missions du DBA

Bases de données 2

Thibaut MADELAINE

octobre 2023

Chers lectrices & lecteurs,

Cette formation PostgreSQL est issue des manuels Dalibo. Ils ont été repris par Thibaut MADELAINE pour rentrer dans le format universitaire avec Cours Magistraux, Travaux Dirigés (sans ordinateurs) et Travaux Pratiques (avec ordinateur).

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

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler sur le site gitlab https://gitlab.com/madtibo/cours_dba_pg_universite/-/issues !

Missions du DBA

Programme de ce cours

- Semaine 1 : découverte de PostgreSQL
- Semaine 2 : transactions et accès concurrents
- **Semaine 3** : missions du DBA
  - sécurité et droits
  - sauvegardes et restaurations
- Semaine 4 : optimisation et indexation
- Semaine 5 : *PL/PgSQL* et triggers

La sécurité dans PostgreSQL

- Sécurité physique d'une instance
- Droits d'accès en PostgreSQL
- Gestion des droits sur les objets

Sécurité physique

- Données stockées dans des fichiers
- Contrôle de l'accès au serveur
- Application des mises à jours de sécurité

PostgreSQL stocke toutes les données dans des fichiers. Si un utilisateur malintentionné accède à ces fichiers, il accède à toutes les données stockées.

PostgreSQL ne démarrera pas si les droits sur le répertoire de l'instance (`data_directory`) est différent de `700` :

```
postgres$ ls -l /var/lib/postgresql/14
total 4
drwx------ 19 postgres postgres 4096 oct.  19 09:38 main
```

Sécuriser les droits sur ces fichiers est important. Sécuriser l'accès physique ou par ssh à la machine l'est tout autant. Si vous donnez un mot de passe à l'utilisateur *postgres*, utilisez un outil tel `apg` pour générer un mot de passe aléatoire solide. Faites de même pour tout utilisateur pouvant se connecter au serveur.

Des failles de sécurité peuvent être découvertes dans des logiciels tierces installés sur votre serveur. Il est important d'appliquer les mises à jours de sécurité régulièrement.

Chiffrements

- Données disques
  - pas en natif
  - pgcrypto
- Connexions
  - SSL
  - avec ou sans certificats serveur et/ou client

PostgreSQL ne chiffre pas les données sur disque. Si l'instance complète doit être chiffrée, il est conseillé d'utiliser un système de fichiers qui propose cette fonctionnalité. Attention au fait que cela ne vous protège que contre la récupération des données sur un disque non monté. Quand le disque est monté, les données sont lisibles suivant les règles d'accès d'Unix.

Néanmoins, il existe un module contrib appelé pgcrypto, permettant d'accéder à des fonctions de chiffrement et de hachage. Cela permet de protéger les informations provenant de colonnes spécifiques. Le chiffrement se fait du côté serveur, ce qui sous-entend que l'information est envoyée en clair sur le réseau. Le chiffrement SSL est donc obligatoire dans ce cas.

Par défaut, les sessions ne sont pas chiffrées. Les requêtes et les données passent donc en clair sur le réseau. Il est possible de les chiffrer avec SSL, ce qui aura une conséquence négative sur les performances. Il est aussi possible d'utiliser les certificats (au niveau serveur et/ou client) pour augmenter encore la sécurité des connexions.

Droits de connexion

- Lors d'une connexion, indication :
  - de l'hôte (socket Unix ou alias/adresse IP)
  - du nom de la base de données
  - du nom du rôle
  - du mot de passe (parfois optionnel)
- Suivant les trois premières informations
  - impose une méthode d'authentification

Lors d'une connexion, l'utilisateur fournit, explicitement ou non, plusieurs informations. PostgreSQL va choisir une méthode d'authentification en se basant sur les informations fournies et sur la configuration d'un fichier appelé pg_hba.conf. *HBA* est l'acronyme de Host Based Authentication.

`pg_hba.conf` et `pg_ident.conf`

- Authentification multiple, suivant l'utilisateur, la base et la source de la connexion.
  - `pg_hba.conf` (*Host Based Authentication*)
  - `pg_ident.conf`, si mécanisme externe d'authentification
  - paramètres `hba_file` et `ident_file`

L'authentification est paramétrée au moyen du fichier `pg_hba.conf`. Dans ce fichier, pour une tentative de connexion à une base donnée, pour un utilisateur donné, pour un transport (IP, IPV6, Socket Unix, SSL ou non), et pour une source donnée, ce fichier permet de spécifier le mécanisme d'authentification attendu.

Si le mécanisme d'authentification s'appuie sur un système externe (LDAP, Kerberos, Radius…), des tables de correspondances entre utilisateur de la base et utilisateur demandant la connexion peuvent être spécifiées dans `pg_ident.conf`.

Ces noms de fichiers ne sont que les noms par défaut. Ils peuvent tout à fait être remplacés en spécifiant de nouvelles valeurs de `hba_file` et `ident_file` dans `postgresql.conf`.

Informations de connexion

- Quatre informations :
  - socket Unix ou adresse/alias IP
  - numéro de port
  - nom de la base
  - nom du rôle
- Fournies explicitement ou implicitement

Tous les outils fournis avec la distribution PostgreSQL (par exemple `createuser`) acceptent des options en ligne de commande pour fournir les informations en question :

- -h pour la socket Unix ou l'adresse/alias IP
- -p pour le numéro de port
- -d pour le nom de la base
- -U pour le nom du rôle

Si l'utilisateur ne passe pas ces informations, plusieurs variables d'environnement sont vérifiées :

- PGHOST pour la socket Unix ou l'adresse/alias IP
- PGPORT pour le numéro de port
- PGDATABASE pour le nom de la base
- PGUSER pour le nom du rôle

Au cas où ces variables ne seraient pas configurées, des valeurs par défaut sont utilisées :

- la socket Unix
- 5432 pour le numéro de port
- suivant l'outil, la base postgres ou le nom de l'utilisateur PostgreSQL, pour le nom de la base
- le nom de l'utilisateur au niveau du système d'exploitation pour le nom du rôle

Autrement dit, quelle que soit la situation, PostgreSQL remplacera les informations non fournies explicitement par des informations provenant des variables d'environnement, voire par des informations par défaut.

Configuration de l'authentification

- PostgreSQL utilise les informations de connexion pour sélectionner la méthode
- Fichier de configuration: `pg_hba.conf`
- Se présente sous la forme d'un tableau
  - 4 colonnes d'informations
  - 1 colonne indiquant la méthode à appliquer
  - 1 colonne optionnelle d'options

Lorsque le serveur PostgreSQL récupère une demande de connexion, il connaît le type de connexion utilisé par le client (socket Unix, connexion TCP SSL, connexion TCP simple, etc). Il connaît aussi l'adresse IP du client (dans le cas d'une connexion via une socket TCP), le nom de la base et celui de l'utilisateur. Il va donc parcourir les lignes du tableau enregistré dans le fichier `pg_hba.conf`. Il les parcourt dans l'ordre. La première qui correspond aux informations fournies lui précise la méthode d'authentification. Il ne lui reste plus qu'à appliquer cette méthode. Si elle fonctionne, la connexion

est autorisée et se poursuit. Si elle ne fonctionne pas, quelle qu'en soit la raison, la connexion est refusée. Aucune autre ligne du fichier ne sera lue.

