAL archivés (`gzip`, `bzip2`, etc), `gzip -4` par défaut ;
- **COMPRESS_SUFFIX** : l'extension du fichier contenant les fichiers WAL archivés compressés, `gz` par défaut, utilisé pour la décompression ;
- **UNCOMPRESS_BIN** : chemin vers l'outil utilisé pour décompresser les fichiers WAL archivés, `gunzip` par défaut ;
- **BACKUP_COMPRESS_BIN** : chemin vers le binaire (+ ses options) utilisé pour la compression des sauvegardes (`gzip`, `bzip2` etc), `gzip -4` par défaut ;
- **BACKUP_COMPRESS_SUFFIX** : l'extension du fichier compressé de la sauvegarde ;
- **BACKUP_UNCOMPRESS_BIN** : chemin vers l'outil utilisé pour décompresser les sauvegardes, `gunzip` par défaut.

Il est possible d'utiliser des outils de compression multithread/multiprocess comme `pigz` ou `pbzip2`.

À noter que la compression de la sauvegarde n'est possible que si la méthode de stockage `tar` est utilisée.

3.5.16 PITRERY - CONFIGURATION DES TRACES

- Activer l'horodatage des traces

```
LOG_TIMESTAMP="yes"
```

- Utiliser syslog pour les traces de l'archivage (`archive_xlog`)

```
SYSLOG="no"
SYSLOG_FACILITY="local0"
SYSLOG_IDENT="postgres"
```

Ces paramètres permettent d'activer la redirection des traces vers `syslog` pour les opérations d'archivage et de restauration des journaux de transactions.

Les autres opérations sont écrites vers les sorties standards.

3.5.17 PITRERY - CONFIGURATION DE HOOKS

- Exécuter un script avant ou après une sauvegarde

```
PRE_BACKUP_COMMAND=
POST_BACKUP_COMMAND=
```

Ces paramètres permettent de spécifier des commandes à exécuter avant ou après une sauvegarde :

- **PRE_BACKUP_COMMAND** : exécutée avant le début de la sauvegarde ;
- **POST_BACKUP_COMMAND** : exécutée après la fin de la sauvegarde.

3.5.18 PITRERY - EFFECTUER UNE SAUVEGARDE

- Pour déclencher une nouvelle sauvegarde

```
$ sudo -u postgres /usr/local/bin/pitrery backup
```

- La plupart des paramètres peuvent être surchargés
- Par exemple, l'option `-s` permet de spécifier un mode, `tar` ou `rsync`

L'exécution de la commande pitrery `backup` déclenche la création immédiate d'une nouvelle sauvegarde de l'instance.

Toutes les commandes `pitrery` doivent être exécutées depuis un utilisateur système ayant un accès en lecture aux fichiers de données de l'instance PostgreSQL.

En général, on utilisera le compte de service PostgreSQL, par défaut `postgres`.

Exemple de sortie de la commande pour une sauvegarde `rsync` :

```
-bash-4.2$ pitrery backup
INFO: preparing directories in bkpsrv:/var/lib/pitrery/backups/pitr
INFO: listing tablespaces
INFO: starting the backup process
INFO: backing up PGDATA with rsync
INFO: preparing hardlinks from previous backup
INFO: transfering data from /var/lib/pgsql/9.4/data
INFO: backing up tablespace "tmptbs" with rsync
INFO: preparing hardlinks from previous backup
INFO: transfering data from /tmp/tmptbs
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
      bkpsrv:/var/lib/pitrery/backups/pitr/2015.10.29_22.08.11
INFO: done
```

On voit à la ligne suivante qu'il s'agit d'une sauvegarde différentielle, seuls les fichiers ayant été modifiés depuis la sauvegarde précédente ont réellement été transférés :

```
INFO: preparing hardlinks from previous backup
```

3.5.19 PITRERY - SUPPRESSION DES SAUVEGARDES OBSOLÈTES

- Suppression des sauvegardes et archives ne satisfaisant plus la politique de sauvegarde

```
$ sudo -u postgres /usr/local/bin/pitrery purge
```

- À déclencher systématiquement après une sauvegarde

17.12

La commande `purge` se charge d'appliquer la politique de rétention et de supprimer les sauvegardes les plus anciennes, ainsi que les journaux de transactions devenus obsolètes.

3.5.20 PITRERY - LISTER LES SAUVEGARDES

- Lister les sauvegardes présentes et leur taille

```
$ sudo -u postgres /usr/local/bin/pitrery list
```

- L'option `-v` permet d'avoir plus de détails
  - mode de sauvegarde
  - date minimale utilisable pour la restauration
  - taille de chaque tablespace

La commande pitrery `list` énumère les sauvegardes présentes, indiquant l'emplacement, la taille, la date et l'heure de la création de chacune d'elles.

L'option `-v` permet d'afficher des informations plus détaillées, notamment la date minimale utilisable pour une restauration effectuée à partir de cette sauvegarde, ce qui est très pratique pour savoir précisément jusqu'à quel point dans le temps il est possible de remonter en utilisant les sauvegardes présentes.

Exemple de sortie de la commande :

```
-bash-4.2$ pitrery list
List of backups on bkpsrv
/var/lib/pitrery/backups/pitr/2015.10.29_22.09.52  36M   2015-10-29 22:09:52 CET
/var/lib/pitrery/backups/pitr/2015.10.29_22.17.15  240M  2015-10-29 22:17:15 CET
/var/lib/pitrery/backups/pitr/2015.10.29_22.28.48  240M  2015-10-29 22:28:48 CET
```

Avec l'option `-v` :

```
-bash-4.2$ pitrery list -v
List of backups on bkpsrv
----------------------------------------------------------------------
Directory:
  /var/lib/pitrery/backups/pitr/2015.10.29_22.09.52
  space used: 36M
  storage: tar with gz compression
Minimum recovery target time:
  2015-10-29 22:09:52 CET
PGDATA:
  pg_default 202 MB
```

```
  pg_global 469 kB
Tablespaces:
  "tmptbs" /tmp/tmptbs (24583) 36 MB

----------------------------------------------------------------------
Directory:
  /var/lib/pitrery/backups/pitr/2015.10.29_22.17.15
  space used: 240M
  storage: rsync
Minimum recovery target time:
  2015-10-29 22:17:15 CET
PGDATA:
  pg_default 202 MB
  pg_global 469 kB
Tablespaces:
  "tmptbs" /tmp/tmptbs (24583) 36 MB

----------------------------------------------------------------------
Directory:
  /var/lib/pitrery/backups/pitr/2015.10.29_22.28.48
  space used: 240M
  storage: rsync
Minimum recovery target time:
  2015-10-29 22:28:48 CET
PGDATA:
  pg_default 202 MB
  pg_global 469 kB
Tablespaces:
  "tmptbs" /tmp/tmptbs (24583) 36 MB
```

3.5.21 PITRERY - PLANIFICATION

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

```
00 00 * * * (    /usr/local/bin/pitrery backup                  \
             && /usr/local/bin/pitrery purge)                   \
```

```
>> /var/log/postgresql/pitrery-$(date +\%Y-\%m-\%d).log 2>&1
```

La planification des sauvegardes et de la suppression des sauvegardes ne respectant plus la politique de rétention peut être faite par n'importe quel outil de planification de tâches, le plus connu étant `cron`.

L'exemple montre la planification d'une sauvegarde suivie de la suppression des sauvegardes obsolètes, via la crontab de l'utilisateur système `postgres`.

La sauvegarde est effectuée tous les jours à minuit, et est suivie d'une purge seulement si elle a réussi.

Les traces (stdout et stderr) sont redirigées vers un fichier journalisé par jour.

3.5.22 PITRERY - RESTAURATION

- Effectuer une restauration

```
$ sudo -u postgres /usr/local/bin/pitrery restore
```

- Nombreuses options à la restauration, notamment
  - l'option `-D` permet de modifier la cible du PGDATA
  - l'option `-t` permet de modifier la cible d'un tablespace
  - l'option `-x` permet de modifier la cible du `pg_wal`
  - l'option `-d` permet de spécifier une date de restauration

La restauration d'une sauvegarde se fait par l'appel à la commande pitrery `restore`.

Plusieurs options peuvent être spécifiées, offrant notamment la possibilité de restaurer le PGDATA (option `-D`), les tablespaces (option `-t`) et le pg_wal (option `-x`) dans des répertoires spécifiques.

Il est également possible de restaurer à une date précise (option `-d`), à condition que la date spécifiée soit postérieure à la date minimale utilisable de la plus ancienne sauvegarde disponible.

Voici un exemple combinant ces différentes options :

```
-bash-4.2$ pitrery restore -d '2015-10-29 22:10:00' \
                           -D /var/lib/pgsql/9.4/data2/ \
                           -x /var/lib/pgsql/9.4/pg_wal2/ \
                           -t tmptbs:/tmp/tmptbs2/
INFO: searching backup directory
INFO: searching for tablespaces information
```

```
INFO:
INFO: backup directory:
INFO:   /var/lib/pitrery/backups/pitr/2015.10.29_22.28.48
INFO:
INFO: destinations directories:
INFO:   PGDATA -> /var/lib/pgsql/9.4/data2/
INFO:   PGDATA/pg_wal -> /var/lib/pgsql/9.4/pg_wal2/
INFO:   tablespace "tmptbs" (24583) -> /tmp/tmptbs2/ (relocated: yes)
INFO:
INFO: recovery configuration:
INFO:   target owner of the restored files: postgres
INFO:   restore_command = '/usr/bin/restore_xlog -h bkpsrv -u pitrery \
                           -d /var/lib/pitrery/archived_wal %f %p'
INFO:   recovery_target_time = '2015-10-29 22:10:00'
INFO:
INFO: checking if /var/lib/pgsql/9.4/data2/ is empty
INFO: setting permissions of /var/lib/pgsql/9.4/data2/
INFO: checking if /var/lib/pgsql/9.4/pg_wal2/ is empty
INFO: setting permissions of /var/lib/pgsql/9.4/pg_wal2/
INFO: checking if /tmp/tmptbs2/ is empty
INFO: setting permissions of /tmp/tmptbs2/
INFO: transfering PGDATA to /var/lib/pgsql/9.4/data2/ with rsync
INFO: transfer of PGDATA successful
INFO: transfering PGDATA to /var/lib/pgsql/9.4/data2/ with rsync
INFO: transfer of tablespace "tmptbs" successful
INFO: creating symbolic link pg_wal to /var/lib/pgsql/9.4/pg_wal2/
INFO: preparing pg_xlog 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:
```

Par cette commande, nous demandons à pitrery de :

- relocaliser le PGDATA dans `/var/lib/pgsql/9.4/data2/` ;
- relocaliser le pg_wal dans `/var/lib/pgsql/9.4/pg_wal2/` ;
- relocaliser le tablespace `tmptbs` dans `/tmp/tmpbbs2/` ;
- inscrire dans le `recovery.conf` que la restauration ne devra pas inclure les transac-

tions au delà du 29 octobre 2015 à 22h10.

pitrery se charge de trouver automatiquement la sauvegarde la plus à jour permettant la restauration de l'instance à cette date précise.

3.6 AUTRES OUTILS DE L'ÉCOSYSTÈME

- De nombreux autres outils existent
- Moins importants que les précédents
  - répondant à des problématiques plus spécifiques
  - parfois anciens et potentiellement obsolètes
  - ou très récents et pas encore stables

Du fait du dynamisme du projet, l'écosystème des outils autour de PostgreSQL est très changeant.

En conséquence, on trouve de très nombreux projets autour du thème de la gestion des sauvegardes.

Certains de ces projets répondaient à des problématiques très spécifiques (WAL-e, walmgr), d'autres sont assez anciens, pas forcément très actifs et souvent dépassés par des projets plus récents (pg_rman, pg_backman).

On trouve néanmoins des nouveaux projets prometteurs (pgBackRest), néanmoins ils sont souvent trop jeunes pour être jugés stables.

3.6.1 PGBACKMAN - PRÉSENTATION

- Gestion de sauvegardes logiques
- Fonctionnalités limités
- Développement arrêté

pgBackMan[6] est un outil d'archivage des sauvegardes produites via pg_dump et pg_dumpall.

Son utilité est donc réduite à la gestion des sauvegardes logiques.

L'outil est très simple, voire limité. De plus, son auteur semble inactif depuis plus d'un an.

Dans le même registre, le projet pg_back est bien plus complet, et mieux maintenu.

[6] http://www.pgbackman.org/

3.6.2 WALMGR - PRÉSENTATION

- Simplifie la réplication *Warm Standby*
- Membre de la suite Skytools
- Permet la gestion de sauvegardes PITR
  - manque de fonctionnalités
  - développement arrêté

walmgr fait partie de la suite Skytools[7] développée par Skype.

C'est outil conçu pour simplifier la gestion de la réplication Warm Standby et les sauvegardes PITR, en automatisant les opérations de mise en place et de bascule.

walmgr était un outil très utile entre 2007 et 2010.

L'arrivée de la génération PostgreSQL 9.x avec la réplication en *streaming* et le *Hot Standby* a considérablement réduit l'intérêt de cette solution.

Aujourd'hui le projet n'est plus actif, la documentation n'est plus disponible et l'avenir du logiciel est incertain.

3.6.3 WAL-E - PRÉSENTATION

- Gestion de sauvegardes PITR
- Focus sur le stockage en *cloud*
- Possibilités de restauration limitées
  - pas de génération du `recovery.conf`
  - pas de re-création des liens vers les tablespaces

WAL-E[8] est un programme très populaire pour gérer l'archivage continu des fichiers WAL et des sauvegardes à plat (*base backups*).

De par sa conception, il est très adapté pour l'archivage des journaux de transactions vers des stockages *cloud*.

Il reste néanmoins assez limité dans les fonctionnalités offertes, notamment au niveau des possibilités de restauration.

Des outils comme Barman ou pitrery sont donc généralement plus appropriés.

