DALIBO

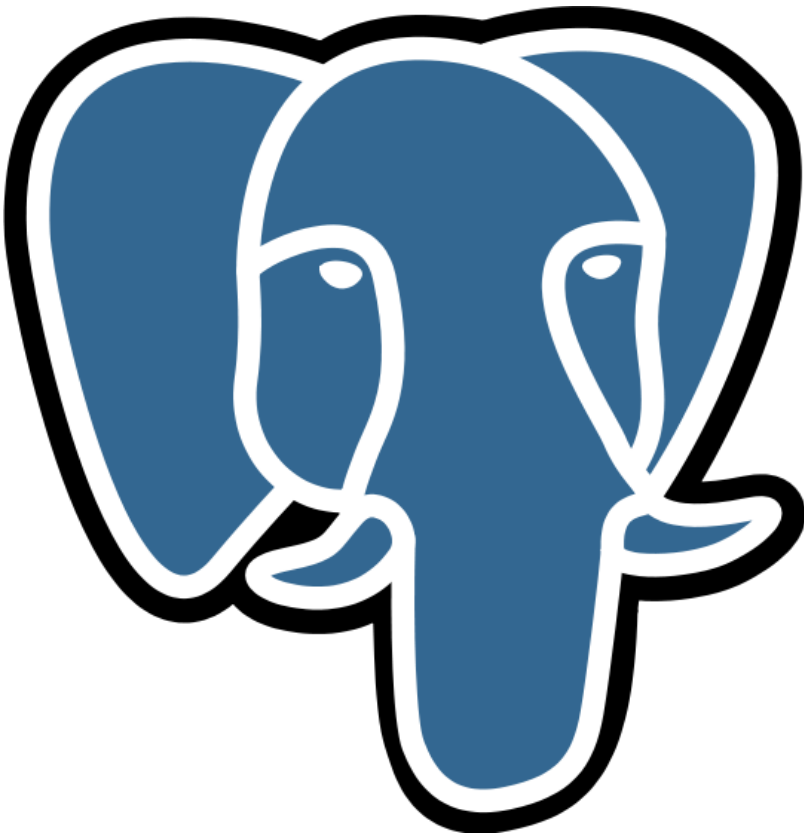
L'expertise PostgreSQL

Sauvegardes Avancées

Pour toute information complémentaire, contactez :
formation@dalibo.com

Table des matières

Sauvegardes Avancées



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 pratique pour produire des scripts de sauvegarde et restauration sûrs et efficaces.

1.1 Au menu

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

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 nécessaire pour produire des scripts de sauvegarde, afin d'intégrer la sauvegarde des bases de données PostgreSQL dans son SI.

1.3 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 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.

2.1 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 Différentes approches

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

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

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

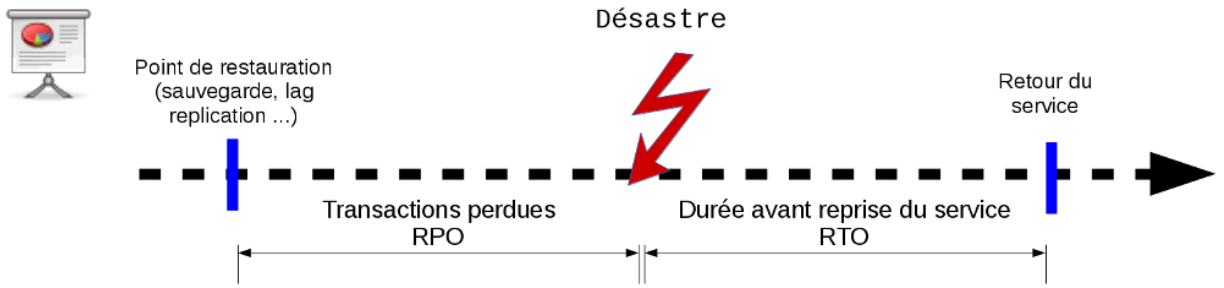
La sauvegarde logique permet de créer un fichier texte de commandes SQL ou un fichier binaire contenant le schéma et les données de la base de données.

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

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

2.3 RTO/RPO

- RPO (Recovery Point Objective) : Perte de Données Maximale Admissible
- RTO (Recovery Time Objective) : Durée Maximale d'Interruption Admissible
- ⇒ Permettent de définir la politique de sauvegarde/restauration



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

- RPO faible : Le perte de données admissible est très faible voire nulle, il faudra s'orienter vers des solutions de type :
  - Sauvegarde à chaud
  - PITR
  - Réplication (asynchrone/synchrone).
- RPO important : On s'autorise une perte de données importante, on peut utiliser des solutions de type :
  - Sauvegarde logique (dump)
  - Sauvegarde fichier à froid
- RTO court : Durée d'interruption courte, le service doit vite remonter. Nécessite des procédures avec le moins de manipulations possible et réduisant le nombre d'acteurs :
  - Réplication
  - Solutions Haut Disponibilité
- RTO long : La durée de reprise du service n'est pas critique on peut utiliser des solutions simple comme :
  - Restauration fichier
  - Restauration sauvegarde logique (dump).