Il est donc essentiel de bien configurer ce fichier pour avoir une protection maximale.

Le tableau se présente ainsi :

```
# local      DATABASE  USER  METHOD  [OPTIONS]
# host       DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
# hostssl    DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
# hostnossl  DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
```

Colonne type

- 4 valeurs possibles
  - `local`
  - `host`
  - `hostssl`
  - `hostnossl`
- `hostssl` nécessite d'avoir activé `ssl` dans `postgresql.conf`

La colonne type peut contenir quatre valeurs différentes. La valeur `local` concerne les connexions via la socket Unix. Toutes les autres valeurs concernent les connexions via la socket TCP. La différence réside dans l'utilisation forcée ou non du SSL :

- `host`, connexion via la socket TCP, avec ou sans SSL ;
- `hostssl`, connexion via la socket TCP, avec SSL ;
- `hostnossl`, connexion via la socket TCP, sans SSL.

Il est à noter que l'option `hostssl` n'est utilisable que si le paramètre `ssl` du fichier `postgresql.conf` est à on.

Colonne database

- Nom de la base

- Plusieurs bases (séparées par des virgules)
- Nom d'un fichier contenant la liste des bases (précédé par une arobase)
- Mais aussi
  - `all` (pour toutes les bases)
  - `sameuser`, `samerole` (pour la base de même nom que le rôle)
  - `replication` (pour les connexions de réplication physique)

La colonne peut recueillir le nom d'une base, le nom de plusieurs bases en les séparant par des virgules, le nom d'un fichier contenant la liste des bases ou quelques valeurs en dur. La valeur all indique toutes les bases. La valeur replication est utilisée pour les connexions de réplication (il n'est pas nécessaire d'avoir une base nommée replication). Enfin, la valeur `sameuser` spécifie que l'enregistrement n'intercepte que si la base de données demandée a le même nom que le rôle demandé, alors que la valeur `samerole` spécifie que le rôle demandé doit être membre du rôle portant le même nom que la base de données demandée.

Colonne user

- Nom du rôle
- Nom d'un groupe (précédé par un signe plus)
- Plusieurs rôles (séparés par des virgules)
- Nom d'un fichier contenant la liste des rôles (précédé par une arobase)
- Mais aussi
  - `all` (pour tous les rôles)

La colonne peut recueillir le nom d'un rôle, le nom d'un groupe en le précédant d'un signe plus, le nom de plusieurs rôles en les séparant par des virgules, le nom d'un fichier contenant la liste des bases ou quelques valeurs en dur. La valeur all indique tous les rôles.

Colonne adresse IP

- Uniquement dans le cas d'une connexion `host`, `hostssl` et `hostnossl`
- Soit l'adresse IP et le masque réseau
- Soit l'adresse au format CIDR

- Soit un nom DNS

La colonne de l'adresse IP permet d'indiquer une adresse IP ou un sous-réseau IP. Il est donc possible de filtrer les connexions par rapport aux adresses IP, ce qui est une excellente protection.

Voici deux exemples d'adresses IP au format adresse et masque de sous-réseau :

```
192.168.168.1 255.255.255.255
192.168.168.0 255.255.255.0
```

Et voici deux exemples d'adresses IP au format CIDR :

```
192.168.168.1/32
192.168.168.0/24
```

Il est possible d'utiliser un nom d'hôte ou un domaine DNS, au prix d'une recherche DNS pour chaque hostname présent, pour chaque nouvelle connexion.

Colonne méthode

- Précise la méthode d'authentification à utiliser
- Deux types de méthodes
  - internes
  - externes
- Possibilité d'ajouter des options dans une dernière colonne

La colonne de la méthode est la dernière colonne, voire l'avant-dernière si vous voulez ajouter une option à la méthode d'authentification.

Colonne options

- Dépend de la méthode d'authentification
- Méthode externe : option `map`

Les options disponibles dépendent de la méthode d'authentification sélectionnée. Toutes les méthodes externes permettent l'utilisation de l'option map. Cette option a pour but d'indiquer la carte de correspondance à sélectionner dans le fichier `pg_ident.conf`.

Méthodes internes

- `trust`: à proscrire
- `password` et `md5` : dépassées
- `scram-sha-256`
- `reject`

Le support de `scram-sha-256` est disponible à partir de la version 10. ELle est devenue la méthode par défaut en version 14. Cette méthode est la plus sûre.

La méthode `reject` est intéressante dans certains cas de figure. Par exemple, on veut que le rôle u1 puisse se connecter à la base de données b1 mais pas aux autres. Voici un moyen de le faire (pour une connexion via les sockets Unix) :

```
local  b1   u1  scram-sha-256
local  all  u1  reject
```

La méthode `trust` est certainement la pire. À partir du moment où le rôle est reconnu, aucun mot de passe n'est demandé. Si le mot de passe est fourni malgré tout, il n'est pas vérifié. Il est donc essentiel de proscrire cette méthode d'authentification.

La méthode `password` force la saisie d'un mot de passe. Cependant, ce dernier est envoyé en clair sur le réseau. Il n'est donc pas conseillé d'utiliser cette méthode, surtout sur un réseau non sécurisé.

La méthode `md5` est la méthode conseillée avant la version 10. La saisie du mot de passe est forcée. De plus, le mot de passe transite chiffré en md5.

Méthodes externes

- `ldap`, `radius`, `cert`
- `gss`, `sspi`

- `ident`, **`peer`**, `pam`

Ces différentes méthodes permettent d'utiliser des annuaires d'entreprise comme RADIUS, LDAP ou un ActiveDirectory. Certaines méthodes sont spécifiques à Unix (comme `ident` et `peer`), voire à Linux (comme `pam`).

La méthode LDAP utilise un serveur LDAP pour authentifier l'utilisateur.

La méthode `GSSAPI` correspond au protocole du standard de l'industrie pour l'authentification sécurisée définie dans RFC 2743. PostgreSQL supporte `GSSAPI` avec l'authentification Kerberos suivant la RFC 1964 ce qui permet de faire du « Single Sign-On ».

La méthode `Radius` permet d'utiliser un serveur RADIUS pour authentifier l'utilisateur.

La méthode `ident` permet d'associer les noms des utilisateurs du système d'exploitation aux noms des utilisateurs du système de gestion de bases de données. Un démon fournissant le service ident est nécessaire.

La méthode `peer` permet d'associer les noms des utilisateurs du système d'exploitation aux noms des utilisateurs du système de gestion de bases de données. Ceci n'est possible qu'avec une connexion locale.

Quant à `pam`, il authentifie l'utilisateur en passant par les `Pluggable Authentication Modules` (PAM) fournis par le système d'exploitation.