[7] https://wiki.postgresql.org/wiki/SkyTools
[8] https://github.com/wal-e/wal-e

3.6.4 PG_RMAN - PRÉSENTATION

- Gestion de sauvegardes PITR
- Développé par NTT
- Peu de documentation
- Peu d'exemples d'utilisation connus

pg_rman[9] , développé par NTT, est un outil directement inspiré par Oracle RMAN dont le but est d'automatiser la gestion des sauvegardes de type PITR.

L'outil maintient un catalogue des sauvegardes disponibles et facilite leur restauration.

pg_rman est un projet actif mais peu dynamique. Même s'il existe une documentation minimale, il y a peu d'exemple d'utilisation en production.

Il est plutôt conseillé d'utiliser des outils plus complets et plus pérennes comme pitrery ou Barman.

3.6.5 OMNIPITR - PRÉSENTATION

- Gestion de l'archivage et du basebackup
- Mise en place de réplication par log-shipping
- Outil obsolète :
  - gestion du log-shipping en interne depuis PostgreSQL 9.0 (2010)
  - pas de génération de `recovery.conf`

OmniPITR, contrairement à ce qu'on pourrait déduire de son nom, ne permet pas de faire du PITR, mais plutôt de mettre en place de la réplication par log-shipping. Il est surtout utile pour les versions de PostgreSQL inférieures ou égales à la 8.4.

Il reste donc limité dans les fonctionnalités offertes au niveau de la restauration. Des outils comme Barman ou pitrery sont donc généralement plus appropriés.

3.6.6 PGBACKREST - PRÉSENTATION

- Gestion de sauvegardes PITR

[9] https://github.com/ossc-db/pg_rman

- Indépendant des commandes système
  - utilise un protocole dédié
- Sauvegardes complètes, différentielles ou incrémentales
- Gestion du multi-thread
- Projet récent (2014), non stabilisé

pgBackRest[10] est un outil de gestion de sauvegardes PITR écrit en perl, en cours de développement par David Steele de Crunchy Data.

Les principales fonctionnalités mises en avant sont :

- un protocole dédié pour le transfert / compression des données ;
- des opérations parallélisables en multi-thread ;
- la possibilité de réaliser des sauvegardes complètes, différentielles et incrémentielles ;
- la possibilité de réaliser l'archivage des WAL de façon asynchrone.

Le projet est actif et les fonctionnalités proposées sont intéressantes.

Cet outil est néanmoins très récent, et n'a pas encore été stabilisé, il est donc préconçu de l'utiliser en environnement de production.

3.7 CONCLUSION

- Des outils pour vous aider!
- Pratiquer, pratiquer et pratiquer
- Superviser les sauvegardes!

Nous venons de vous présenter des outils qui vont vous permettre de vous simplifier la tâche dans la mise en place d'une solution de sauvegarde fiable et robuste de vos instance PostgreSQL.

Cependant, leur maîtrise passera par de la pratique, et en particulier, la pratique de la restauration.

Le jour où la restauration d'une instance de production se présente, ce n'est généralement pas une situation confortable à cause du stress lié à une perte/corruption de données, interruption du service, etc. Autant maîtriser les outils qui vous permettront de sortir de ce mauvais pas.

N'oubliez pas également l'importance de la supervision des sauvegardes !

[10] https://github.com/pgmasters/backrest

3.8 TRAVAUX PRATIQUES

3.8.1 ÉNONCÉS

pg_basebackup

Configurer PostgreSQL pour réaliser une sauvegarde avec pg_basebackup. Sauvegarder avec pg_basebackup dans une archive en incluant les fichiers WAL.

Barman

a. Installer Barman à partir des paquets.

b. Configurer barman pour :

- Compresser les fichiers WAL archivés
- Activer le stockage incrémental des base backups

c. Configurer la sauvegarde du serveur PostgreSQL local,

- Utiliser le nom `instance-locale` dans la configuration de Barman
- Barman doit pouvoir se connecter à PostgreSQL avec un super-utilisateur PostgreSQL nommé `barman`
- Barman doit pouvoir récupérer les fichiers à partir d'une connexion SSH vers `localhost` utilisant des clé SSH sans passphrase

d. Configurer l'archivage des journaux de transactions de PostgreSQL

- Obtenir les informations du serveur dans Barman pour identifier le répertoire incoming pour les fichiers WAL
- Archiver les journaux de transactions avec `rsync` à travers SSH, le serveur Barman est sur la machine `localhost`. Mettre en place des clés SSH sans passphrase.

e. Vérifier la configuration de Barman et lancer une sauvegarde

f. Ajouter des données :

- Se connecter à la base `cave`, et compter le nombre de lignes dans la table `stock`.
- Voir le stock pour l'enregistrement: `vin_id = 1, contenant_id = 1, annee = 1950`.
- Effectuer une modification du stock (+5) pour l'enregistrement: `vin_id = 1, contenant_id = 1, annee = 1950`.
- Forcer la rotation du journal de transaction courant afin de s'assurer que les dernières modifications sont archivées.

g. Simulation d'un incident :

- Supprimer tout le contenu de la table `stock`

h. Restaurer les données avant l'incident à l'aide de Barman

- utiliser une restauration « à distance » pour envoyer les données avec une connexion ssh utilisant l'utilisateur `postgres`.

i. Restaurer les données avant l'incident de sorte que PostgreSQL obtienne les journaux de transactions archivés de Barman à travers le réseau.

pitrery

a. Installer pitrery à partir des paquets.

b. Configurer pitrery pour :

- Sauvegarder le serveur PostgreSQL en local dans `/var/lib/pgsql/10/backups`
- La sauvegarde doit être compressée
- Les journaux de transactions archivés doivent être compressés

c. Configurer l'archivage des journaux de transactions de PostgreSQL avec `archive_xlog`

d. Lancer une sauvegarde. Afficher les détails de cette sauvegarde

e. Ajouter des données :

- Se connecter à la base `cave`, et compter le nombre de lignes dans la table `stock`.
- Voir le stock pour l'enregistrement: `vin_id = 1, contenant_id = 1, annee = 1950`.
- Effectuer une modification du stock (+5) pour l'enregistrement: `vin_id = 1, contenant_id = 1, annee = 1950`.
- Forcer la rotation du journal de transaction courant afin de s'assurer que les dernières modifications sont archivées.

g. Simulation d'un incident :

- Supprimer tout le contenu de la table `stock`

h. Restaurer les données avant l'incident à l'aide de pitrery.

3.8.2 SOLUTIONS

pg_basebackup

Configurer la réplication dans `postgresql.conf`. L'archivage doit être actif, ici on choisit de ne pas archiver réellement et ne passer que par la réplication :

17.12

```
archive_mode = on
archive_command = '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 :

```
# service postgresql-10 restart
```

Lancer pg_basebackup :

```
postgres$ pg_basebackup -D /var/lib/pgsql/10/backups/basebackup \
                        -Ft -z -P -h /tmp
```

```
postgres$ cd /var/lib/pgsql/10/backups/basebackup
postgres$ ls -l
total 24348
-rw-r--r-- 1 postgres postgres 24894104 20 sept. 05:32 base.tar.gz
-rw------- 1 postgres postgres    29452 20 sept. 05:32 pg_wal.tar.gz
```

Barman

a. Installer barman :

```
root# yum install barman rsync
```

b. Editer `/etc/barman.conf` en root et modifier ces éléments :

```
[barman]
...
compression = gzip
reuse_backup = link
...
```

c. A la fin du fichier `/etc/barman.conf` ajouter une section pour le serveur local :

```
[instance-locale]
description =  "Instance PostgreSQL locale"
ssh_command = ssh postgres@localhost
conninfo = host=127.0.0.1 user=barman dbname=postgres
archiver = on
```

Ajouter les clés SSH :

- Coté serveur PostgreSQL, configurer le mot de passe de l'utilisateur système `postgres` :

```
root# passwd postgres
```

- Générer des clés SSH sans passphrase pour `postgres` :

```
root# sudo -iu postgres
postgres$ ssh-keygen
```

- Coté serveur Barman, configurer le mot de passe de l'utilisateur système `barman` :

```
root# passwd barman
```

- Générer des clés SSH sans passphrase pour `barman` :

```
root# sudo -iu barman
barman$ ssh-keygen
```

- Installer la clé publique de `postgres` sur le compte `barman` du serveur de backup. Coté serveur PostgreSQL :

```
root# sudo -iu postgres
postgres$ ssh-copy-id barman@localhost
The authenticity of host 'localhost (::1)' can't be established.
RSA key fingerprint is 72:7f:3f:89:91:c5:f7:00:cc:38:07:02:cd:0e:23:10.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (RSA) to the list of known hosts.
barman@localhost's password:
Now try logging into the machine, with "ssh 'barman@localhost'", and check in:

  .ssh/authorized_keys

to make sure we haven't added extra keys that you weren't expecting.
```

- Installer la clé publique de `barman` sur le compte `postgres` du serveur PostgreSQL : coté serveur Barman :

```
# sudo -iu barman
barman$ ssh-copy-id postgres@localhost
The authenticity of host 'localhost (::1)' can't be established.
RSA key fingerprint is 72:7f:3f:89:91:c5:f7:00:cc:38:07:02:cd:0e:23:10.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (RSA) to the list of known hosts.
postgres@localhost's password:
Now try logging into the machine, with "ssh 'postgres@localhost'", and check in:

  .ssh/authorized_keys
```

```
to make sure we haven't added extra keys that you weren't expecting.
```

Ajouter le super-utilisateur barman dans PostgreSQL :

```
postgres$ psql
postgres=# create role barman login superuser password 'Backup!3';
```

Ajouter une autorisation dans le fichier `$PGDATA/pg_hba.conf` :

```
# Après la dernière ligne commencant par "local"
host    all    barman     127.0.0.1/32    md5
```

Recharger la configuration de PostgreSQL :

```
postgres=# SELECT pg_reload_conf();
```

Sur le serveur Barman, ajouter le mot de passe dans un fichier `.pgpass` :

```
barman$ vi ~/.pgpass
```

```
127.0.0.1:5432:*:barman:Backup!3
```

```
barman$ chmod 0600 ~/.pgpass
```

Tester la connexion :

```
barman$ psql -h 127.0.0.1 -U barman postgres
```

Tester les paramètres avec Barman, il créera aussi les répertoires manquants :

```
barman$ barman check instance-locale
```

d. Barman gère les archives de WAL en deux temps. D'abord les serveurs PostgreSQL envoient dans leur répertoire `incoming` et puis le job cron de Barman les compresse le cas échéant et les déplace dans le répertoire final.

En listant les détails du serveur déclaré dans la configuration, on peut obtenir le chemin du répertoire `incoming` :

```
barman$ barman list-server
instance-locale - Instance PostgreSQL locale

barman$ barman show-server instance-locale
Server instance-locale:
    active: True
    archive_command: None
    archive_mode: None
    backup_directory: /var/lib/barman/instance-locale
    backup_options: BackupOptions(['exclusive_backup'])
    bandwidth_limit: None
```

```
basebackup_retry_sleep: 30
basebackup_retry_times: 0
basebackups_directory: /var/lib/barman/instance-locale/base
compression: gzip
conninfo: host=localhost user=barman
copy_method: rsync
current_xlog: None
custom_compression_filter: None
custom_decompression_filter: None
data_directory: None
description: Instance PostgreSQL locale
disabled: False
immediate_checkpoint: False
incoming_wals_directory: /var/lib/barman/instance-locale/incoming
last_backup_maximum_age: None
minimum_redundancy: 0
network_compression: False
pgespresso_installed: None
post_archive_retry_script: None
post_archive_script: None
post_backup_retry_script: None
post_backup_script: None
pre_archive_retry_script: None
pre_archive_script: None
pre_backup_retry_script: None
pre_backup_script: None
recovery_options: RecoveryOptions([])
retention_policy: None
retention_policy_mode: auto
reuse_backup: link
server_txt_version: None
ssh_command: ssh postgres@localhost
tablespace_bandwidth_limit: None
wal_level: None
wal_retention_policy: main
wals_directory: /var/lib/barman/instance-locale/wals
```

Il s'agit de la variable `incoming_wals_directory`, le répertoire est donc :

```
/var/lib/barman/instance-locale/incoming
```

17.12

Configurer l'archivage des journaux de transactions dans `postgresql.conf` sur le serveur PostgreSQL :

```
wal_level = replica
archive_mode = on
archive_command = 'rsync -a %p \
                     barman@localhost:/var/lib/barman/instance-locale/incoming/%f'
archive_timeout = 300
```

Redémarrer l'instance PostgreSQL (la méthode de redémarrage peut dépendre du type d'installation réalisée) :

```
# exemple avec pg_ctl et l'utilisateur système postgres :
$ pg_ctl -D $PGDATA restart

# exemple avec le script init.d sous Debian et l'utilisateur système root
# (redémarre toutes les instances s'il y en a plus d'une) :
$ service postgresql restart

# exemple avec le script init.d sous CentOS 6 et l'utilisateur système root :
$ service postgresql-10 restart

# exemple avec le script systemd sous CentOS 7 et l'utilisateur système root :
$ systemctl restart postgresql-10
```

Vérifier le bon fonctionnement de l'archivage :

```
# Forcer la création d un nouveau journal:
postgres$ psql
postgres=# CHECKPOINT;
postgres=# select pg_switch_wal();

-- Interroger la vue pg_stat_archiver:
postgres=# SELECT * FROM pg_stat_archiver;

# Lister le contenu du répertoire backup/instance-locale/incoming/
# avec l utilisateur barman:
barman$ ls /var/lib/barman/instance-locale/incoming
```

e. Lancer manuellement une nouvelle sauvegarde avec la commande barman `backup`:

```
barman$ barman backup instance-locale
Starting backup for server instance-locale in
        /var/lib/barman/instance-locale/base/20170913T142542
Backup start at xlog location: 0/20000028 (000000010000000000000020, 00000028)
Copying files.
Copy done.
```