Plus le besoin en RTO/RPO sera court 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.

2.4 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 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 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.

2.5 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 à 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.

2.6 Points d'attention

- Externaliser les sauvegardes
- Stocker plusieurs copies
  - 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. Simplement 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.

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.

3 Sauvegardes logiques

- Extraction à chaud des données dans un fichier
- Photo des données au début de l'opération
  - Quelle que soit la durée
  - 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.

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 (rappel : une instance PostgreSQL contient plusieurs bases de données) dans différents formats.

`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 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.

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 gzip, 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 gzip.

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 ;
- définition des droits d'accès aux objets.

Les index sont la dernière étape 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 les contraintes. 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
```

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

3.3 Limites

- Le format `plain` (SQL) rend très difficile les restaurations partielles
- Le format `tar` souffre des limites inhérentes au format tar
- Seul le format `directory` permet la parallélisation du dump (9.3+)
- Seul `pg_dumpall` sauvegarde la définition des objets globaux (rôles, tablespaces)
- `pg_dumpall` ne supporte que le format plain

⇒ 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.

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 privilégiera 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.

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.

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 à la restore

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. Il est donc nécessaire de lancer au moins l'opération de maintenance ANALYZE sur les objets restaurés.

3.6 Options de connexion

- -h / $PGHOST / socket Unix
- -p / $PGPORT / 5432
- -U / $PGUSER / utilisateur du système
- -d / $PGDATABASE / utilisateur de connexion
- $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 psql à demander un mot de passe (en interactif donc).

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 exécute 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.

3.7 Impact des privilèges

- Les outils se comportent comme des clients pour PostgreSQL
- Préférer un rôle super-utilisateur autorisé dans pg_hba.conf
- Sinon :
  - 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 de 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 restaurer qu'avec un super-utilisateur.

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.

3.9 pg_dumpall - Options

- Limiter le dump au données globales (rôles et tablespaces) : `-g`
  - Limiter le dump aux rôles : `-r`
  - Limiter le dump aux tablespaces : `-t`
- Sélection : `-a`, `-s`, `-0`, `-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.

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 créé 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.

Dans ce cas, les options permettent de contrôler le comportement en cas d'erreur :

- `-v ON_ERROR_ROLLBACK=ON` : psql créé 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.

3.11 pg_restore - Options

- **Attention** : `-f` indique le 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 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.

3.12 pg_restore - Base de données

- -d indique la base de données de connexion
- Avec -C (créer la base de données cible) :
  1. pg_restore se connecte (-d) et exécute CREATE DATABASE
  2. 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, puis il se connecte à la base nouvelle 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';
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`.

3.13 pg_restore - Table des matières

- Pour les sélections plus complexes :
  1. Obtenir la table des matières avec `-l`
  2. Choisir les éléments à restaurer
  3. 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 fichier particulier. Par exemple :

```
$ pg_restore -l b1.dump
;
; Archive created at Tue Dec 13 09:35:17 2011
;     dbname: b1
;     TOC Entries: 16
;     Compression: -1
;     Dump Version: 1.12-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 9.1.3
;     Dumped by pg_dump version: 9.1.3
;
;
; Selected TOC Entries:
;
2752; 1262 37147 DATABASE - b1 guillaume
5; 2615 2200 SCHEMA - public guillaume
2753; 0 0 COMMENT - SCHEMA public guillaume
2754; 0 0 ACL - public guillaume
164; 3079 12528 EXTENSION - plpgsql
2755; 0 0 COMMENT - EXTENSION plpgsql
165; 1255 37161 FUNCTION public f1() guillaume
161; 1259 37148 TABLE public t1 guillaume
```