Le pg_hba.conf par défaut

Voici les entrées par défaut du fichier `pg_hba.conf`
en version 14 sur Debian :

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             postgres                                 peer
local   all             all                                      peer
host    all           all           127.0.0.1/32          scram-sha-256
host    all            all           ::1/128              scram-sha-256
local   replication     all                                      peer
host    replication    all           127.0.0.1/32          scram-sha-256
host    replication     all           ::1/128              scram-sha-256
```

Droits sur les objets

- Droits sur les objets
- Droits sur les méta-données
- Héritage des droits

À l'installation de PostgreSQL, il est essentiel de s'assurer de la sécurité du serveur : sécurité au niveau des accès, au niveau des objets, ainsi qu'au niveau des données.

Ce chapitre va faire le point sur ce qu'un utilisateur peut faire par défaut et sur ce qu'il ne peut pas faire. Nous verrons ensuite comment restreindre les droits.

Pour bien comprendre l'intérêt des utilisateurs, il faut bien comprendre la gestion des droits. Les droits sur les objets vont permettre aux utilisateurs de créer des objets ou de les utiliser. Les commandes GRANT et REVOKE sont essentielles pour cela. Modifier la définition d'un objet demande un autre type de droit, que les commandes précédentes ne permettent pas d'obtenir.

Donner des droits à chaque utilisateur peut paraître long et difficile. C'est pour cela qu'il est généralement préférable de donner des droits à une entité spécifique dont certains utilisateurs hériteront.

Agir sur les droits

- Donner un droit : GRANT
- Retirer un droit : REVOKE
- Droits spécifiques pour chaque type d'objets
- Avoir le droit de donner le droit : `WITH GRANT OPTION`

Par défaut, seul le propriétaire a des droits sur son objet. Les superutilisateurs n'ont pas de droit spécifique sur les objets mais étant donné leur statut de superutilisateur, ils peuvent tout faire sur tous les objets.

Le propriétaire d'un objet peut décider de donner certains droits sur cet objet à certains rôles. Il le fera avec la commande GRANT :

```
GRANT droits ON type_objet nom_objet TO role
```

Les droits disponibles dépendent du type d'objet visé. Par exemple, il est possible de donner le droit SELECT sur une table mais pas sur une fonction. Une fonction ne se lit pas, elle s'exécute. Il est donc possible de donner le droit EXECUTE sur une fonction. Il faut donner les droits aux différents objets séparément. De plus, donner le droit ALL sur une base de données donne tous les droits sur la base de

données, autrement dit l'objet base de donnée, pas sur les objets à l'intérieur de la base de données. GRANT n'est pas une commande récursive. Prenons un exemple :

```
b1=# CREATE ROLE u20 LOGIN;
CREATE ROLE
b1=# CREATE ROLE u21 LOGIN;
CREATE ROLE
b1=# \c b1 u20
You are now connected to database "b1" as user "u20".
b1=> CREATE SCHEMA s1;
ERROR:  permission denied for database b1
b1=> \c b1 postgres
You are now connected to database "b1" as user "postgres".
b1=# GRANT CREATE ON DATABASE b1 TO u20;
GRANT
b1=# \c b1 u20
You are now connected to database "b1" as user "u20".
b1=> CREATE SCHEMA s1;
CREATE SCHEMA
b1=> CREATE TABLE s1.t1 (c1 integer);
CREATE TABLE
b1=> INSERT INTO s1.t1 VALUES (1), (2);
INSERT 0 2
b1=> SELECT * FROM s1.t1;
 c1
----
  1
  2
(2 rows)

b1=> \c b1 u21
You are now connected to database "b1" as user "u21".
b1=> SELECT * FROM s1.t1;
ERROR:  permission denied for schema s1
LINE 1: SELECT * FROM s1.t1;
                      ^
b1=> \c b1 u20
You are now connected to database "b1" as user "u20".
```

```
b1=> GRANT SELECT ON TABLE s1.t1 TO u21;
GRANT
b1=> \c b1 u21
You are now connected to database "b1" as user "u21".
b1=> SELECT * FROM s1.t1;
ERROR:  permission denied for schema s1
LINE 1: SELECT * FROM s1.t1;
                      ^
b1=> \c b1 u20
You are now connected to database "b1" as user "u20".
b1=> GRANT USAGE ON SCHEMA s1 TO u21;
GRANT
b1=> \c b1 u21
You are now connected to database "b1" as user "u21".
b1=> SELECT * FROM s1.t1;
 c1
----
  1
  2
(2 rows)

b1=> INSERT INTO s1.t1 VALUES (3);
ERROR:  permission denied for relation t1
```

Le problème de ce fonctionnement est qu'il faut indiquer les droits pour chaque utilisateur, ce qui peut devenir difficile et long. Imaginez avoir à donner le droit SELECT sur les 400 tables d'un schéma… long et fastidieux… Il est néanmoins possible de donner les droits sur tous les objets d'un certain type dans un schéma. Voici un exemple :

```
GRANT SELECT ON ALL TABLES IN SCHEMA s1 to u21;
```

Notez aussi que, lors de la création d'une base, PostgreSQL ajoute automatiquement un schéma nommé public. Tous les droits sont donnés sur ce schéma à un pseudo-rôle appelé public. Tous les rôles sont automatiquement membres de ce pseudo-rôle. Si vous voulez gérer complètement les droits sur ce schéma, il faut d'abord penser à enlever les droits pour ce pseudo-rôle. Pour cela, il vous faut utiliser la commande REVOKE ainsi :

```
REVOKE ALL ON SCHEMA public FROM public;
```

Il est possible d'ajouter des droits pour des objets qui n'ont pas encore été créés. En fait, la commande ALTER DEFAULT PRIVILEGES permet de donner des droits par défaut à certains rôles. De cette

façon, sur un schéma qui a tendance à changer fréquemment, il n'est plus nécessaire de se préoccuper des droits sur les objets.

Lorsqu'un droit est donné à un rôle, par défaut, ce rôle ne peut pas le donner à un autre. Pour lui donner en plus le droit de donner ce droit à un autre rôle, il faut utiliser la clause `WITH GRANT OPTION` comme le montre cet exemple :

```
b1=# CREATE TABLE t2 (id integer);
CREATE TABLE
b1=# INSERT INTO t2 VALUES (1);
INSERT 0 1
b1=# SELECT * FROM t2;
 id
----
  1
(1 row)

b1=# \c b1 u1
You are now connected to database "b1" as user "u1".
b1=> SELECT * FROM t2;
ERROR:  permission denied for relation t2
b1=> \c b1 postgres
You are now connected to database "b1" as user "postgres".
b1=# GRANT SELECT ON TABLE t2 TO u1;
GRANT
b1=# \c b1 u1
You are now connected to database "b1" as user "u1".
b1=> SELECT * FROM t2;
 id
----
  1
(1 row)

b1=> \c b1 u2
You are now connected to database "b1" as user "u2".
b1=> SELECT * FROM t2;
ERROR:  permission denied for relation t2
b1=> \c b1 u1
You are now connected to database "b1" as user "u1".
```