```
Asking PostgreSQL server to finalize the backup.
Backup size: 118.2 MiB. Actual size on disk: 118.2 MiB
             (-0.00% deduplication ratio).
Backup end at xlog location: 0/200000B8 (000000010000000000000020, 000000B8)
Backup completed
Processing xlog segments for instance-locale
    000000010000000000000020
    000000010000000000000020.00000028.backup
```

f. Se connecter à la base `cave`, et compter le nombre de lignes dans la table `stock` :

```
cave=# SELECT COUNT(*) FROM stock;
 count
--------
 860139
(1 row)
```

Voir le stock pour l'enregistrement: `vin_id = 1`, `contenant_id = 1`, `annee = 1950` :

```
cave=# SELECT * FROM stock WHERE vin_id = 1 AND contenant_id = 1
AND annee = 1950;
 vin_id | contenant_id | annee | nombre
--------+--------------+-------+--------
      1 |            1 |  1950 |      7
(1 row)
```

Effectuer une modification du stock (+5) pour l'enregistrement: `vin_id = 1`, `contenant_id = 1`, `annee = 1950` :

```
cave=# UPDATE stock SET nombre = nombre + 5 WHERE vin_id = 1
AND contenant_id = 1 AND annee = 1950;
```

Forcer la rotation du journal de transaction courant afin de s'assurer que les dernières modifications sont archivées :

```
cave=# SELECT pg_switch_wal();
```

g. Création d'un incident : supprimer tout le contenu de la table stock et récupérer le numéro de la transaction courante :

```
cave=# BEGIN;
BEGIN
cave=# SELECT txid_current();
 txid_current
--------------
         2350
(1 row)
```

```
cave=# TRUNCATE stock;
TRUNCATE TABLE

cave=# COMMIT;
COMMIT

cave=# SELECT COUNT(*) FROM stock ;
 count
-------
     0
(1 row)
```

h. Restauration des données :

Arrêter l'instance PostgreSQL (la méthode d'arrêt peut dépendre de la méthode d'installation utilisée) :

```
# exemple avec pg_ctl et l'utilisateur système postgres :
postgres$ pg_ctl -D $PGDATA stop

# exemple avec le script init.d sous Debian et l'utilisateur système root
# (arrête toutes les instances s'il y en a plus d'une) :
postgres$ service postgresql stop

# exemple avec le script init.d sous CentOS 6 et l'utilisateur système root :
postgres$ service postgresql-10 stop

# exemple avec le script systemd sous CentOS 7 et l'utilisateur système root :
postgres$ systemctl stop postgresql-10
```

Lister les sauvegardes à disposition via la commande barman `list-backup` et récupérer l'identifiant de la dernière sauvegarde, c'est à partir de celle-ci que l'on va restaurer les données :

```
barman$ barman list-backup instance-locale
instance-locale 20170913T142542 - Wed Sep 13 14:26:03 2017 - Size: 118.2 MiB -
                                  WAL Size: 26.4 KiB
```

Restaurer à partir de la dernière sauvegarde jusqu'à la transaction identifiée, en l'excluant et en envoyant les données à travers SSH:

```
barman$ barman recover --target-xid 2350 --exclusive --remote-ssh-command \
```

```
                       'ssh postgres@localhost' instance-locale \
                       20170913T142542 /var/lib/pgsql/10/data
Starting remote restore for server instance-locale using backup 20170913T142542
Destination directory: /var/lib/pgsql/10/data
Doing PITR. Recovery target xid: '2350'
Copying the base backup.
Copying required WAL segments.
Generating recovery.conf
Identify dangerous settings in destination directory.

IMPORTANT
These settings have been modified to prevent data losses

postgresql.conf line 207: archive_command = false

Your PostgreSQL server has been successfully prepared for recovery!
```

Démarrer l'instance PostgreSQL:

```
# exemple avec pg_ctl et l'utilisateur système postgres :
$ pg_ctl -D $PGDATA start

# exemple avec le script init.d sous Debian et l'utilisateur système root
# (démarre toutes les instances s'il y en a plus d'une) :
$ service postgresql start

# exemple avec le script init.d sous CentOS 6 et l'utilisateur système root :
$ service postgresql-10 start

# exemple avec le script systemd sous CentOS 7 et l'utilisateur système root :
$ systemctl start postgresql-10
```

Vérifier l'état de la table `stock`:

```
postgres$ psql cave
cave=# SELECT COUNT(*) FROM stock ;
 count
--------
 860139
(1 row)

cave=# SELECT * FROM stock WHERE vin_id = 1 AND contenant_id = 1
AND annee = 1950;
```

17.12

```
 vin_id | contenant_id | annee | nombre
--------+--------------+-------+--------
      1 |            1 |  1950 |     12
(1 row)
```

pitrery

a. Pour installer pitrery à partir des paquets, aller sur le site http://dalibo.github.io/pitrery/downloads.html et télécharger le paquet RPM, puis l'installer :

```
root# wget --no-check-certificate \
      https://dl.dalibo.com/public/pitrery/rpms/pitrery-2.0-1.el7.centos.noarch.rpm
root# rpm -ivh pitrery-2.0-1.el7.centos.noarch.rpm
```

b. Configurer pitrery, pour cela éditer le fichier `/etc/pitrery/pitr.conf` :

```
PGDATA="/var/lib/pgsql/10/data"
PGPSQL="psql"
PGUSER="postgres"
PGPORT=5432
PGHOST="/tmp"
PGDATABASE="postgres"

# Le backup ce fait en local
BACKUP_DIR="/var/lib/pgsql/10/backups"

# Les backups seront stockés sous forme d'archive tar compressées
STORAGE="tar"

# Les archives seront stockées en local par le script archive_xlog
ARCHIVE_COMPRESS="yes"
```

c. Configurer l'archivage des journaux de transaction dans `$PGDATA/postgresql.conf` :

```
wal_level = replica
archive_mode = on
archive_command = 'archive_xlog -C pitr %p'
archive_timeout = 300
```

Redémarrer PostgreSQL :

```
postgres$ pg_ctl restart
```

Vérifier que l'archivage fonctionne :

```
postgres$ ls /var/lib/pgsql/10/backups/pitr/archived_wal/
```

```
postgres$ psql
postgres=# select * from pg_stat_archiver;
```

d. Lancer une sauvegarde, le nom du fichier de configuration `pitr` est facultatif, c'est la valeur par défaut :

```
postgres$ pitrery backup
INFO: preparing directories in /var/lib/pgsql/10/backups/pitr
INFO: listing tablespaces
INFO: starting the backup process
INFO: backing up PGDATA with tar
INFO: archiving /var/lib/pgsql/10/data
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 /var/lib/pgsql/10/backups/pitr/2017.09.13_15.17.20
INFO: done
```

Lister les sauvegardes avec les détails :

```
postgres$ pitrery list -v
List of local backups
----------------------------------------------------------------------
Directory:
  /var/lib/pgsql/10/backups/pitr/2017.09.13_15.17.20
  space used: 20M
  storage: tar with gz compression
Minimum recovery target time:
  2017-09-13 15:17:20 CET
PGDATA:
  pg_default 117 MB
  pg_global 437 kB
Tablespaces:
```

e. Ajouter des données : se connecter à la base `cave`, et compter le nombre de lignes dans la table `stock` :

```
cave=# SELECT COUNT(*) FROM stock;
 count
--------
 860139
(1 row)
```

Voir le stock pour l'enregistrement : `vin_id = 1`, `contenant_id = 1`, `annee = 1950` :

```
cave=# SELECT * FROM stock WHERE vin_id = 1 AND contenant_id = 1
AND annee = 1950;
 vin_id | contenant_id | annee | nombre
--------+--------------+-------+--------
      1 |            1 |  1950 |      7
(1 row)
```

Effectuer une modification du stock (+5) pour l'enregistrement: `vin_id = 1`, `contenant_id = 1`, `annee = 1950` :

```
cave=# UPDATE stock SET nombre = nombre + 5 WHERE vin_id = 1
AND contenant_id = 1 AND annee = 1950;
```

Forcer la rotation du journal de transaction courant afin de s'assurer que les dernières modifications sont archivées :

```
cave=# SELECT pg_switch_wal();
```

g. Pour simuler un incident, on vide la table stock, en ayant récupéré la date au préalable :

```
cave=# BEGIN;
BEGIN
cave=# select now();
              now
-------------------------------
 2017-09-13 15:20:49.169791+01
(1 row)

cave=# truncate stock;
TRUNCATE TABLE
cave=# commit;
COMMIT
```

h. Restaurer les données à la date obtenu précédemment. Il faut d'abord stopper PostgreSQL.

```
postgres$ pg_ctl stop
```

Lancer la commande de restauration :

```
postgres$ pitrery restore -R -d '2017-09-13 15:20:49' -D /var/lib/pgsql/10/data
INFO: searching backup directory
INFO: searching for tablespaces information
INFO:
INFO: backup directory:
INFO:   /var/lib/pgsql/10/backups/pitr/2017.09.13_15.17.20
```

```
INFO:
INFO: destinations directories:
INFO:   PGDATA -> /var/lib/pgsql/10/data
INFO:
INFO: recovery configuration:
INFO:   target owner of the restored files: postgres
INFO:   restore_command = '/usr/bin/restore_xlog  %f %p'
INFO:   recovery_target_time = '2017-09-13 15:20:49'
INFO:
INFO: checking if /var/lib/pgsql/10/data is empty
INFO: /var/lib/pgsql/10/data is not empty, its contents will be overwritten
INFO: Removing contents of /var/lib/pgsql/10/data
INFO: extracting PGDATA to /var/lib/pgsql/10/data
INFO: extraction of PGDATA successful
INFO: preparing pg_xlog 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:
```

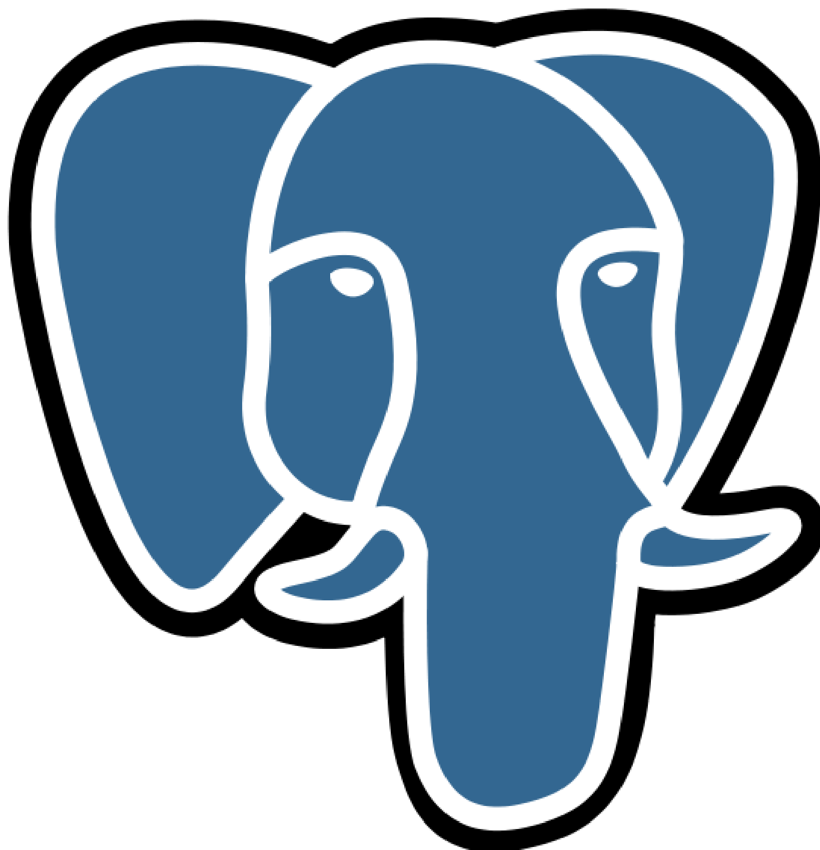
Démarrer PostgreSQL :

```
pg_ctl start
```

Vérifier les logs et la base cave :