```
162; 1259 37151 TABLE public t2 guillaume
163; 1259 37157 VIEW public v1 guillaume
2748; 0 37148 TABLE DATA public t1 guillaume
2749; 0 37151 TABLE DATA public t2 guillaume
2747; 2606 37163 CONSTRAINT public t2_pkey guillaume
2745; 1259 37164 INDEX public t1_c1_idx guillaume
```

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 t1
pg_restore: creating TABLE t2
pg_restore: creating VIEW v1
pg_restore: restoring data for table "t1"
pg_restore: creating CONSTRAINT t2_pkey
pg_restore: setting owner and privileges for SCHEMA public
pg_restore: setting owner and privileges for COMMENT SCHEMA public
pg_restore: setting owner and privileges for ACL public
pg_restore: setting owner and privileges for EXTENSION plpgsql
pg_restore: setting owner and privileges for COMMENT EXTENSION plpgsql
pg_restore: setting owner and privileges for TABLE t1
pg_restore: setting owner and privileges for TABLE t2
pg_restore: setting owner and privileges for VIEW v1
pg_restore: setting owner and privileges for TABLE DATA t1
pg_restore: setting owner and privileges for CONSTRAINT 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.

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_xlog, 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.

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 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 :

1. Arrêt de l'instance PostgreSQL
2. Création des snapshots de l'ensemble des systèmes de fichiers
3. Démarrage de l'instance PostgreSQL
4. Sauvegarde des fichiers à partir des snapshots
5. Destruction des snapshots

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

1. rsync de l'ensemble des fichiers de l'instance PostgreSQL démarrée
2. Arrêt de l'instance PostgreSQL
3. rsync de l'ensemble des fichiers de l'instance pour ne transférer que les différences
4. Démarrage de l'instance PostgreSQL

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 lancer PostgreSQL, les index sont conservés et aucune configuration supplémentaire n'est nécessaire.

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 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.

5 Sauvegardes physiques à chaud et PITR

- Sauvegarde en deux parties :
  1. Les fichiers de données
  2. 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.

5.1 Le mode recovery

- PostgreSQL écrit les données deux fois :
  1. D'abord dans les journaux de transactions (fichiers WAL)
  2. 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 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 :

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

5.2 Archivage des journaux de transaction

- Choisir le répertoire d'archivage
- Configurer PostgreSQL, dans postgresql.conf :
  - `wal_level` : `archive`, `hot_standby` ou `logical`
  - `archive_mode` : on
  - `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 `archive`, `hot_standby` ou `logical` (9.4+). La valeur `logical` inclut `hot_standby`, qui inclut `archive`, la valeur minimum est donc `archive`.

Ensuite, il faut activer l'archivage en plaçant `archive_mode` à on.

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_xlog/000000010000000E0000001F
/var/lib/pgsql/archives/000000010000000E0000001F
```

Il est tout à fait possible de copier le fichiers 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_xlog`. 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 puis 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 vides, 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.

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)
  - Tablespaces
  - Configuration
- 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 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`.

Ensuite, il faut copier l'ensemble des fichiers de l'instance, à l'exception de `pg_xlog`. En effet, puisque la restauration utilisera sur les journaux de transactions archivés, le contenu de ce répertoire n'est pas nécessaire. Hormis `pg_xlog`, tous les répertoires de $PGDATA doivent être sauvegardés. 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()` (sans arguments) pour indiquer à PostgreSQL que la sauvegarde est terminée. 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`.

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_xlog` (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 strictement 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 permissions 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 permissions 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_xlog`) 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 va empêcher le démarrage de l'instance.

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_name`, `recovery_target_time`, `recovery_target_xid`
  - `recovery_target`
  - `recovery_end_command`, `pause_at_recovery_target`

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 000000010000582000000F4)
STOP WAL LOCATION: 583/9B0752D8 (file 000000010000005830000009B)
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 000000010000582000000F4 et 000000010000005830000009B sont nécessaires pour restaurer la sauvegarde. Les fichiers antérieurs au WAL 000000010000582000000F4 ne sont pas nécessaires. Les fichiers postérieurs au WAL 000000010000005830000009B 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, c'est le paramètre qui 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_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.

Le paramètre `pause_at_recovery_target` n'est pris en compte que si le paramètre `hot_standby` est activé dans le fichier `postgresql.conf` de l'instance. 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 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_xlog_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'
```

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*).

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 archivé 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 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_xlog/
000000010000000000000007
000000010000000000000008
000000010000000000000009
00000001000000000000000A
00000001000000000000000B
00000001000000000000000C
00000001000000000000000D
00000001000000000000000E
00000001000000000000000F
000000010000000000000010
000000010000000000000011
```

```
000000020000000000000009
000000020000000000000000A
000000020000000000000000B
000000020000000000000000C
00000002.history
```

A 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 :