```
b1=> GRANT SELECT ON TABLE t2 TO u2;
WARNING:  no privileges were granted for "t2"
GRANT
b1=> \c b1 postgres
You are now connected to database "b1" as user "postgres".
b1=# GRANT SELECT ON TABLE t2 TO u1 WITH GRANT OPTION;
GRANT
b1=# \c b1 u1
You are now connected to database "b1" as user "u1".
b1=> GRANT SELECT ON TABLE t2 TO u2;
GRANT
b1=> \c b1 u2
You are now connected to database "b1" as user "u2".
b1=> SELECT * FROM t2;
 id
----
  1
(1 row)
```

Droits sur les métadonnées

- Seul le propriétaire peut changer la structure d'un objet
  - le renommer
  - le changer de schéma ou de tablespace
  - lui ajouter/retirer des colonnes
- Un seul propriétaire
  - mais qui peut être un groupe

Les droits sur les objets ne concernent pas le changement des méta-données et de la structure de l'objet. Seul le propriétaire (et les superutilisateurs) peut le faire. S'il est nécessaire que plusieurs personnes puissent utiliser la commande `ALTER` sur l'objet, il faut que ces différentes personnes aient un rôle qui soit membre du rôle propriétaire de l'objet. Prenons un exemple :

```
b1=# \c b1 u21
You are now connected to database "b1" as user "u21".
```

```
b1=> ALTER TABLE s1.t1 ADD COLUMN c2 text;
ERROR:  must be owner of relation t1
b1=> \c b1 u20
You are now connected to database "b1" as user "u20".
b1=> GRANT u20 TO u21;
GRANT ROLE
b1=> \du
                                   List of roles
 Role name  |                         Attributes          |      Member of
------------+-----------------------------------------------+---------
-----------
 admin      |                                               | {}
 caviste    |                                               | {}
 postgres   | Superuser, Create role, Create DB,           +| {}
            | Replication, Bypass RLS                       |
 stagiaire2 |                                               | {}
 u1         |                                    | {pg_signal_backend}
 u11        |                                               | {}
 u2         | Create DB                                     | {}
 u20        |                                               | {}
 u21        |                                               | {u20}
 u3         | Cannot login                                  | {}
 u4         | Create role, Cannot login                     | {}
 u5         |                                               | {u2,u6}
 u6         | Cannot login                                  | {}
 u7         | Create role                                   | {}
 u8         |                                               | {}
 u9         | Create DB                                     | {}

b1=> \c b1 u21
You are now connected to database "b1" as user "u21".
b1=> ALTER TABLE s1.t1 ADD COLUMN c2 text;
ALTER TABLE
```

Pour assigner un propriétaire différent aux objets ayant un certain propriétaire, il est possible de faire appel à l'ordre `REASSIGN OWNED`. De même, il est possible de supprimer tous les objets appartenant à un utilisateur avec l'ordre `DROP OWNED`. Voici un exemple de ces deux commandes :

```
b1=# \d
         List of relations
 Schema |   Name   | Type  | Owner
--------+----------+-------+-------
 public | clients  | table | r31
 public | factures | table | r31
 public | t1       | table | r31
 public | t2       | table | r32
(4 rows)

b1=# REASSIGN OWNED BY r31 TO u1;
REASSIGN OWNED
b1=# \d
         List of relations
 Schema |   Name   | Type  | Owner
--------+----------+-------+-------
 public | clients  | table | u1
 public | factures | table | u1
 public | t1       | table | u1
 public | t2       | table | r32
(4 rows)

b1=# DROP OWNED BY u1;
DROP OWNED
b1=# \d
       List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
 public | t2   | table | r32
(1 row)
```

Héritage des droits

- Créer un rôle sans droit de connexion
- Donner les droits à ce rôle
- Placer les utilisateurs concernés comme membre de ce rôle

Plutôt que d'avoir à donner les droits sur chaque objet à chaque ajout d'un rôle, il est beaucoup plus simple d'utiliser le système d'héritage des droits.

Supposons qu'une nouvelle personne arrive dans le service de facturation. Elle doit avoir accès à toutes les tables concernant ce service. Sans utiliser l'héritage, il faudra récupérer les droits d'une autre personne du service pour retrouver la liste des droits à donner à cette nouvelle personne. De plus, si un nouvel objet est créé et que chaque personne du service doit pouvoir y accéder, il faudra ajouter l'objet et ajouter les droits pour chaque personne du service sur cet objet. C'est long et sujet à erreur. Il est préférable de créer un rôle facturation, de donner les droits sur ce rôle, puis d'ajouter chaque rôle du service facturation comme membre du rôle facturation. L'ajout et la suppression d'un objet est très simple : il suffit d'ajouter ou de retirer le droit sur le rôle facturation, et cela impactera tous les rôles membres.

Voici un exemple complet :

```
b1=# CREATE ROLE facturation;
CREATE ROLE
b1=# CREATE TABLE factures(id integer, dcreation date, libelle text,
montant numeric);
CREATE TABLE
b1=# GRANT ALL ON TABLE factures TO facturation;
GRANT
b1=# CREATE TABLE clients (id integer, nom text);
CREATE TABLE
b1=# GRANT ALL ON TABLE clients TO facturation;
GRANT
b1=# CREATE ROLE r1 LOGIN;
CREATE ROLE
b1=# GRANT facturation TO r1;
GRANT ROLE
b1=# \c b1 r1
You are now connected to database "b1" as user "r1".
b1=> SELECT * FROM factures;
 id | dcreation | libelle | montant
----+-----------+---------+---------
(0 rows)

b1=# CREATE ROLE r2 LOGIN;
CREATE ROLE
```

```
b1=# \c b1 r2
You are now connected to database "b1" as user "r2".
b1=> SELECT * FROM factures;
ERROR:  permission denied for relation factures
```

Droits par défaut

- Un utilisateur standard peut
  - accéder à toutes les bases de données
  - créer des objets dans le schéma PUBLIC de toute base de données
  - créer des objets temporaires
  - modifier les paramètres de la session
  - créer des fonctions
  - exécuter des fonctions définies par d'autres dans le schéma PUBLIC
  - récupérer des informations sur l'instance
  - visualiser le source des vues et des fonctions

Par défaut, un utilisateur a beaucoup de droits.

Il peut accéder à toutes les bases de données. Il faut modifier le fichier `pg_hba.conf` pour éviter cela. Il est aussi possible de supprimer ce droit avec l'ordre suivant :

```
REVOKE CONNECT ON DATABASE nom_base FROM nom_utilisateur;
```

Il peut créer des objets dans le schéma disponible par défaut (nommé `public`) sur chacune des bases de données où il peut se connecter. Il est possible de supprimer ce droit avec l'ordre suivant :

```
REVOKE CREATE ON SCHEMA public FROM nom_utilisateur;
```

Il peut créer des objets temporaires sur chacune des bases de données où il peut se connecter. Il est possible de supprimer ce droit avec l'ordre suivant :

```
REVOKE TEMP ON DATABASE nom_base FROM nom_utilisateur;
```

Il peut modifier les paramètres de session (par exemple le paramètre `work_mem` pour s'allouer beaucoup plus de mémoire). Il est impossible d'empêcher cela.

Il peut créer des fonctions, uniquement avec les langages de confiance, uniquement dans les schémas où il a le droit de créer des objets. Il existe deux solutions :

- supprimer le droit d'utiliser un langage