```
cave=# SELECT * FROM stock WHERE vin_id = 1 AND contenant_id = 1
AND annee = 1950;
 vin_id | contenant_id | annee | nombre
--------+--------------+-------+--------
      1 |            1 |  1950 |     12
(1 row)
```

4 POSTGRESQL : GESTION D'UN SINISTRE



4.1 INTRODUCTION

- Une bonne politique de sauvegardes est cruciale
  - mais elle n'empêche pas les incidents
- Il faut être prêt à y faire face

Ce module se propose de faire une description des bonnes et mauvaises pratiques en cas de coup dur :

- crash de l'instance ;
- suppression / corruption de fichiers ;
- problèmes matériels ;
- sauvegardes corrompues…

Seront également présentées les situations classiques de désastres, ainsi que certaines méthodes et outils dangereux et déconseillés.

L'objectif est d'aider à convaincre de l'intérêt qu'il y a à anticiper les problèmes, à mettre en place une politique de sauvegarde pérenne, et à ne pas tenter de manipulation dangereuse sans comprendre précisément à quoi l'on s'expose.

Ce module est en grande partie inspiré de la présentation de Christophe Pettus[11]

4.1.1 AU MENU

- Anticiper les désastres
- Réagir aux désastres
- Rechercher l'origine du problème
- Outils utiles
- Cas type de désastres

4.2 ANTICIPER LES DÉSASTRES

- Un désastre peut toujours survenir
- Il faut savoir le détecter le plus tôt possible
  - et s'être préparé à y répondre

Il est impossible de parer à tous les cas de désastres imaginables.

Le matériel peut subir des pannes, une faille logicielle non connue peut être exploitée, une modification d'infrastructure ou de configuration peut avoir des conséquences imprévues à long terme, une erreur humaine est toujours possible.

Les principes de base de la haute disponibilité (redondance, surveillance…) permettent de mitiger le problème, mais jamais de l'éliminer complètement.

[11] http://thebuild.com/presentations/worst-day-fosdem-2014.pdf

Il est donc extrêmement important de se préparer au mieux, de procéder à des simulations, de remettre en question chaque brique de l'infrastructure pour être capable de détecter une défaillance et d'y réagir rapidement.

4.2.1 DOCUMENTATION

- Documentation complète et à jour
  - emplacement et fréquence des sauvegardes
  - emplacement des traces
  - procédures et scripts d'exploitation
- Sauvegarder et versionner la documentation

Par nature, les désastres arrivent de façon inattendue.

Il faut donc se préparer à devoir agir en urgence, sans préparation, dans un environnement perturbé et stressant - par exemple, en pleine nuit, la veille d'un jour particulièrement critique pour l'activité de la production.

Un des premiers points d'importance est donc de s'assurer de la présence d'une documentation claire, précise et à jour, afin de minimiser le risque d'erreurs humaines.

Cette documentation devrait détailler l'architecture dans son ensemble, et particulièrement la politique de sauvegarde choisie, l'emplacement de celles-ci, les procédures de restauration et éventuellement de bascule vers un environnement de secours.

Les procédures d'exploitation doivent y être expliquées, de façon détaillée mais claire, afin qu'il n'y ait pas de doute sur les actions à effectuer une fois la cause du problème identifié.

La méthode d'accès aux informations utiles (traces de l'instance, du système, supervision…) devrait également être soigneusement documentée afin que le diagnostic du problème soit aussi simple que possible.

Toutes ces informations doivent être organisées de façon claire, afin qu'elles soient immédiatement accessibles et exploitables aux intervenants lors d'un problème.

Il est évidemment tout aussi important de penser à versionner et sauvegarder cette documentation, afin que celle-ci soit toujours accessible même en cas de désastre majeur (perte d'un site).

4.2.2 PROCÉDURES ET SCRIPTS

- Procédures détaillées de restauration / PRA
  - préparer des scripts / utiliser des outils
  - minimiser le nombre d'actions manuelles
- Tester les procédures régulièrement
  - s'assurer que chacun les maîtrise
- Sauvegarder et versionner les scripts

La gestion d'un désastre est une situation particulièrement stressante, le risque d'erreur humaine est donc accru.

Un DBA devant restaurer d'urgence l'instance de production en pleine nuit courera plus de risques de faire une fausse manipulation s'il doit taper une vingtaine de commandes en suivant une procédure dans une autre fenêtre (voire un autre poste) que s'il n'a qu'un script à exécuter.

En conséquence, il est important de minimiser le nombre d'actions manuelles à effectuer dans les procédures, en privilégiant l'usage de scripts d'exploitation ou d'outils dédiés (comme pitrery ou barman pour restaurer une instance PostgreSQL).

Néanmoins, même cette pratique ne suffit pas à exclure tout risque.

L'utilisation de ces scripts ou de ces outils doit également être correctement documentée, et les procédures régulièrement testées.

Dans le cas contraire, l'utilisation d'un script ou d'un outil peut aggraver le problème, parfois de façon dramatique - par exemple, l'écrasement d'un environnement sain lors d'une restauration parce que la procédure ne mentionne pas que le script doit être lancé depuis un serveur particulier.

L'aspect le plus important est de s'assurer par des tests réguliers **et manuels** que les procédures sont à jour, n'ont pas de comportement inattendu, et sont maîtrisées par toute l'équipe d'exploitation.

Tout comme pour la documentation, les scripts d'exploitation doivent également être sauvegardés et versionnés.

4.2.3 SUPERVISION ET HISTORISATION

- Tout doit être supervisé
  - réseau, matériel, système, logiciels…
  - les niveaux d'alerte doivent être significatifs

- Les métriques importantes doivent être historisées
  - cela permet de retrouver le moment où le problème est apparu
  - quand cela a un sens, faire des graphes

La supervision est un sujet vaste, qui touche plus au domaine de la haute disponibilité.

Un désastre sera d'autant plus difficile à gérer qu'il est détecté tard. La supervision en place doit donc être pensée pour détecter tout type de défaillance (penser également à superviser la supervision !).

Attention à bien calibrer les niveaux d'alerte, la présence de trop de messages augmente le risque que l'un d'eux passe inaperçu, et donc que l'incident ne soit détecté que tardivement.

Pour aider la phase de diagnostic de l'origine du problème, il faut prévoir d'historiser un maximum d'informations.

La présentation de celles-ci est également importante : il est plus facile de distinguer un pic brutal du nombre de connexions sur un graphique que dans un fichier de traces de plusieurs Go !

4.2.4 AUTOMATISATION

- Des outils existent
  - Pacemaker, repmgr…
- Automatiser une bascule est complexe
- Cela peut mener à davantage d'incidents
  - voire à des désastres importants (*split brain*)

Si on poursuit jusqu'au bout le raisonnement précédent sur le risque à faire effectuer de nombreuses opérations manuelles lors d'un incident, la conclusion logique est que la solution idéale serait de les éliminer complètement, et d'automatiser complètement le déclenchement et l'exécution de la procédure.

Le problème est que toute solution visant à automatiser une tâche se base sur un nombre limité de paramètres et sur une vision restreinte de l'architecture.

Cette vision peut être erronée : typiquement, il est difficile à un outil de bascule automatique de diagnostiquer correctement certains types d'incident, par exemple une partition réseau.

L'outil peut donc détecter à tort à un incident, surtout s'il est réglé de façon à être assez sensible, et ainsi provoquer lui-même une coupure de service inutile.

Dans le pire des cas, l'outil peut être amené à prendre une mauvaise décision amenant à une situation de désastre, comme un *split brain* (deux instances PostgreSQL se retrouvent ouvertes en écriture en même temps sur les mêmes données).

Il est donc fortement préférable de laisser un administrateur prendre les décisions potentiellement dangereuses, comme une bascule ou une restauration.

4.3 RÉAGIR AUX DÉSASTRES

- Savoir identifier un problème majeur
- Bons réflexes
- Mauvais réflexes

En dépit de toutes les précautions que l'on peut être amené à prendre, rien ne peut garantir qu'aucun problème ne surviendra.

Il faut donc être capable d'identifier le problème lorsqu'il survient, et être prêt à y répondre.

4.3.1 SYMPTÔMES D'UN DÉSASTRE

- Crash de l'instance
- Résultats de requêtes erronnés
- Messages d'erreurs dans les traces
- Dégradation importante des temps d'exécution
- Processus manquants
  - ou en court d'exécution depuis trop longtemps

De très nombreux éléments peuvent aider à identifier que l'on est en situation d'incident grave.

Le plus flagrant est évidemment le crash complet de l'instance PostgreSQL, ou du serveur l'hébergeant, et l'impossibilité pour PostgreSQL de redémarrer.

Les désastres les plus importants ne sont toutefois pas toujours aussi simples à détecter.

Les crash peuvent se produire uniquement de façon ponctuelle, et il existe des cas où l'instance redémarre immédiatement après (typiquement suite au `kill -9` d'un processus backend PostgreSQL).

Cas encore plus délicat, il peut également arriver que les résultats de requêtes soient erronés (par exemple en cas de corruption de fichiers d'index) sans qu'aucune erreur n'apparaisse.

Les symptômes classiques permettant de détecter un problème majeur sont :

- la présence de messages d'erreurs dans les traces de PostgreSQL (notamment des messages `PANIC` ou `FATAL`, mais les messages `ERROR` et `WARNING` sont également très significatifs, particulièrement s'ils apparaissent soudainement en très grand nombre) ;
- la présence de messages d'erreurs dans les traces du système d'exploitation (notamment concernant la mémoire ou le système de stockage) ;
- le constat d'une dégradation importante des temps d'exécution des requêtes sur l'instance ;
- l'absence de certains processus critiques de PostgreSQL ;
- la présence de processus présents depuis une durée inhabituelle (plusieurs semaines, mois, …).

4.3.2 BONS RÉFLEXES 1

- Garder la tête froide
- Répartir les tâches clairement
- Minimiser les canaux de communication
- Garder des notes de chaque action entreprise

Une fois que l'incident est repéré, il est important de ne pas foncer tête baissée dans des manipulations.

Il faut bien sûr prendre en considération la criticité du problème, notamment pour définir la priorité des actions (par exemple, en cas de perte totale d'un site, quels sont les applications à basculer en priorité ?), mais quelle que soit la criticité ou l'impact, il ne faut jamais effectuer une action sans en avoir parfaitement saisi l'impact et s'être assuré qu'elle répondait bien au problème rencontré.

Si le travail s'effectue en équipe, il faut bien faire attention à répartir les tâches clairement, afin d'éviter des manipulations concurrentes ou des oublis qui pourraient aggraver la situation.

Il faut également éviter de multiplier les canaux de communication, cela risque de favoriser la perte d'information, ce qui est critique dans une situation de crise.

Surtout, une règle majeure est de prendre le temps de noter systématiquement toutes les actions entreprises.

Les commandes passées, les options utilisées, l'heure d'exécution, toutes ces informations sont très importantes, déjà pour pouvoir agir efficacement en cas de fausse manipulation, mais également pour documenter la gestion de l'incident après coup, et ainsi en conserver une trace qui sera précieuse si celui-ci venait à se reproduire.

4.3.3 BONS RÉFLEXES 2

- Se prémunir contre une aggravation du problème
  - couper les accès applicatifs
- Si une corruption est suspectée
  - arrêter immédiatement l'instance
  - faire une sauvegarde immédiate des fichiers
  - travailler sur une copie

S'il y a suspicion de potentielle corruption de données, il est primordial de s'assurer au plus vite de couper tous les accès applicatifs vers l'instance afin de ne pas aggraver la situation.

Il est généralement préférable d'avoir une coupure de service plutôt qu'un grand volume de données irrécupérables.

Ensuite, il faut impérativement faire une sauvegarde complète de l'instance avant de procéder à toute manipulation. En fonction de la nature du problème rencontré, le type de sauvegarde pouvant être effectué peut varier (un export de données ne sera possible que si l'instance est démarrée et que les fichiers sont lisibles par exemple). En cas de doute, la sauvegarde la plus fiable qu'il est possible d'effectuer est une copie des fichiers à froid (instance arrêtée) - toute autre action (y compris un export de données) pourrait avoir des conséquences indésirables.

Si des manipulations doivent être tentées pour tenter de récupérer des données, il faut impérativement travailler sur une copie de l'instance, restaurée à partir de cette sauvegarde. Ne jamais travailler directement sur une instance de production corrompue, la moindre action (même en lecture) pourrait aggraver le problème !

Pour plus d'information, voir sur le wiki PostgreSQL[12] .

---

[12] https://wiki.postgresql.org/wiki/Corruption

17.12

4.3.4 BONS RÉFLEXES 3

- Déterminer le moment de démarrage du désastre
- Adopter une vision générale plutôt que focalisée sur un détail
- Remettre en cause chaque élément de l'architecture
  - aussi stable (et/ou coûteux/complexe) soit-il
- Éliminer en priorité les causes possibles côté hardware, système
- Isoler le comportement précis du problème
  - identifier les requêtes / tables / index impliqués

La première chose à identifier est l'instant précis où le problème a commencé à se manifester. Cette information est en effet déterminante pour identifier la cause du problème, et le résoudre - notamment pour savoir à quel instant il faut restaurer l'instance si cela est nécessaire.

Il convient pour cela d'utiliser les outils de supervision et de traces (système, applicatif et PostgreSQL) pour remonter au moment d'apparition des premiers symptômes. Attention toutefois à ne pas confondre les symptômes avec le problème lui-même ! Les symptômes les plus visibles ne sont pas forcément apparus les premiers. Par exemple, la charge sur la machine est un symptôme, mais n'est jamais la cause du problème. Elle est liée à d'autres phénomènes, comme des problèmes avec les disques ou un grand nombre de connexions, qui peuvent avoir commencé à se manifester bien avant que la charge ne commence réellement à augmenter.

Si la nature du problème n'est pas évidente à ce stade, il faut examiner l'ensemble de l'architecture en cause, sans en exclure d'office certains composants (baie de stockage, progiciel…), quels que soient leur complexité / coût / stabilité supposés. Si le comportement observé côté PostgreSQL est difficile à expliquer (crashs plus ou moins aléatoires, nombreux messages d'erreur sans lien apparent…), il est préférable de commencer par s'assurer qu'il n'y a pas un problème de plus grande ampleur (système de stockage, virtualisation, réseau, système d'exploitation).

Un bon indicateur consiste à regarder si d'autres instances / applications / processus rencontrent des problèmes similaires.

Ensuite, une fois que l'ampleur du problème a été cernée, il faut procéder méthodiquement pour en déterminer la cause et les éléments affectés.

Pour cela, les informations les plus utiles se trouvent dans les traces, généralement de PostgreSQL ou du système, qui vont permettre d'identifier précisément les éventuels fichiers ou relations corrompus.

4.3.5 BONS RÉFLEXES 4

- En cas de défaillance matérielle, s'assurer de travailler sur du hardware sain et non affecté !!!

Cette recommandation peut paraître aller de soi, mais si les problèmes sont provoqués par une défaillance matérielle, il est impératif de s'assurer que le travail de correction soit effectué sur un environnement non affecté.

Cela peut s'avérer problématique dans le cadre d'architecture mutualisant les ressources, comme des environnements virtualisés ou utilisant une baie de stockage.

Prendre également la précaution de vérifier que l'intégrité des sauvegardes n'est pas affectée par le problème.

4.3.6 BONS RÉFLEXES 5

- Communiquer, ne pas rester isolé
- Demander de l'aide si le problème est trop complexe
  - autres équipes
  - support
  - forums
  - listes

La communication est très importante dans la gestion d'un désastre.

Il est préférable de minimiser le nombre de canaux de communication plutôt que de les multiplier (téléphone, e-mail, chat, ticket…), ce qui pourrait amener à une perte d'informations et à des délais indésirables.

Il est primordial de rapidement cerner l'ampleur du problème, et pour cela il est généralement nécessaire de demander l'expertise d'autres administrateurs / équipes (applicatif, système, réseau, virtualisation, SAN…). Il ne faut pas rester isolé et risquer que la vision étroite que l'on a des symptômes (notamment en terme de supervision / accès aux traces) empêche l'identification de la nature réelle du problème.

Si la situation semble échapper à tout contrôle, et dépasser les compétences de l'équipe en cours d'intervention, il faut chercher de l'aide auprès de personnes compétentes, par exemple auprès d'autres équipes, du support.

En aucun cas, il ne faut se mettre à suivre des recommandations glanées sur Internet, qui ne se rapporteraient que très approximativement au problème rencontré, voire pas

du tout. Si nécessaire, on trouve en ligne des forums et des listes de discussions spécialisées sur lesquels il est également possible d'obtenir des conseils - il est néanmoins indispensable de prendre en compte que les personnes intervenant sur ces médias le font de manière bénévole. Il est déraisonnable de s'attendre à une réaction immédiate, aussi urgent le problème soit-il, et les suggestions effectuées le sont sans aucune garantie.

4.3.7 BONS RÉFLEXES 6

- Dérouler les procédures comme prévu
- En cas de situation non prévue, s'arrêter pour faire le point
  - ne pas hésiter à remettre en cause l'analyse
  - ou la procédure elle-même

Dans l'idéal, des procédures détaillant les actions à effectuer ont été écrites pour le cas de figure rencontré. Dans ce cas, une fois que l'on s'est assuré d'avoir identifié la procédure appropriée, il faut la dérouler méthodiquement, point par point, et valider à chaque étape que tout se déroule comme prévu.

Si une étape de la procédure ne se passe pas comme prévu, il ne faut pas tenter de poursuivre tout de même son exécution sans avoir compris ce qui s'est passé et les conséquences. Cela pourrait être dangereux.

Il faut au contraire prendre le temps de comprendre le problème en procédant comme décrit précédemment, quitte à remettre en cause toute l'analyse menée auparavant, et la procédure ou les scripts utilisés.

C'est également pour parer à ce type de cas de figure qu'il est important de travailler sur une copie et non sur l'environnement de production directement.

4.3.8 BONS RÉFLEXES 7

- En cas de bug avéré
  - tenter de le cerner et de le reproduire au mieux
  - le signaler à la communauté de préférence en détaillant le moyen de le reproduire

Ce n'est heureusement pas fréquent, mais il est possible que l'origine du problème soit liée à un bug de PostgreSQL lui-même.

Dans ce cas, la méthodologie appropriée consiste à essayer de reproduire le problème le plus fidèlement possible et de façon systématique, pour le cerner au mieux.

Il est ensuite très important de le signaler au plus vite à la communauté, généralement sur la liste pgsql-bugs@postgresql.org (cela nécessite une inscription préalable), en respectant les règles définies dans la documentation[13] .

Notamment (liste non exhaustive) :

- indiquer la version précise de PostgreSQL installée, et la méthode d'installation utilisée ;
- préciser la plate-forme utilisée, notamment la version du système d'exploitation utilisé et la configuration des ressources du serveur ;
- signaler uniquement les faits observés, éviter les spéculations sur l'origine du problème ;
- joindre le détail des messages d'erreurs observés (augmenter la verbosité des erreurs avec le paramètre `log_error_verbosity`) ;
- joindre un cas complet permettant de reproduire le problème de façon aussi simple que possible.

Pour les problèmes relevant du domaine de la sécurité (découverte d'une faille), la liste adéquate est security@postgresql.org.

4.3.9 BONS RÉFLEXES 8

- Tester complètement l'intégrité des données
  - pour détecter tous les problèmes
  - pour valider après restauration / correction

Un fois les actions correctives réalisées (restauration, recréation d'objets, mise à jour des données…), il faut tester intensivement pour s'assurer que le problème est bien complètement résolu.

Il est donc extrêmement important d'avoir préparé des cas de tests permettant de reproduire le problème de façon certaine, afin de valider la solution appliquée.

En cas de corruption de données, il est également important de tenter de procéder à la lecture de la totalité des données depuis PostgreSQL.

La commande suivante, exécutée avec l'utilisateur système propriétaire de l'instance (généralement `postgres`) effectue une lecture complète de toutes les relations :

[13] http://www.postgresql.org/docs/current/static/bug-reporting.html

17.12