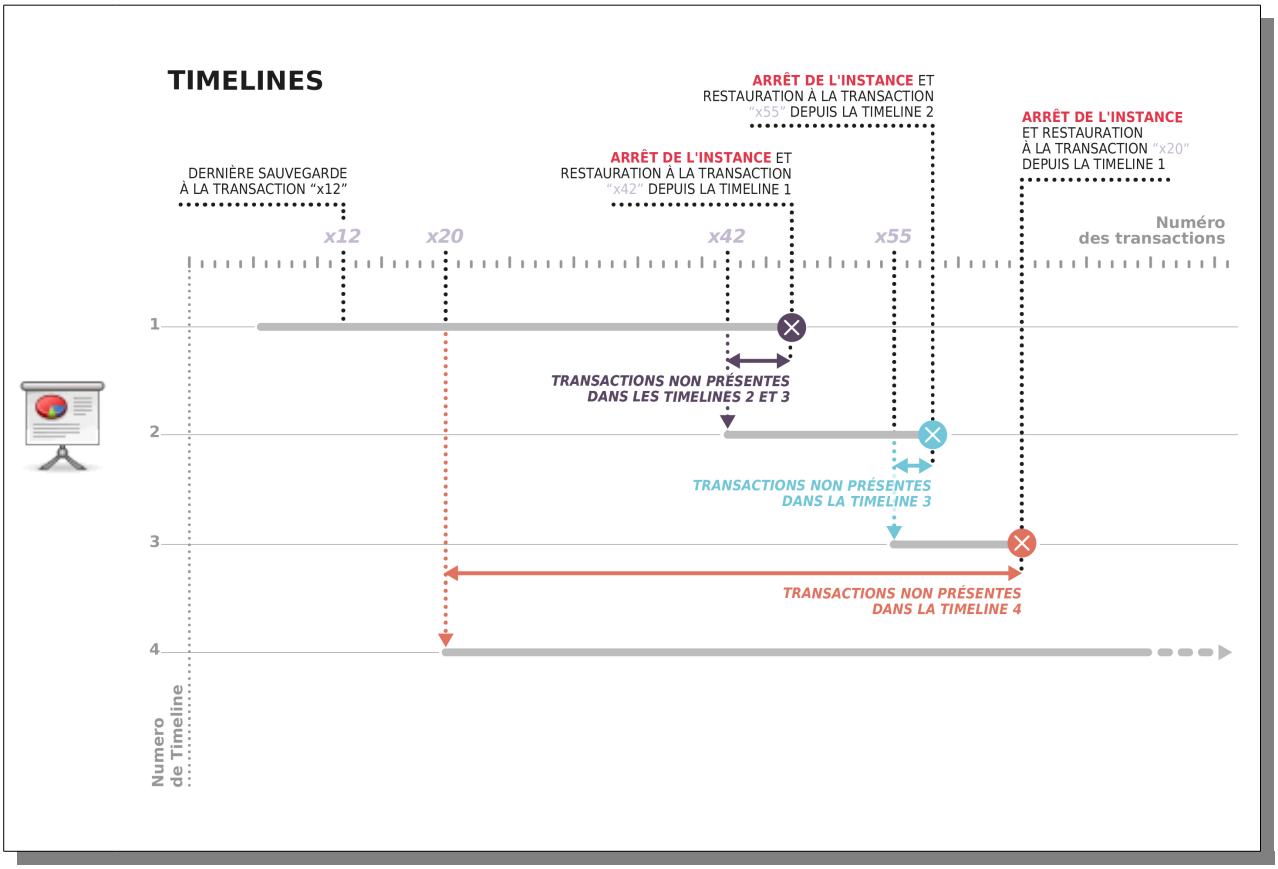
```
$ cat 00000003.history
1	0/9765A80	before 2015-10-20 16:59:30.103317+02

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.

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

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 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 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 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.

5.8 Adapter la configuration

- postgresql.conf
  - port
  - listen_adresses
  - 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.

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 limitation 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.

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.

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.

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 en cours 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.

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.

6.3 Spécificités d'un script de sauvegarde logique

- `pg_dumpall -g` : définition des rôles et des tablespaces
- `pg_dump` : contenu de chaque base 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 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 commande `psql \l`. La configuration par base de données et rôle est disponible en utilisant la commande `psql \drds`.

6.4 Spécificités d'un script de sauvegarde PITR - archivage des WAL

- 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 !

Pour qu'un script de sauvegarde basée 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 une 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`.

6.5 Spécificités d'un script de sauvegarde PITR - suite

- Utiliser `pg_start_backup()` et `pg_stop_backup()`
  - même en cas d'utilisation d'un mécanisme de snapshot (virtualisation, baie...)
- 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`
- 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

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(<label>)`. 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 ouvert 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 pas défaut relocalisés 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.

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 que lorsque l'application de la politique de rétention supprime une sauvegarde, seuls les WAL archivés qui ne servent qu'à cette sauvegarde ne 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.

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.

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 en cours 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'il contiennent des informations aussi détaillées que possible sur le contexte de l'erreur et les causes possibles.

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 documentation 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 ...

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`

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.

8 Conclusion

- Les techniques de sauvegarde de PostgreSQL sont :
  - Complémentaires
  - Automatisables
- La maîtrise de ces techniques est indispensable pour assurer un service fiable.