```
REVOKE USAGE ON LANGUAGE nom_langage FROM nom_utilisateur;
```

- supprimer le droit de créer des objets dans un schéma

```
REVOKE CREATE ON SCHEMA nom_schema FROM nom_utilisateur;
```

Il peut exécuter toute fonction, y compris définie par d'autres, à condition qu'elles soient créées dans des schémas où il a accès. Il est possible d'empêcher cela en supprimant le droit d'exécution d'une fonction:

```
REVOKE EXECUTE ON FUNCTION nom_fonction FROM nom_utilisateur;
```

Il peut récupérer des informations sur l'instance car il a le droit de lire tous les catalogues systèmes. Par exemple, en lisant pg_class, il peut connaitre la liste des tables, vues, séquences, etc. En parcourant pg_proc, il dispose de la liste des fonctions. Il n'y a pas de contournement à cela : un utilisateur doit pouvoir accéder aux catalogues systèmes pour travailler normalement.

Enfin, il peut visualiser le source des vues et des fonctions. Il existe des modules propriétaires de chiffrement (ou plutôt d'obfuscation) du code mais rien de natif. Le plus simple est certainement de coder les fonctions sensibles en C.

Par défaut (suite)

- Un utilisateur standard ne peut pas
  - créer une base
  - créer un rôle
  - accéder au contenu des objets créés par d'autres
  - modifier le contenu d'objets créés par d'autres

Un utilisateur standard ne peut pas créer de bases et de rôles. Il a besoin pour cela d'attributs particuliers (respectivement CREATEDB et CREATEROLE).

Il ne peut pas accéder au contenu (aux données) d'objets créés par d'autres utilisateurs. Ces derniers doivent lui donner ce droit explicitement. De même, il ne peut pas modifier le contenu et la définition d'objets créés par d'autres utilisateurs. Là-aussi, ce droit doit être donné explicitement.

La sauvegarde / restauration en base de données

- Opération essentielle pour la sécurisation des données
- PostgreSQL propose différentes solutions
  - de sauvegarde à froid ou à chaud, mais cohérentes
  - des méthodes de restauration partielle ou complète

La mise en place d'une solution de sauvegarde est une des opérations les plus importantes après avoir installé un serveur PostgreSQL. En effet, nul n'est à l'abri d'un bug logiciel, d'une panne matérielle, voire d'une erreur humaine.

Cette opération est néanmoins plus complexe qu'une sauvegarde standard car elle doit pouvoir s'adapter aux besoins des utilisateurs. Quand le serveur ne peut jamais être arrêté, la sauvegarde à froid des fichiers ne peut convenir. Il faudra passer dans ce cas par un outil qui pourra sauvegarder les données alors que les utilisateurs travaillent et qui devra respecter les contraintes ACID pour fournir une sauvegarde cohérente des données.

PostgreSQL va donc proposer des méthodes de sauvegardes à froid (autrement dit serveur arrêté) comme à chaud, mais de toute façon cohérente. Les sauvegardes pourront être partielles ou complètes, suivant le besoin des utilisateurs.

La méthode de sauvegarde dictera l'outil de restauration. Suivant l'outil, il fonctionnera à froid ou à chaud, et permettra même dans certains cas de faire une restauration partielle.

Plan

- Politique de sauvegarde
- Sauvegarde logique
- Sauvegarde physique

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.

Objectifs

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

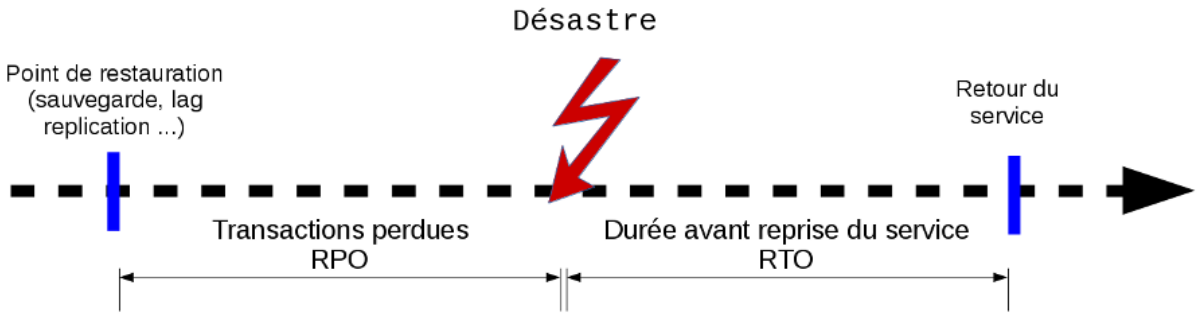
L'objectif essentiel de la sauvegarde est la sécurisation des données. Autrement dit, l'utilisateur cherche à se protéger d'une panne matérielle ou d'une erreur humaine (un utilisateur qui supprimerait des données essentielles). La sauvegarde permet de restaurer les données perdues. Mais ce n 'est pas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

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.

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.

Sauvegardes logiques

- À chaud
- Cohérente
- Locale ou à distance
- 2 outils
  - `pg_dump`
  - `pg_dumpall`

La sauvegarde logique nécessite que le serveur soit en cours d'exécution. Un outil se connecte à la base et récupère la déclaration des différents objets ainsi que les données des tables.

Comme ce type d'outil n'a besoin que d'une connexion standard à la base de données, il peut se connecter en local comme à distance. Cela implique qu'il doive aussi respecter les autorisations de connexion configurées dans le fichier `pg_hba.conf`.

Il existe deux outils de ce type pour la sauvegarde logique dans la distribution officielle de PostgreSQL :

- `pg_dump`, pour sauvegarder une base (complètement ou partiellement) ;
- `pg_dumpall` pour sauvegarder toutes les bases ainsi que les objets globaux.

---

pg_dump

- Sauvegarde une base de données
- Sauvegarde complète ou partielle
- Format de sortie :

  - texte SQL : `plain`
  - binaire : `tar`, `custom` ou `directory`

pg_dump est l'outil le plus utilisé pour sauvegarder une base de données PostgreSQL. Une sauvegarde peut se faire de façon très simple. Par exemple :