```
pg_dumpall > /dev/null
```

Cette commande ne devrait renvoyer aucune erreur.

Même si la lecture des données ne renvoie aucune erreur, il est toujours possible que des problèmes subsistent, par exemple des corruptions silencieuses, des index incohérents avec les données…

Dans les situations les plus extrêmes (problème de stockage, fichiers corrompus), il est important de tester la validité des données dans une nouvelle instance en effectuant un export/import complet des données.

Par exemple, initialiser une nouvelle instance avec `initdb`, sur un autre système de stockage, voire sur un autre serveur, puis lancer la commande suivante (l'application doit être coupée, ce qui est normalement le cas depuis la détection de l'incident si les conseils précédents ont été suivis) pour exporter et importer à la volée :

```
pg_dumpall -h <serveur_corrompu> -U postgres | psql -h <nouveau_serveur> \
                                                   -U postgres postgres
vacuumdb -z -h <nouveau_serveur> -U postgres postgres
```

D'éventuels problèmes peuvent être détectés lors de l'import des données. Il faut alors procéder au cas par cas.

Enfin, même si cette étape s'est déroulée sans erreur, tout risque n'est pas écarté, il reste la possibilité de corruption de données silencieuses. Sauf si la fonctionnalité de checksum de PostgreSQL a été activée sur l'instance, le seul moyen de détecter ce type de problème est de valider les données fonctionnellement.

Dans tous les cas, en cas de suspicion de corruption de données en profondeur, il est fortement préférable d'accepter une perte de données et de restaurer avant le début de l'incident, plutôt que de continuer à travailler avec des données dont l'intégrité n'est pas assurée.

4.3.10 MAUVAIS RÉFLEXES 1

- Paniquer
- Prendre une décision hâtive
  - exemple, supprimer des fichiers du répertoire `pg_wal`
- Lancer une commande sans la comprendre
  - exemple, `pg_resetwal`

Quelle que soit la criticité du problème rencontré, la panique peut en faire quelque chose de pire.

Il faut impérativement garder son calme, et résister au mieux au stress et aux pressions qu'une situation de désastre ne manque pas de provoquer.

Il est également préférable d'éviter de sauter immédiatement à la conclusion la plus évidente. Il ne faut pas hésiter à retirer les mains du clavier pour prendre de la distance par rapport aux conséquences du problème, réfléchir aux causes possibles, prendre le temps d'aller chercher de l'information pour réévaluer l'ampleur réelle du problème.

La plus mauvaise décision que l'on peut être amenée à prendre lors de la gestion d'un incident est celle que l'on prend dans la précipitation, sans avoir bien réfléchi et mesuré son impact. Cela peut provoquer des dégâts irrécupérables, et transformer une situation d'incident en situation de crise majeure.

Un exemple classique de ce type de comportement est le cas où PostgreSQL est arrêté suite au remplissage du système de fichiers contenant les fichiers WAL, `pg_wal`.

Le réflexe immédiat d'un administrateur non averti pourrait être de supprimer les plus vieux fichiers dans ce répertoire, ce qui répond bien aux symptômes observés mais reste une erreur dramatique qui va rendre le démarrage de l'instance impossible.

Quoi qu'il arrive, ne jamais exécuter une commande sans être certain qu'elle correspond bien à la situation rencontrée, et sans en maîtriser complètement les impacts. Même si cette commande provient d'un document mentionnant les mêmes messages d'erreur que ceux rencontrés (et tout particulièrement si le document a été trouvé via une recherche hâtive sur Internet) !

Là encore, nous disposons comme exemple d'une erreur malheureusement fréquente, l'exécution de la commande `pg_resetwal` sur une instance rencontrant un problème. Comme l'indique la documentation, « *[cette commande] ne doit être utilisée qu'en dernier ressort quand le serveur ne démarre plus du fait d'une telle corruption* » et « *il ne faut pas perdre de vue que la base de données peut contenir des données incohérentes du fait de transactions partiellement validées* » (documentation[14]). Nous reviendrons ultérieurement sur les (rares) cas d'usage réels de cette commande, mais dans l'immense majorité des cas, l'utiliser va aggraver le problème, en ajoutant des problématiques de corruption logique des données !

Il convient donc de bien s'assurer de comprendre les conséquence de l'exécution de chaque action effectuée.

---

[14] http://docs.postgresqlfr.org/current/app-pgresetwal.html

4.3.11 MAUVAIS RÉFLEXES 2

- Arrêter le diagnostic quand les symptômes disparaissent
- Ne pas pousser l'analyse jusqu'au bout

Il est important de pousser la réflexion jusqu'à avoir complètement compris l'origine du problème et ses conséquences.

En premier lieu, même si les symptômes semblent avoir disparus, il est tout à fait possible que le problème soit toujours sous-jacent, ou qu'il ait eu des conséquences moins visibles mais tout aussi graves (par exemple, une corruption logique de données).

Ensuite, même si le problème est effectivement corrigé, prendre le temps de comprendre et de documenter l'origine du problème a une valeur inestimable pour prendre les mesures afin d'éviter que le problème ne se reproduise, et retrouver rapidement les informations utiles s'il venait à se reproduire malgré tout.

4.3.12 MAUVAIS RÉFLEXES 3

- Ne pas documenter
  - le résultat de l'investigation
  - les actions effectuées

Après s'être assuré d'avoir bien compris le problème recontré, il est tout aussi important de le documenter soigneusement, avec les actions de diagnostic et de correction effectuées.

Ne pas le faire, c'est perdre un excellente occasion de gagner un temps précieux si le problème venait à se reproduire.

C'est également un risque supplémentaire dans le cas où les actions correctives menées n'auraient pas suffit à complètement corriger le problème ou auraient eu un effet de bord inattendu.

Dans ce cas, avoir pris le temps de noter le détail des actions effectuées fera là encore gagner un temps précieux.

4.4 RECHERCHER L'ORIGINE DU PROBLÈME

- Quelques pistes de recherche pour cerner le problème
- Liste non exhaustive

Les problèmes pouvant survenir sont trop nombreux pour pouvoir tous les lister, chaque élément matériel ou logiciel d'une architecture pouvant subir de nombreux types de défaillances.

Cette section liste quelques pistes classiques d'investigation à ne pas négliger pour s'efforcer de cerner au mieux l'étendue du problème, et en déterminer les conséquences.

4.4.1 PRÉREQUIS

- Avant de commencer à creuser
  - référencer les symptômes
  - identifier au mieux l'instant de démarrage du problème

La première étape est de déterminer aussi précisément que possible les symptômes observés, sans en négliger, et à partir de quel moment ils sont apparus.

Cela donne des informations précieuses sur l'étendue du problème, et permet d'éviter de se focaliser sur un symptôme particulier, parce que plus visible (par exemple l'arrêt brutal de l'instance), alors que la cause réelle est plus ancienne (par exemple des erreurs IO dans les traces système, ou une montée progressive de la charge sur le serveur).

4.4.2 RECHERCHE D'HISTORIQUE

- Ces symptômes ont-ils déjà été rencontrés dans le passé ?
- Ces symptômes ont-ils déjà été rencontrés par d'autres ?
- Attention à ne pas prendre les informations trouvées pour argent comptant !

Une fois les principaux symptômes identifiés, il est utile de prendre un moment pour déterminer si ce problème est déjà connu.

Notamment, identifier dans la base de connaissances si ces symptômes ont déjà été rencontrés dans le passé (d'où l'importance de bien documenter les problèmes).

Au delà de la documentation interne, il est également possible de rechercher si ces symptômes ont déjà été rencontrés par d'autres.

Pour ce type de recherche, il est préférable de privilégier les sources fiables (documentation officielle, listes de discussion, plate-forme de support...) plutôt qu'un quelconque document d'un auteur non identifié.

Dans tous les cas, il faut faire très attention à ne pas prendre les informations trouvées pour argent comptant, et ce même si elles proviennent de la documentation interne ou d'une source fiable !

Il est toujours possible que les symptômes soient similaires mais que la cause soit différente. Il s'agit donc ici de mettre en place une base de travail, qui doit être complétée par une observation directe et une analyse.

4.4.3 MATÉRIEL

- Vérifier le système disque (SAN, carte RAID, disques)
- Rechercher toute erreur matérielle
- Firmwares pas à jour
  - ou récemment mis à jour
- Matériel récemment changé

Les défaillances du matériel, et notamment du système de stockage, sont de celles qui peuvent avoir les impacts les plus importants et les plus étendus sur une instance et sur les données qu'elle contient.

Ce type de problème peut également être difficile à diagnostiquer en se contentant d'observer les symptômes les plus visibles. Il est facile de sous-estimer l'ampleur des dégâts.

Parmi les bonnes pratiques, il convient de vérifier la configuration et l'état du système disque (SAN, carte RAID, disques).

Quelques éléments étant une source habituelle de problèmes :

- le système disque n'honore pas les ordres `fsync` ? (SAN ? virtualisation ?)
- quel est l'état de la batterie du cache en écriture ?

Il faut évidemment rechercher la présence de toute erreur matérielle, au niveau des disques, de la mémoire, des CPU…

Vérifier également la version des firmwares installés. Il est possible qu'une nouvelle version corrige le problème rencontré, ou à l'inverse que le déploiement d'une nouvelle version soit à l'origine du problème.

Dans le même esprit, il faut vérifier si du matériel a récemment été changé. Il arrive que de nouveaux éléments soient défaillants.

Il convient de noter que l'investigation à ce niveau peut être grandement complexifiée par l'utilisation de certaines technologies (virtualisation, baies de stockage), du fait de la

mutualisation des ressources, et de la séparation des compétences et des informations de supervision entre différentes équipes.

4.4.4 VIRTUALISATION

- Problèmes de mutualisation des ressources
- Configuration du stockage virtualisé
- Rechercher toute erreur sur l'hôte / la console d'administration
- Mises à jour non appliquées
  - ou appliquées récemment
- Modifications de configuration récentes

Tout comme pour les problèmes au niveau du matériel, les problèmes au niveau du système de virtualisation peuvent être complexes à détecter et à diagnostiquer correctement.

Le principal facteur de problème avec la virtualisation est lié à une mutualisation excessive des ressources.

Il est ainsi possible d'avoir un total de ressources allouées aux VM supérieur à celles disponibles sur l'hyperviseur, ce qui amène à des comportements de fort ralentissement, voire de blocage des systèmes virtualisés.

Si ce type d'architecture est couplé à un système de gestion de bascule automatique (Pacemaker, repmgr…), il est possible d'avoir des situations de bascules impromptues, voire des situations de *split brain*, qui peuvent provoquer des pertes de données importantes. Il est donc important de prêter une attention particulière à l'utilisation des ressources de l'hyperviseur, et d'éviter à tout prix la sur-allocation.

Par ailleurs, lorsque l'architecture inclut une brique de virtualisation, il est important de prendre en compte que certains problèmes ne peuvent être observés qu'à partir de l'hyperviseur, et pas à partir du système virtualisé. Par exemple, les erreurs matérielles ou système risquent d'être invisibles depuis une VM, il convient donc d'être vigilant, et de rechercher toute erreur sur l'hôte.

Il faut également vérifier si des modifications ont été effectuées peu avant l'incident, comme des modifications de configuration ou l'application de mises à jour.

Comme indiqué dans la partie traitant du matériel, l'investigation peut être grandement freinée par la séparation des compétences et des informations de supervision entre différentes équipes. Une bonne communication est alors la clé de la résolution rapide du problème.

4.4.5 SYSTÈME D'EXPLOITATION 1

- Erreurs dans les traces
- Mises à jour système non appliquées
- Modifications de configuration récentes

Après avoir vérifié les couches matérielles et la virtualisation, il faut ensuite s'assurer de l'intégrité du système d'exploitation.

La première des vérifications à effectuer est de consulter les traces du système pour en extraire les éventuels messages d'erreur :

- sous Linux, on trouvera ce type d'informations en sortie de la commande `dmesg`, et dans les fichiers traces du système, généralement situés sous `/var/log` ;
- sous Windows, on consultera à cet effet le journal des événements (les `event logs`).

Tout comme pour les autres briques, il faut également voir s'il existe des mises à jour des paquets qui n'auraient pas été appliquées, ou à l'inverse si des mises à jour, installations ou modifications de configuration ont été effectuées récemment.

4.4.6 SYSTÈME D'EXPLOITATION 2

- Opération d'IO impossible
  - FS plein ?
  - FS monté en lecture seule ?
- Tester l'écriture sur PGDATA
- Tester la lecture sur PGDATA

Parmi les problèmes fréquemment rencontrés se trouve l'impossibilité pour PostgreSQL d'accéder en lecture ou en écriture à un ou plusieurs fichiers.

La première chose à vérifier est de déterminer si le système de fichiers sous-jacent ne serait pas rempli à 100% (commande `df` sous Linux) ou monté en lecture seule (commande `mount` sous Linux).

On peut aussi tester les opérations d'écriture et de lecture sur le système de fichiers pour déterminer si le comportement y est global :

- pour tester une écriture dans le répertoire `PGDATA`, sous Linux :