```
postgres$ pg_dump b1 > b1.dump
```

sauvegardera la base b1 de l'instance locale sur le port 5432 dans un fichier b1.dump.

Mais pg_dump permet d'aller bien plus loin que la sauvegarde d'une base de données complète. Il existe pour cela de nombreuses options en ligne de commande.

---

pg_dumpall

- Sauvegarde d'une instance complète
  - Objets globaux
  - Bases de données
- Toujours en format texte

pg_dump sauvegarde toute la structure et toutes les données locales à une base de données. Cette commande ne sauvegarde pas la définition des objets globaux, comme par exemple les utilisateurs et les tablespaces.

De plus, il peut être intéressant d'avoir une commande capable de sauvegarder toutes les bases de l'instance. Reconstruire l'instance est beaucoup plus simple car il suffit de rejouer ce seul fichier de sauvegarde.

Restauration d'une sauvegarde logique

- Sauvegarde texte (option **p**) : *psql*
- Sauvegarde binaire (options **t**, **c** ou **d**) : *pg_restore*

pg_dump permet de réaliser deux types de sauvegarde : une sauvegarde texte (via le format plain) et une sauvegarde binaire (via les formats tar, personnalisé et répertoire).

Chaque type de sauvegarde aura son outil :

- `psql` pour les sauvegardes textes ;
- `pg_restore` pour les sauvegardes binaires.

---

psql

- client standard PostgreSQL
- capable d'exécuter des requêtes
- donc capable de restaurer une sauvegarde au format texte
- très limité dans les options de restauration

`psql` est la console interactive de PostgreSQL. Elle permet de se connecter à une base de données et d'y exécuter des requêtes, soit une par une, soit un script complet. Or, la sauvegarde texte de `pg_dump` et de `pg_dumpall` fournit un script SQL. Ce dernier est exécutable via `psql`.

---

pg_restore

- restaure uniquement les sauvegardes au format binaire
  - donc tar, custom ou directory
  - format autodétecté (`-F` inutile, même si présent)
- nombreuses options très intéressantes
- restaure une base de données
  - complètement ou partiellement

`pg_restore` est un outil capable de restaurer les sauvegardes au format binaire, quel qu'en soit le format. Il offre de nombreuses options très intéressantes, la plus essentielle étant de permettre une restauration partielle de façon aisée.

L'exemple typique d'utilisation de `pg_restore` est le suivant :

```
pg_restore -d b1 b1.dump
```

La base de données où la sauvegarde va être restaurée est indiquée avec l'option `-d` et le nom du fichier de sauvegarde est le dernier argument dans la ligne de commande.

Sauvegarde au niveau système de fichiers

- À froid
- Donc cohérente
- Beaucoup d'outils
  - aucun spécifique à PostgreSQL
- Attention à ne pas oublier les tablespaces

Toutes les données d'une instance PostgreSQL se trouvent dans des fichiers. Donc sauvegarder les fichiers permet de sauvegarder une instance. Cependant, cela ne peut pas se faire aussi simplement que ça. Lorsque PostgreSQL est en cours d'exécution, il modifie certains fichiers du fait de l'activité des utilisateurs ou des processus (interne ou non) de maintenances diverses. Sauvegarder les fichiers de la base sans plus de manipulation ne peut donc se faire qu'à froid. Il faut arrêter PostgreSQL pour disposer d'une sauvegarde cohérente si la sauvegarde se fait au niveau du système de fichiers.

Le gros avantage de cette sauvegarde se trouve dans le fait que vous pouvez utiliser tout outil de sauvegarde de fichier : `cp`, `scp`, `tar`, `ftp`, `rsync`, etc.

Il est cependant essentiel d'être attentif aux données qui ne se trouvent pas directement dans le répertoire des données. Notamment le répertoire des journaux de transactions, qui est souvent placé dans un autre système de fichiers pour gagner en performances. Si c'est le cas et que ce répertoire n'est pas sauvegardé, la sauvegarde ne sera pas utilisable. De même, si des tablespaces sont créés, il est essentiel d'intégrer ces autres répertoires dans la sauvegarde de fichiers. Dans le cas contraire, une partie des données manquera.

Voici un exemple de sauvegarde :

```
$ /etc/init.d/postgresql stop
$ tar cvfj data.tar.bz2 /var/lib/postgresql
$ /etc/init.d/postgresql start
```

Il est possible de réaliser les sauvegardes de fichiers sans arrêter l'instance (*à chaud*), mais il s'agit d'une technique avancée (dite *PITR*, ou *Point In Time Recovery*), qui nécessite la compréhension de concepts non abordés dans le cadre de cette formation, comme l'archivage des fichiers WAL.

Avantages

- Rapide à la sauvegarde

- Rapide à la restauration
- Beaucoup d'outils disponibles

L'avantage de ce type de sauvegarde est sa rapidité. Cela se voit essentiellement à la restauration où les fichiers ont seulement besoin d'être créés. Les index ne sont pas recalculés par exemple, ce qui est certainement le plus long dans la restauration d'une sauvegarde logique.

Inconvénients

- Arrêt de la production
- Sauvegarde de l'instance complète (donc aucune granularité)
- Restauration de l'instance complète
- Conservation de la fragmentation
- Impossible de changer d'architecture

Il existe aussi de nombreux inconvénients à cette méthode.

Le plus important est certainement le fait qu'il faut arrêter la production. L'instance PostgreSQL doit être arrêtée pour que la sauvegarde puisse être effectuée.

Il ne sera pas possible de réaliser une sauvegarde ou une restauration partielle, il n'y a pas de granularité. C'est forcément l'intégralité de l'instance qui sera prise en compte.

Étant donné que les fichiers sont sauvegardés, toute la fragmentation des tables et des index est conservée.

De plus, la structure interne des fichiers implique l'architecture où cette sauvegarde sera restaurée. Donc une telle sauvegarde impose de conserver un serveur 32 bits pour la restauration si la sauvegarde a été effectuée sur un serveur 32 bits. De même, l'architecture LittleEndian/BigEndian doit être respectée.

Tous ces inconvénients ne sont pas présents pour la sauvegarde logique. Cependant, cette sauvegarde a aussi ses propres inconvénients, comme une lenteur importante à la restauration.

Diminuer l'immobilisation

- Utilisation de rsync

- Une fois avant l'arrêt
- Une fois après

Il est possible de diminuer l'immobilisation d'une sauvegarde de fichiers en utilisant la commande `rsync`.

`rsync` permet de synchroniser des fichiers entre deux répertoires, en local ou à distance. Il va comparer les fichiers pour ne transférer que ceux qui ont été modifiés. Il est donc possible d'exécuter `rsync` avec PostgreSQL en cours d'exécution pour récupérer un maximum de données, puis d'arrêter PostgreSQL, de relancer `rsync` pour ne récupérer que les données modifiées entre temps, et enfin de relancer PostgreSQL. Voici un exemple de ce cas d'utilisation :