```
touch $PGDATA/test_write
```

- pour tester une lecture dans le répertoire `PGDATA`, sous Linux :

```
cat $PGDATA/PGVERSION
```

Pour identifier précisément les fichiers présentant des problèmes, il est possible de tester la lecture complète des fichiers dans le point de montage :

```
tar cvf /dev/null $PGDATA
```

4.4.7 SYSTÈME D'EXPLOITATION 3

- Consommation excessive des ressources
  - OOM killer
- Après un crash, vérifier les processus actifs
  - ne pas tenter de redémarrer si des processus persistent

Sous Linux, l'installation d'outils d'aide au diagnostic sur les serveurs est très important pour mener une analyse efficace, particulièrement le paquet `systat` qui permet d'utiliser la commande `sar`.

La lecture des traces système et des traces PostgreSQL permettent également d'avancer dans le diagnostic.

Un problème de consommation excessive des ressources peut généralement être anticipée grâce à une supervision sur l'utilisation des ressources et des seuils d'alerte appropriés. Il arrive néanmoins parfois que la consommation soit très rapide et qu'il ne soit pas possible de réagir suffisamment rapidement.

Dans le cas d'une consommation mémoire d'un serveur Linux qui menacerait de dépasser la quantité totale de mémoire allouable, le comportement par défaut de Linux est d'autoriser par défaut la tentative d'allocation.

Si l'allocation dépasse effectivement la mémoire disponible, alors le système va déclencher un processus *Out Of Memory Killer* (OOM Killer) qui va se charger de tuer les processus les plus consommateurs.

Dans le cas d'un serveur dédié à une instance PostgreSQL, il y a de grandes chances que le processus en question appartienne à l'instance.

S'il s'agit d'un *OOM Killer* effectuant un arrêt brutal (`kill -9`) sur un backend, l'instance PostgreSQL va arrêter immédiatement tous les processus afin de prévenir une corruption de la mémoire et les redémarrer.

S'il s'agit du processus principal de l'instance (*postmaster*), les conséquences peuvent être bien plus dramatiques, surtout si une tentative est faite de redémarrer l'instance sans vérifier si des processus actifs existent encore.

Pour un serveur dédié à PostgreSQL, la recommendation est habituellement de désactiver la sur-allocation de la mémoire, empêchant ainsi le déclenchement de ce phénomène.

Voir pour cela les paramètres kernel `vm.overcommit_memory` et `vm.overcommit_ratio`.

4.4.8 POSTGRESQL

- Relever les erreurs dans les traces
  - ou messages inhabituels
- Vérifier les mises à jour mineures

Tout comme pour l'analyse autour du système d'exploitation, la première chose à faire est rechercher toute erreur ou message inhabituel dans les traces de l'instance. Ces messages sont habituellement assez informatifs, et permettent de cerner la nature du problème. Par exemple, si PostgreSQL ne parvient pas à écrire dans un fichier, il indiquera précisément de quel fichier il s'agit.

Si l'instance est arrêtée suite à un crash, et que les tentatives de redémarrage échouent avant qu'un message puisse être écrit dans les traces, il est possible de tenter de démarrer l'instance en exécutant directement le binaire `postgres` afin que les premiers messages soient envoyés vers la sortie standard.

Il convient également de vérifier si des mises à jour qui n'auraient pas été appliquées ne corrigeraient pas un problème similaire à celui rencontré.

Identifier les mises à jours appliquées récemment et les modifications de configuration peut également aider à comprendre la nature du problème.

4.4.9 PARAMÉTRAGE DE POSTGRESQL 1

- La désactivation de certains paramètres est dangereuse
  - `fsync`
  - `full_page_write`

Si des corruptions de données sont relevées suite à un crash de l'instance, il convient particulièrement de vérifier la valeur du paramètre `fsync`.

En effet, si celui-ci est désactivé, les écritures dans les journaux de transactions ne sont pas effectuées de façon synchrone, ce qui implique que l'ordre des écritures ne sera pas conservé en cas de crash.

Le processus de recovery de PostgreSQL risque alors de provoquer des corruptions si l'instance est malgré tout redémarrée.

Ce paramètre ne devrait jamais être positionné à une autre valeur que `on`, sauf pour répondre à des cas extrêment particuliers (en bref, si l'on peut se permettre de restaurer intégralement les données en cas de crash, par exemple si les tables ne sont utilisées que pour faire du chargement de données).

Le paramètre `full_page_write` indique à PostgreSQL d'effectuer une écriture complète d'une page chaque fois qu'elle reçoit une nouvelle écriture après un checkpoint, pour éviter un éventuel mélange entre des anciennes et nouvelles données en cas d'écriture partielle.

La désactivation de ce paramètre peut avoir le même type de conséquences que la désactivation de `fsync`.

4.4.10 PARAMÉTRAGE DE POSTGRESQL 2

- Option `--data-checksums` de `initdb`
- Disponible depuis PostgreSQL 9.3
- Détecte les corruptions silencieuses
  - au prix d'un impact sur les performances

L'apparition des sommes de contrôles (checksums) permet de se prémunir contre des corruptions silencieuses de données. Pour s'en convaincre, voici un petit exemple.

Tout d'abord, créons une instance sans utiliser les checksums, et une autre qui les utilisera. Attention, on ne peut modifier une instance déjà existante pour changer ce paramètre.

```
$ initdb -D /tmp/sans_checksums/
$ initdb -D /tmp/avec_checksums/ --data-checksums
```

Insérons une valeur de test, sur chacun des deux clusters :

```
postgres=# CREATE TABLE test (name text);
CREATE TABLE

postgres=# INSERT INTO test (name) VALUES ('toto');
INSERT 0 1
```

17.12

On récupère le chemin du fichier de la table pour aller le corrompre à la main (seul celui sans checksums est montré en exemple).

```
postgres=# SELECT pg_relation_filepath('test');
 pg_relation_filepath
----------------------
 base/12036/16317
(1 row)
```

Ensuite, on va s'attacher à corrompre ce fichier, en remplaçant la valeur toto avec un éditeur hexadécimal :

```
$ hexedit /tmp/sans_checksums/base/12036/16317
$ hexedit /tmp/avec_checksums/base/12036/16399
```

Enfin, on peut ensuite exécuter des requêtes sur ces deux clusters.

Sans checksums :

```
postgres=# TABLE test;
 name
------
 qoto
```

Avec checksums :

```
postgres=# TABLE test;
WARNING:  page verification failed, calculated checksum 16321
          but expected 21348
ERROR:  invalid page in block 0 of relation base/12036/16387
```

Côté performances, on peut aussi réaliser un benchmark rapide. Celui-ci a été réalisé en utilisant pgbench, avec une échelle de 1000 (base de 16 Go), en utilisant les tests par défaut, avec la configuration de PostgreSQL laissée par défaut. En utilisant la moyenne de trois exécutions, on constate une dégradation de performations importante, de l'ordre de 20% (226 transactions par seconde avec checksum contre 281 sans).

4.4.11 ERREUR DE MANIPULATION

- Traces système, traces PostgreSQL
- Revue des dernières manipulations effectuées
- Historique des commandes

L'erreur humaine fait également partie des principales causes de désastre.

Une commande de suppression tapée trop rapidement, un oubli de clause `WHERE` dans une requête de mise à jour, nombreuses sont les opérations qui peuvent provoquer des pertes de données ou un crash de l'instance.

Il convient donc de revoir les dernières opérations effectuées sur le serveur, en commençant par les interventions planifiées, et si possible récupérer l'historique des commandes passées.

Des exemples de commandes particulièrement dangereuses :

- `kill -9`
- `rm -rf`
- `rsync`
- `find` (souvent couplé avec des commandes destructices comme `rm`, `mv`, `gzip`...)

4.5 OUTILS

- Quelques outils peuvent aider
  - à diagnostiquer la nature du problème
  - à valider la correction apportée
  - à appliquer un contournement
- ATTENTION
  - certains de ces outils peuvent corrompre les données !

4.5.1 OUTILS - PG_CONTROLDATA

- Fournit des informations de contrôle sur l'instance
- Ne nécessite pas que l'instance soit démarrée

L'outil `pg_controldata` lit les informations du fichier de contrôle d'une instance PostgreSQL.

Cet outil ne se connecte pas à l'instance, il a juste besoin d'avoir un accès en lecture sur le répertoire `PGDATA` de l'instance.

Les informations qu'il récupère ne sont donc pas du temps réel, il s'agit d'une vision de l'instance telle qu'elle était la dernière fois que le fichier de contrôle a été mis à jour. L'avantage est qu'elle peut être utilisée même si l'instance est arrêtée.

pg_controldata affiche notamment les informations initialisées lors d'initdb, telles que la version du catalogue, ou la taille des blocs, qui peuvent être cruciales si l'on veut restaurer une instance sur un nouveau serveur à partir d'une copie des fichiers.

Il affiche également de nombreuses informations utiles sur le traitement des journaux de transactions et des checkpoints, par exemple :

- positions de l'avant-dernier checkpoint et du dernier checkpoint dans les WAL ;
- nom du WAL correspondant au dernier WAL ;
- timeline sur laquelle se situe le dernier checkpoint ;
- instant précis du dernier checkpoint.

Quelques informations de paramétrage sont également renvoyées, comme la configuration du niveau de WAL, ou le nombre maximal de connexions autorisées.

En complément, le dernier état connu de l'instance est également affiché. Les états potentiels sont :

- `in production` : l'instance est démarrée et est ouverte en écriture ;
- `shut down` : l'instance est arrêtée ;
- `in archive recovery` : l'instance est démarrée et est en mode `recovery` (`Warm` ou `Hot Standby`) ;
- `shut down in recovery` : l'instance est arrêtée et un fichier `recovery.conf` existe ;
- `shutting down` : état transitoire, l'instance est en cours d'arrêt ;
- `in crash recovery` : état transitoire, l'instance est en cours de démarrage suite à un crash ;
- `starting up` : état transitoire, concrètement jamais utilisé.

Bien entendu, comme ces informations ne sont pas mises à jour en temps réel, elles peuvent être erronées.

Cet asynchronisme est intéressant pour diagnostiquer un problème, par exemple si `pg_controldata` renvoie l'état `in production` mais que l'instance est arrêtée, cela signifie que l'arrêt n'a pas été effectué proprement (*crash* de l'instance, qui sera donc suivi d'un `recovery` au démarrage).

Exemple de sortie de la commande :