```
$ rsync /var/lib/postgresql /var/lib/postgresql2
$ /etc/init.d/postgresql stop
$ rsync -av /var/lib/postgresql /var/lib/postgresql2
$ /etc/init.d/postgresql start
```

Snapshot de partition

- Avec certains systèmes de fichiers
- Avec LVM
- Avec la majorité des SAN

Certains systèmes de fichiers (principalement ZFS et le système de fichiers en cours de développement BTRFS) ainsi que la majorité des SAN sont capables de faire une sauvegarde d'un système de fichiers en instantané. En fait, ils figent les blocs utiles à la sauvegarde. S'il est nécessaire de modifier un bloc figé, ils utilisent un autre bloc pour stocker la nouvelle valeur. Cela revient un peu au fonctionnement de PostgreSQL dans ses fichiers.

L'avantage est de pouvoir sauvegarder instantanément un système de fichiers. L'inconvénient est que cela ne peut survenir que sur un seul système de fichiers : impossible dans ce cas de déplacer les journaux de transactions sur un autre système de fichiers pour gagner en performance ou d'utiliser des tablespaces pour gagner en performance et faciliter la gestion de la volumétrie des disques. De plus, comme PostgreSQL n'est pas arrêté au moment de la sauvegarde, au démarrage de PostgreSQL sur la sauvegarde restaurée, ce dernier devra rejouer les journaux de transactions.

Une baie SAN assez haut de gamme pourra disposer d'une fonctionnalité de snapshot cohérent sur plusieurs volumes (« LUN »), ce qui permettra, si elle est bien paramétrée, de réaliser un snapshot de tous les systèmes de fichiers composant la base de façon cohérente.

Néanmoins, cela reste une méthode de sauvegarde très appréciable quand on veut qu'elle ait le moins d'impact possible sur les utilisateurs.

PITR

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

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

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

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

Principes

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

Quand une transaction est validée, les données à écrire dans les fichiers de données sont d'abord écrites dans un journal de transactions. Ces journaux décrivent donc toutes les modifications survenant sur les fichiers de données, que ce soit les objets utilisateurs comme les objets systèmes. Pour reconstruire un système, il suffit donc d'avoir ces journaux et d'avoir un état des fichiers du répertoire des données à un instant t. Toutes les actions effectuées après cet instant t pourront être rejouées en

demandant à PostgreSQL d'appliquer les actions contenues dans les journaux. Les opérations stockées dans les journaux correspondent à des modifications physiques de fichiers, il faut donc partir d'une sauvegarde au niveau du système de fichier, un export avec pg_dump n'est pas utilisable.

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

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

Avantages

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

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

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

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

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

Inconvénients

- Sauvegarde de l'instance complète
- Nécessite un grand espace de stockage (données + journaux)
- Risque d'accumulation des journaux en cas d'échec d'archivage
- Restauration de l'instance complète
- Impossible de changer d'architecture
- Plus complexe

Certains inconvénients viennent directement du fait qu'on copie les fichiers : sauvegarde et restauration complète (impossible de ne restaurer qu'une seule base ou que quelques tables), restauration sur la même architecture (32/64 bits, *little/big endian*), voire probablement le même système d'exploitation.

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

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

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

Mise en place de la sauvegarde PITR

- 2 étapes :
  - Archivage des journaux de transactions
  - Sauvegarde des fichiers de données

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

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

Outils de sauvegarde PITR

- Nombreux outils existants
- Le plus avancé : pgBackRest

Les outils permettent de faciliter la mise en place de sauvegardes PITR. Ils gèrent à la fois la sauvegarde et la restauration.

pgBackRest[1] est écrit en C et perl, par David Steele de Crunchy Data. Logiciel très complet, il permet de réaliser des sauvegardes incrémentales avec une granularité au niveau du bloc. Il permet à la fois la sauvegarde et la restauration.

Il s'agit d'un logiciel à part entière. Sa prise en main et sa mise en oeuvre demande un certain investissement.

Matrice

	Simplicité	Coupure	Restauration	Fragmentation
copie à froid	facile	longue	rapide	conservée
snapshot FS	facile	aucune	rapide	conservée
pg_dump	facile	aucune	lente	perdue
rsync + copie à froid	moyen	courte	rapide	conservée
PITR	difficile	aucune	rapide	conservée

Ce tableau indique les points importants de chaque type de sauvegarde. Il permet de faciliter un choix entre les différentes méthodes.

---

[1] https://pgbackrest.org/

Travaux Dirigés 3

- Les outils de sauvegardes et restaurations
  - *pg_dump* et *pg_dumpall*
  - *psql* et *pg_restore*

Questions de cours

- Quel fichier permet de configurer les droits d'accès à l'instance ?
- Y a-t-il un lien entre utilisateur système et rôle PostgreSQL ?
- Quel ordre SQL permet de donner des droits sur une table ?

Options de connexions des outils

- `-h` / `$PGHOST` / socket Unix
- `-p` / `$PGPORT` / 5432
- `-U` / `$PGUSER` / utilisateur du système
- `$PGPASSWORD`
- `.pgpass`

Les commandes `pg_dump` et `pg_dumpall` 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 ;
- `-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).

À partir de la version 10, il est possible d'indiquer plusieurs hôtes et ports. L'hôte sélectionné est le premier qui répond au paquet de démarrage. Si l'authentication ne passe pas, la connexion sera en erreur. Il est aussi possible de préciser si la connexion doit se faire sur un serveur en lecture/écriture ou en lecture seule.

Par exemple on effectuera une sauvegarde depuis le premier serveur disponible ainsi :

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

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

---

pg_dump

L'outil de sauvegarde logique d'une base de données

---

pg_dump - Format de sortie

- `-F`
  - p : plain, SQL
  - `t` : tar
  - c : custom (spécifique PostgreSQL)
  - d : directory

`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.

---

pg_dump - Fichier ou sortie standard

- `-f` : fichier où est stockée la sauvegarde
- sans `-f`, sur la sortie standard

Par défaut, et en dehors du format répertoire, toutes les données d'une sauvegarde sont renvoyées sur la sortie standard de `pg_dump`. Il faut donc utiliser une redirection pour renvoyer dans un fichier.

Cependant, il est aussi possible d'utiliser l'option `-f` pour spécifier le fichier de la sauvegarde. L'utilisation de cette option est conseillée car elle permet à `pg_restore` de trouver plus efficacement les objets à restaurer dans le cadre d'une restauration partielle.

---

pg_dump - Sélection de sections

- `--section`
  - `pre-data`, la définition des objets (hors contraintes et index)
  - `data`, les données
  - `post-data`, la définition des contraintes et index

Historiquement, il est possible de sauvegarder que la structure avec l'option `--schema-only` ou uniquement les données avec l'option `--data-only`. Cependant, lors de la restauration, les index et contraintes seront créées en même temps que le schéma. Lors de l'insertion des données, les vérifications de contraintes et ajout de clé d'index seront faites à l'insertion de chaque ligne.

Il est possible de sauvegarder une base par section. En fait, un fichier de sauvegarde complet est composé de trois sections :

- la définition des objets,
- les données,
- la définition des contraintes et index.

Le fait de séparer ainsi les sauvegardes permet de gagner en performance à la vérification des contraintes et à la création d'index.

---

pg_dump - Sélection d'objets

- `-n <schema>` : uniquement ce schéma
- `-N <schema>` : tous les schémas sauf celui-là

- `-t <table>` : uniquement cette table
- `-T <table>` : toutes les tables sauf celle-là
- En option
  - possibilité d'en mettre plusieurs
  - exclure les données avec `--exclude-table-data=<table>`
  - avoir une erreur si l'objet est inconnu avec`--strict-names`

En dehors de la distinction structure/données, il est possible de demander de ne sauvegarder qu'un objet. Les seuls objets sélectionnables au niveau de `pg_dump` sont les tables et les schémas. L'option `-n` permet de sauvegarder seulement le schéma cité après alors que l'option `-N` permet de sauvegarder tous les schémas sauf celui cité après l'option. Le même système existe pour les tables avec les options `-t` et `-T`. Il est possible de mettre ces options plusieurs fois pour sauvegarder plusieurs tables spécifiques ou plusieurs schémas.

Les équivalents longs de ces options sont: `--schema`, `--exclude-schema`, `--table` et `--exclude-table`.

Par défaut, si certains objets sont trouvés et d'autres non, `pg_dump` ne dit rien, et l'opération se termine avec succès. Ajouter l'option `--strict-names` permet de s'assurer d'être averti avec une erreur sur le fait que `pg_dump` n'a pas sauvegardé tous les objets souhaités. En voici un exemple (`t1` existe, `t20` n'existe pas) :