```
-bash-4.2$ /usr/pgsql-10/bin/pg_controldata /var/lib/pgsql/10/data
pg_control version number:            1002
Catalog version number:               201707211
Database system identifier:           6451139765284827825
Database cluster state:               in production
pg_control last modified:             Mon 28 Aug 2017 03:40:30 PM CEST
```

```
Latest checkpoint location:           1/2B04EC0
Prior checkpoint location:            1/2B04DE8
Latest checkpoint's REDO location:    1/2B04E88
Latest checkpoint's REDO WAL file:    000000010000000100000002
Latest checkpoint's TimeLineID:       1
Latest checkpoint's PrevTimeLineID:   1
Latest checkpoint's full_page_writes: on
Latest checkpoint's NextXID:          0:1023
Latest checkpoint's NextOID:          41064
Latest checkpoint's NextMultiXactId:  1
Latest checkpoint's NextMultiOffset:  0
Latest checkpoint's oldestXID:        548
Latest checkpoint's oldestXID's DB:   1
Latest checkpoint's oldestActiveXID:  1022
Latest checkpoint's oldestMultiXid:   1
Latest checkpoint's oldestMulti's DB: 1
Latest checkpoint's oldestCommitTsXid:0
Latest checkpoint's newestCommitTsXid:0
Time of latest checkpoint:            Mon 28 Aug 2017 03:40:30 PM CEST
Fake LSN counter for unlogged rels:   0/1
Minimum recovery ending location:     0/0
Min recovery ending loc's timeline:   0
Backup start location:                0/0
Backup end location:                  0/0
End-of-backup record required:        no
wal_level setting:                    replica
wal_log_hints setting:                off
max_connections setting:              100
max_worker_processes setting:         8
max_prepared_xacts setting:           0
max_locks_per_xact setting:           64
track_commit_timestamp setting:       off
Maximum data alignment:               8
Database block size:                  8192
Blocks per segment of large relation: 131072
WAL block size:                       8192
Bytes per WAL segment:                16777216
Maximum length of identifiers:        64
Maximum columns in an index:          32
```

17.12

```
Maximum size of a TOAST chunk:        1996
Size of a large-object chunk:         2048
Date/time type storage:               64-bit integers
Float4 argument passing:              by value
Float8 argument passing:              by value
Data page checksum version:           0
Mock authentication nonce:            7fb23aca2465c69b2c0f54ccf03e0ece3c0933c5f0e5f2c0
```

4.5.2 OUTILS - EXPORT/IMPORT DE DONNÉES

- pg_dump
- pg_dumpall
- COPY
- psql / pg_restore

Les outils `pg_dump` et `pg_dumpall` permettent d'exporter des données à partir d'une instance démarrée.

Dans le cadre d'un incident grave, il est possible de les utiliser pour :

- extraire le contenu de l'instance ;
- extraire le contenu des bases de données ;
- tester si les données sont lisibles dans un format compréhensible par PostgreSQL.

Par exemple, un moyen rapide de s'assurer que tous les fichiers des tables de l'instance sont lisibles est de forcer leur lecture complète grâce à la commande suivante :

```
pg_dumpall > /dev/null
```

Attention, les fichiers associés aux index ne sont pas parcourus pendant cette opération.

Par ailleurs, ne pas avoir d'erreur ne garantit en aucun cas pas l'intégrité fonctionnelle des données : les corruptions peuvent très bien être silencieuses.

Si pg_dumpall ou pg_dump renvoient des messages d'erreur et ne parviennent pas à exporter certaines tables, il est possible de contourner le problème à l'aide de la commande `COPY`, en sélectionnant exclusivement les données lisibles autour du bloc corrompu.

Il convient ensuite d'utiliser psql ou pg_restore pour importer les données dans une nouvelle instance, probablement sur un nouveau serveur, dans un environnement non affecté par le problème.

Pour rappel, même après un export / import de données réalisé avec succès, des corruptions logiques peuvent encore être présentes.

Il faut donc être particulièrement vigilant et prendre le temps de valider l'intégrité fonctionnelle des données.

4.5.3 OUTILS - PAGEINSPECT

- Extension
- Vision du contenu d'un bloc
- Sans le dictionnaire, donc sans décodage des données
- Affichage brut
- Utilisé surtout en debug, ou dans les cas de corruption
- Fonctions de décodage pour heap (table), bt (btree), entête de page, et FSM
- Nécessite de connaître le code de PostgreSQL

Voici quelques exemples.

Contenu d'une page d'une table :

```
=# select * from heap_page_items(get_raw_page('dspam_token_data',0)) limit 2;
-[ RECORD 1 ]------
lp          | 1
lp_off      | 8152
lp_flags    | 1
lp_len      | 40
t_xmin      | 837
t_xmax      | 839
t_field3    | 0
t_ctid      | (0,7)
t_infomask2 | 3
t_infomask  | 1282
t_hoff      | 24
t_bits      |
t_oid       |
t_data      | \x01000000010000000100000001000000
-[ RECORD 2 ]------
lp          | 2
lp_off      | 8112
lp_flags    | 1
lp_len      | 40
t_xmin      | 837
t_xmax      | 839
t_field3    | 0
t_ctid      | (0,8)
t_infomask2 | 3
t_infomask  | 1282
```

```
t_hoff       | 24
t_bits       |
t_oid        |
t_data       | \x0200000001000000010000002000000
```

Et son entête :

```
=# select * from page_header(get_raw_page('dspam_token_data',0));
-[ RECORD 1 ]--------
lsn       | F1A/5A6EAC40
checksum  | 0
flags     | 0
lower     | 56
upper     | 7872
special   | 8192
pagesize  | 8192
version   | 4
prune_xid | 839
```

Méta-données d'un index (contenu dans la première page) :

```
=# select * from bt_metap('dspam_token_data_uid_key');
-[ RECORD 1 ]-----
magic     | 340322
version   | 2
root      | 243
level     | 2
fastroot  | 243
fastlevel | 2
```

La page racine est la 243. Allons la voir :

```
=# select * from bt_page_items('dspam_token_data_uid_key',243) limit 10;
 offset |    ctid     | len |nulls|vars|                  data
--------+-------------+-----+-----+----+-------------------------------------
      1 | (3,1)       |   8 | f   | f  |
      2 | (44565,1)   |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00
      3 | (242,1)     |  20 | f   | f  | 77 c6 0d 6f a6 92 db 81 28 00 00 00
      4 | (43569,1)   |  20 | f   | f  | 47 a6 aa be 29 e3 13 83 18 00 00 00
      5 | (481,1)     |  20 | f   | f  | 30 17 dd 8e d9 72 7d 84 0a 00 00 00
      6 | (43077,1)   |  20 | f   | f  | 5c 3c 7b c5 5b 7a 4e 85 0a 00 00 00
      7 | (719,1)     |  20 | f   | f  | 0d 91 d5 78 a9 72 88 86 26 00 00 00
      8 | (41209,1)   |  20 | f   | f  | a7 8a da 17 95 17 cd 87 0a 00 00 00
      9 | (957,1)     |  20 | f   | f  | 78 e9 64 e9 64 a9 52 89 26 00 00 00
     10 | (40849,1)   |  20 | f   | f  | 53 11 e9 64 e9 1b c3 8a 26 00 00 00
```

La première entrée de la page 243, correspondant à la donnée `f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00` est stockée dans la page 3 de notre index :

```
=# select * from bt_page_stats('dspam_token_data_uid_key',3);
-[ RECORD 1 ]-+-----
blkno         | 3
type          | i
live_items    | 202
dead_items    | 0
avg_item_size | 19
page_size     | 8192
free_size     | 3312
btpo_prev     | 0
btpo_next     | 44565
btpo          | 1
btpo_flags    | 0

=# select * from bt_page_items('dspam_token_data_uid_key',3) limit 10;
 offset |    ctid     | len |nulls|vars|                data
--------+-------------+-----+-----+----+-------------------------------------
      1 | (38065,1)  |   20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00
      2 | (1,1)      |    8 | f   | f  |
      3 | (37361,1)  |   20 | f   | f  | 30 fd 30 b8 70 c9 01 80 26 00 00 00
      4 | (2,1)      |   20 | f   | f  | 18 2c 37 36 27 03 03 80 27 00 00 00
      5 | (4,1)      |   20 | f   | f  | 36 61 f3 b6 c5 1b 03 80 0f 00 00 00
      6 | (43997,1)  |   20 | f   | f  | 30 4a 32 58 c8 44 03 80 27 00 00 00
      7 | (5,1)      |   20 | f   | f  | 88 fe 97 6f 7e 5a 03 80 27 00 00 00
      8 | (51136,1)  |   20 | f   | f  | 74 a8 5a 9b 15 5d 03 80 28 00 00 00
      9 | (6,1)      |   20 | f   | f  | 44 41 3c ee c8 fe 03 80 0a 00 00 00
     10 | (45317,1)  |   20 | f   | f  | d4 b0 7c fd 5d 8d 05 80 26 00 00 00
```

Le type de la page est `i`, c'est à dire «internal», donc une page interne de l'arbre. Continuons notre descente, allons voir la page 38065 :

```
=# select * from bt_page_stats('dspam_token_data_uid_key',38065);
-[ RECORD 1 ]-+-----
blkno         | 38065
type          | l
live_items    | 169
dead_items    | 21
avg_item_size | 20
page_size     | 8192
free_size     | 3588
btpo_prev     | 118
btpo_next     | 119
btpo          | 0
btpo_flags    | 65

=# select * from bt_page_items('dspam_token_data_uid_key',38065) limit 10;
 offset |     ctid      | len |nulls|vars|                data
```

```
--------+-------------+-----+-----+----+-------------------------------------
       1 | (11128,118) |   20 | f    | f    | 33 37 89 95 b9 23 cc 80 0a 00 00 00
       2 | (45713,181) |   20 | f    | f    | f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00
       3 | (45424,97)  |   20 | f    | f    | f3 4b 2e 8c 39 a3 cb 80 26 00 00 00
       4 | (45255,28)  |   20 | f    | f    | f3 4b 2e 8c 39 a3 cb 80 27 00 00 00
       5 | (15672,172) |   20 | f    | f    | f3 4b 2e 8c 39 a3 cb 80 28 00 00 00
       6 | (5456,118)  |   20 | f    | f    | f3 bf 29 a2 39 a3 cb 80 0f 00 00 00
       7 | (8356,206)  |   20 | f    | f    | f3 bf 29 a2 39 a3 cb 80 28 00 00 00
       8 | (33895,272) |   20 | f    | f    | f3 4b 8e 37 99 a3 cb 80 0a 00 00 00
       9 | (5176,108)  |   20 | f    | f    | f3 4b 8e 37 99 a3 cb 80 0f 00 00 00
      10 | (5466,41)   |   20 | f    | f    | f3 4b 8e 37 99 a3 cb 80 26 00 00 00
```

Nous avons trouvé une feuille (type `1`). Les ctid pointés sont maintenant les adresses dans la table :

```
=# select * from dspam_token_data where ctid = '(11128,118)';
 uid |         token          | spam_hits | innocent_hits |  last_hit
-----+----------------------+-----------+---------------+------------
  40 | -6317261189288392210 |         0 |             3 | 2014-11-10
```

4.5.4 OUTILS - PG_RESETWAL

- Efface les WAL courants
- Permet à l'instance de démarrer en cas de corruption d'un WAL
  - comme si elle était dans un état cohérent
  - ce qui n'est pas le cas
- Cet outil est dangereux !!!
- Utiliser cet outil va corrompre des données

`pg_resetwal` est un outil fourni avec PostgreSQL.

Il s'utilise manuellement, en ligne de commande. Sa fonctionnalité principale est d'effacer les fichiers WAL courants, et il se charge également de réinitialiser les informations correspondantes du fichier de contrôle.

Il est possible de lui spécifier les valeurs à initialiser dans le fichier de contrôle si l'outil ne parvient pas à les déterminer (par exemple, si tous les WAL dans le répertoire `pg_wal` ont été supprimés).

Attention, pg_resetwal ne doit **jamais** être utilisé sur une instance démarrée.

Avant d'exécuter l'outil, il faut toujours vérifier qu'il ne reste aucun processus de l'instance.

L'objectif de pg_resetwal est de pouvoir démarrer une instance après un crash si des corruptions de fichiers (typiquement WAL ou fichier de contrôle) empêche ce démarrage.

Cette action n'est pas une action de réparation ! La réinitialisation des journaux de transactions implique que des transactions qui n'étaient que partiellement validées ne seront pas détectées comme telles, et ne seront donc pas annulées lors du *recovery*.

La conséquence est que les données de l'instance ne sont plus cohérentes. Il est fort possible d'y trouver des violations de contraintes diverses (notamment clés étrangères), ou d'autres cas d'incohérences plus difficiles à détecter.

Après la réinitialisation des WAL, une fois que l'instance a démarré, **il ne faut surtout pas ouvrir les accès à l'application** ! Comme indiqué, les données présentent sans aucun doute des incohérences, et toute action en écriture à ce point ne ferait qu'aggraver le problème.

L'étape suivante est donc de faire un export immédiat des données, de les restaurer dans une nouvelle instance initialisée à cet effet (de préférence sur un nouveau serveur, surtout si l'origine de la corruption n'a pas été clairement identifiée), et ensuite de procéder à une validation méthodique des données.

Il est probable que certaines données incohérentes puissent être identifier à l'import, lors de la phase de recréation des contraintes : celles-ci échoueront si les données ne les respectent, ce qui permettra de les identifier.

En ce qui concerne les incohérences qui passeront au travers de ces tests, il faudra les trouver et les corriger manuellement, en procédant à une validation fonctionnelle des données.

ATTENTION !!!

Il faut donc bien retenir les points suivants :

- pg_resetwal n'est pas magique ;
- pg_resetwal rend les données incohérentes (ce qui est souvent pire qu'une simple perte d'une partie des données, comme on aurait en restaurant une sauvegarde).
- n'utiliser pg_resetwal que s'il n'y a aucun autre moyen de faire autrement pour récupérer les données ;
- ne pas l'utiliser sur l'instance ayant subit le problème, mais sur une copie complète effectuée à froid ;
- après usage, exporter toutes les données et les importer dans une nouvelle instance ;
- valider soigneuseuement les données de la nouvelle instance.

4.6 CAS TYPE DE DÉSASTRES

- Les cas suivants sont assez rares
- Ils nécessitent généralement une restauration
- Certaines manipulations à haut risque sont possibles
  - mais complètement déconseillées !

Cette section décrit quelques-unes des pires situations de corruptions que l'on peut être amené à observer.

Dans la quasi-totalité des cas, la seule bonne réponse est la restauration de l'instance à partir d'une sauvegarde fiable.

4.6.1 AVERTISSEMENT

- Privilégier une solution fiable (restauration, bascule)
- Les actions listées ici sont destructives
- La plupart peuvent (et vont) provoquer des incohérences
- Travailler sur une copie

La plupart des manipulations mentionnées dans cette partie sont destructives, et peuvent (et vont) provoquer des incohérences dans les données.

Tous les experts s'accordent pour dire que l'utilisation de telles méthodes pour récupérer une instance tend à aggraver le problème existant ou à en provoquer de nouveaux, plus graves. S'il est possible de l'éviter, ne pas les tenter (ie : préférer la restauration d'une sauvegarde) !

S'il n'est pas possible de faire autrement (ie : pas de sauvegarde utilisable, données vitales à extraire…), alors TRAVAILLER SUR UNE COPIE.

Il ne faut pas non plus oublier que chaque situation est unique, il faut prendre le temps de bien cerner l'origine du problème, documenter chaque action prise, s'assurer qu'un retour arrière est toujours possible.

4.6.2 CORRUPTION DE BLOCS DANS DES INDEX

- Messages d'erreur lors des accès par l'index
- Données différentes entre un indexscan et un seqscan
- Supprimer et recréer l'index (`REINDEX`)

Les index sont des objets de structure complexe, ils sont donc particulièrement vulnérables aux corruptions.

Lorsqu'un index est corrompu, on aura généralement des messages d'erreur de ce type :

```
ERROR: invalid page header in block 5869177 of relation base/17291/17420
```

Il peut arriver qu'un bloc corrompu ne renvoie pas de message d'erreur à l'accès, mais que les données elles-mêmes soient altérées.

Ce cas est néanmoins très rare dans un bloc d'index.

Dans la plupart des cas, si les données de la table sous-jacente ne sont pas affectées, il est possible de réparer l'index en le reconstruisant intégralement grâce à la commande `REINDEX`.

4.6.3 CORRUPTION DE BLOCS DANS DES TABLES 1

- Cas plus problématique
- Restauration probablement nécessaire

Les corruptions de blocs vont généralement déclencher des erreurs du type suivant :

```
ERROR: invalid page header in block 32570 of relation base/16390/2663
ERROR: could not read block 32570 of relation base/16390/2663:
       read only 0 of 8192 bytes
```

Si la relation concernée est une table, tout ou partie des données contenues dans ces blocs est perdu.

L'apparition de ce type d'erreur est un signal fort qu'une restauration est certainement nécessaire.

4.6.4 CORRUPTION DE BLOCS DANS DES TABLES 2

- Le paramètre `zero_damaged_pages` peut aider
- Des données vont certainement être perdues

Néanmoins, s'il est nécessaire de lire le maximum de données possibles de la table, il est possible d'utiliser l'option de PostgreSQL `zero_damaged_pages` pour demander au moteur de réinitialiser les blocs invalides à zéro lorsqu'ils sont lus.

Par exemple :

17.12

```
> SET zero_damaged_pages = true ;
SET
> VACUUM FULL mycorruptedtable ;
WARNING:  invalid page header in block 32570 of relation base/16390/2663;
          zeroing out page
VACUUM
```

Si cela se termine sans erreur, les blocs invalides ont été réinitialisés.

Les données qu'ils contenaient sont évidemment perdues, mais la table peut désormais être accédée dans son intégralité en lecture, permettant ainsi par exemple de réaliser un export des données pour récupérer ce qui peut l'être.

Attention, du fait des données perdues, le résultat peut être incohérent (contraintes non respectées...).

Par ailleurs, par défaut PostgreSQL ne détecte pas les corruptions logiques, c'est-à-dire n'affectant pas la structure des données mais uniquement le contenu.

Il ne faut donc pas penser que la procédure d'export complet de données suivi d'un import sans erreur garanti l'absence de corruption.

4.6.5 CORRUPTION DE BLOCS DANS DES TABLES 3

- Si la corruption est importante, l'accès au bloc peut faire crasher l'instance
- Il est tout de même possible de réinitialiser le bloc
  - identifier le fichier à l'aide de `pg_relation_filepath()`
  - trouver le bloc avec `ctid` / `pageinspect`
  - réinitialiser le bloc avec `dd`
  - il faut vraiment ne pas avoir d'autre choix

Dans certains cas, il arrive que la corruption soit suffisamment importante pour que le simple accès au bloc fasse crasher l'instance.

Dans ce cas, le seul moyen de réinitialiser le bloc est de le faire manuellement au niveau du fichier, instance arrêtée, par exemple avec la commande `dd`.

Pour identifier le fichier associé à la table corrompue, il est possible d'utiliser la fonction `pg_relation_filepath()` :

```
> select pg_relation_filepath('test_corruptindex') ;
 pg_relation_filepath
----------------------
 base/16390/40995
(1 row)
```

Le résultat donne le chemin vers le fichier principal de la table, relatif au `PGDATA` de l'instance.

Attention, une table peut contenir plusieurs fichiers.

Par défaut une instance PostgreSQL sépare les fichiers en segments de 1 Go (paramètre `segment_size`).

Une table dépassant cette taille aura donc des fichiers supplémentaires (`base/16390/40995.1`, `base/16390/40995.2`…).

Pour trouver le fichier contenant le bloc corrompu, il faudra donc prendre en compte le numéro du bloc trouvé dans le champs `ctid`, multiplier ce numéro par la taille du bloc (paramètre `block_size`, 8 Ko par défaut), et diviser le tout par la taille du segment.

Cette manipulation est évidemment extrêmement risquée, la moindre erreur pouvant rendre irrécupérables de grandes portions de données.

Il est donc fortement déconseillé de se lancer dans ce genre de manipulations à moins d'être absolument certain que c'est indispensable.

Encore une fois, ne pas oublier de travailler sur une copie, et pas directement sur l'instance de production.

4.6.6 CORRUPTION DES WAL 1

- Situés dans le répertoire `pg_wal`
- Les WAL sont nécessaires au *recovery*
- Démarrage impossible s'ils sont corrompus ou manquants
- Si les fichiers WAL ont été archivés, les récupérer
- Sinon, la restauration est la seule solution viable

Les fichiers WAL sont les journaux de transactions de PostgreSQL.

Leur fonction est d'assurer que les transactions qui ont été effectuées depuis le dernier checkpoint ne seront pas perdues en cas de crash de l'instance.

Si certains sont corrompus ou manquants (rappel : il ne faut JAMAIS supprimer les fichiers WAL si le système de fichiers est plein !), alors PostgreSQL ne pourra pas redémarrer.

Si l'archivage était activé et que les fichiers WAL affectés ont bien été archivés, alors il est possible de les restaurer avant de tenter un nouveau démarrage.

Si ce n'est pas possible ou des fichiers WAL archivés ont également été corrompus ou supprimés, l'instance ne pourra pas redémarrer.

Dans cette situation, comme dans la plupart des autres évoquées ici, la seule solution permettant de s'assurer que les données le seront pas corrompues est de procéder à une restauration de l'instance.

4.6.7 CORRUPTION DES WAL 2

- `pg_resetwal` permet de forcer le démarrage
- ATTENTION !!!
  - cela va provoquer des pertes de données
  - des corruptions de données sont également probables
  - ce n'est pas une action corrective !

L'utilitaire `pg_resetwal` a comme fonction principale de supprimer les fichiers WAL courants et d'en créer un nouveau, avant de mettre à jour le fichier de contrôle pour permettre le redémarrage.

Au minimum, cette action va provoquer la perte de toutes les transactions validées effectuées depuis le dernier checkpoint.

Il est également probable que des incohérences vont apparaître, certaines relativement simples à détecter via un export / import (incohérences dans les clés étrangères par exemple), certaines complètement invisibles.

L'utilisation de cet utilitaire est extrêmement dangereuse, n'est pas recommandée, et ne peut jamais être considérée comme une action corrective.

Il faut toujours privilégier la restauration d'une sauvegarde plutôt que son exécution.

Si l'utilisation de `pg_resetwal` était néanmoins nécessaire (par exemple pour récupérer quelques données absentes de la sauvegarde), alors il faut travailler sur une copie des fichiers de l'instance, récupérer ce qui peut l'être à l'aide d'un export de données, et les importer dans une autre instance.

Les données récupérées de cette manière devraient également être soigneusement validées avant d'être importée de façon à s'assurer qu'il n'y a pas de corruption silencieuse.

Il ne faut en aucun cas remettre une instance en production après une réinitialisation des WAL.

4.6.8 CORRUPTION DU FICHIER DE CONTRÔLE

- Fichier `global/pg_control`
- Contient les informations liées au dernier checkpoint
- Sans lui, l'instance ne peut pas démarrer
- Restauration nécessaire

Le fichier de contrôle de l'instance contient de nombreuses informations liées à l'activité et au statut de l'instance, notamment l'instant du dernier checkpoint, la position correspondante dans les WAL, le numéro de transaction courant et le prochain à venir...

Ce fichier est le premier lu par l'instance. S'il est corrompu ou supprimé, l'instance ne pourra pas démarrer.

Il est possible de forcer la réinitialisation de ce fichier à l'aide de la commande `pg_resetwal`, qui va se baser par défaut sur les informations contenues dans les fichiers WAL présents pour tenter de "deviner" le contenu du fichier de contrôle.

Ces informations seront très certainement erronées, potentiellement à tel point que même l'accès aux bases de données par leur nom ne sera pas possible :

```
-bash-4.2$ pg_isready
/var/run/postgresql:5432 - accepting connections

-bash-4.2$ psql postgres
psql: FATAL:  database "postgres" does not exist
```

Encore une fois, utiliser `pg_resetwal` n'est en aucun cas une solution, mais doit uniquement être considéré comme un contournement temporaire à une situation désastreuse.

Une instance altérée par cet outil ne doit pas être considérée comme saine.

4.6.9 CORRUPTION DU CLOG

- Fichier contenu dans `pg_xact`
- Statut des différentes transactions
- Son altération risque de causer des incohérences

Le fichier CLOG (*Commit Log*) contient le statut des différentes transactions, notamment si celles-ci sont en cours, validées ou annulées.

S'il est altéré ou supprimé, il est possible que des transactions qui avaient été marquées

comme annulées soient désormais considérées comme valide, et donc que les modifications de données correspondantes deviennent visibles aux autres transactions.

C'est évidemment un problème d'incohérence majeur, tout problème avec ce fichier devrait donc être soigneusement analysé.

Il est préférable dans le doute de procéder à une restauration et d'accepter une perte de données plutôt que de risquer de maintenir des données incohérentes dans la base.

4.6.10 CORRUPTION DU CATALOGUE SYSTÈME

- Le catalogue contient la définition du schéma
- Sans lui, les données sont inaccessibles
- Situation très délicate

Le catalogue système contient la définition de toutes les relations, les méthodes d'accès, la correspondance entre un objet et un fichier sur disque, les types de données existantes…

S'il est incomplet, corrompu ou inaccessible, l'accès aux données en SQL risque de ne pas être possible du tout.

Cette situation est très délicate, et appelle là encore une restauration.

Si le catalogue était complètement inaccessible, sans sauvegarde la seule solution restante serait de tenter d'extraire les données directement des fichiers data de l'instance, en oubliant toute notion de cohérence, de type de données, de relation…

Personne ne veut faire ça.

4.7 CONCLUSION

- Les désastres peuvent arriver
- Il faut s'y être préparé
- Faites des sauvegardes !
  - et testez les

4.8 TRAVAUX PRATIQUES

4.8.1 ÉNONCÉ

L'objectif de ce TP est d'effectuer des manipulations de diagnostic, d'extraction de données et de restauration sur une instance corrompue.

Le formateur va passer sur chacun des postes pour simuler un cas de crash et de corruption de l'instance. Plusieurs cas sont possibles.

Le travail est de tester l'état de l'instance, de détecter où se situe la corruption, de comprendre l'origine du problème, et d'extraire les données qui peuvent être sauvées lorsque c'est possible.

Une fois les actions effectuées (et documentées), restaurez l'instance à partir d'une sauvegarde antérieure à la corruption.

4.8.2 CONSEILS

Ne pas oublier :

- travaillez sur une copie des fichiers corrompus ;
- validez le bon fonctionnement en exportant les données et en les important dans une nouvelle instance ;
- assurez-vous d'avoir une sauvegarde valide avant que le formateur ne lance le script ;
- et de savoir la restaurer ;
- documentez chaque action entreprise, chaque diagnostic effectué.

Regardez attentivement les messages d'erreur (si aucun n'est écrit dans les traces, lancez le processus à la main pour que les messages s'affichent à l'écran).

Si l'instance ne démarre pas du tout, il faudra certainement utiliser `pg_resetwal` … attention toutefois à bien s'assurer que c'est le seul moyen possible, et regardez ensuite attentivement les conséquences sur la cohérence des données.

Attention, toutes les corruptions ne déclenchent pas forcément de message d'erreur ! Pensez à valider l'intégrité des donnés.