```
$ pg_dump -t t1 -t t20 -f postgres.dump postgres
$ echo $?
0
$ pg_dump -t t1 -t t20 --strict-names -f postgres.dump postgres
pg_dump: no matching tables were found for pattern "t20"
$ echo $?
1
```

---

pg_dump - Option divers

- `-j <nombre_de_threads>` : en format *directory*
- `-Z` : option de compression, de 0 à 9
- `-O` : ignorer le propriétaire
- `-x` : ignorer les droits
- `-v` : pour voir la progression

Historiquement, pg_dump n'utilise qu'une seule connexion à la base de données pour sauvegarder la définition des objets et les données. Cependant, une fois que la première étape de récupération de la définition des objets est réalisée, l'étape de sauvegarde des données peut être parallélisée pour profiter des nombreux processeurs disponibles sur un serveur.
L'option `-j` permet de préciser le nombre de connexions réalisées vers la base de données. Chaque connexion est gérée dans pg_dump par un processus sous Unix et par un thread sous Windows. Par contre, cette option est compatible uniquement avec le format de sortie *directory* (option `-F d`).
Cela permet d'améliorer considérablement la vitesse de sauvegarde.

Il est possible de compresser la sauvegarde. La compression va de 0, pas de compression, à 9, compression maximum.
Par défaut, pg_dump utilise le niveau de compression par défaut de la libz (`Z_DEFAULT_COMPRESSION`) qui correspond au meilleur compromis entre compression et vitesse, équivalent au niveau 6.

Les options `--no-owner` et `--no-privileges` permettent de ne pas indiquer respectivement le propriétaire et les droits de l'objet dans la sauvegarde.

Enfin, l'option `-v` (ou `--verbose`) permet de voir la progression de la commande.

---

pg_dumpall

L'outil de sauvegarde logique d'une instance complète

---

pg_dumpall - Format de sortie

- `-F`
  - p : plain, SQL

Contrairement à pg_dump, `pg_dumpall` ne dispose que d'un format en sortie : le fichier SQL. L'option est donc inutile.

---

pg_dumpall - Fichier ou sortie standard

- `-f` : fichier où est stockée la sauvegarde
- sans `-f`, sur la sortie standard

La sauvegarde est automatiquement envoyée sur la sortie standard, sauf si la ligne de commande précise l'option `-f` (ou `--file`) et le nom du fichier.

---

pg_dumpall - Sélection des objets

- `-g` : tous les objets globaux
- `-r` : uniquement les rôles
- `-t` : uniquement les tablespaces
- `--no-role-passwords` : pour ne pas sauvegarder les mots de passe
  - permet de ne pas être superutilisateur

`pg_dumpall` étant créé pour sauvegarder l'instance complète, il disposera de moins d'options de sélection d'objets. Néanmoins, il permet de ne sauvegarder que la déclaration des objets globaux, ou des rôles, ou des tablespaces. Leur versions longues sont respectivement: `--globals-only`, `--roles-only` et `--tablespaces-only`.

Restauration d'une sauvegarde logique

- Sauvegarde texte (option **p**) : *psql*
- Sauvegarde binaire (options **t**, **c** ou **d**) : *pg_restore*

---

psql - Options

- `-f` pour indiquer le fichier contenant la sauvegarde
  - sans option `-f`, lit l'entrée standard
- `-1` pour tout restaurer en une seule transaction
- `-e` pour afficher les ordres SQL exécutés

- ON_ERROR_ROLLBACK/ON_ERROR_STOP

Pour cela, il existe plusieurs moyens :

- envoyer le script sur l'entrée standard de psql :

```
cat b1.dump | psql b1
```

- utiliser l'option en ligne de commande -f :

```
psql -f b1.dump b1
```

- utiliser la méta-commande \i :

```
b1 =# \i b1.dump
```

Dans les deux premiers cas, la restauration peut se faire à distance alors que dans le dernier cas, le fichier de la sauvegarde doit se trouver sur le serveur de bases de données.

Le script est exécuté comme tout autre script SQL. Comme il n'y a pas d'instruction `BEGIN` au début, l'échec d'une requête ne va pas empêcher l'exécution de la suite du script, ce qui va généralement apporter un flot d'erreurs. De plus, après une erreur, les requêtes précédentes sont toujours validées. La base de données sera donc dans un état à moitié modifié, ce qui peut poser un problème s'il ne s'agissait pas d'une base vierge. Il est donc préférable parfois d'utiliser l'option en ligne de commande `-1` pour que le script complet soit exécuté dans une seule transaction. Dans ce cas, si une requête échoue, aucune modification n'aura réellement lieu sur la base, et il sera possible de relancer la restauration après correction du problème.

Enfin, il est à noter qu'une restauration partielle de la sauvegarde est assez complexe à faire. Deux solutions possibles, mais pénibles :

- modifier le script SQL dans un éditeur de texte, ce qui peut être impossible si ce fichier est suffisamment gros
- utiliser des outils tels que grep et/ou sed pour extraire les portions voulues, ce qui peut facilement devenir long et complexe

Deux variables psql peuvent être modifiées, ce qui permet d'affiner le comportement de psql lors de l'exécution du script :

- ON_ERROR_ROLLBACK: par défaut (valeur `off`), dans **une transaction**, toute erreur entraîne le ROLLBACK de toute la transaction. Activer ce paramètre permet que seule la commande en erreur soit annulée. psql effectue des *savepoints* avant chaque ordre, et y retourne en cas d'erreur, avant de continuer le script. Ceci est utile surtout si vous avez utilisé l'option `-1`. Il peut valoir `interactive` (ne s'arrêter dans le script qu'en mode interactif, c'est-à-dire quand c'est une commande `\i` qui est lancée) ou on dans quel cas il est actif en permanence.

- ON_ERROR_STOP: par défaut, dans **un script**, une erreur n'arrête pas le déroulement du script. On se retrouve donc souvent avec un ordre en erreur, et beaucoup de mal pour le retrouver, puisqu'il est noyé dans la masse des messages. Quand ON_ERROR_STOP est positionné (à on), le script est interrompu dès qu'une erreur est détectée.

Les variables psql peuvent être modifiées :

- par édition du `.psqlrc` (à déconseiller, cela va modifier le comportement de psql pour toute personne utilisant le compte):

```
cat .psqlrc
\set ON_ERROR_ROLLBACK interactive
```

- en option de ligne de commande de psql :

```
$ psql --set=ON_ERROR_ROLLBACK='on'
```

- de façon interactive dans psql:

```
psql> \set ON_ERROR_ROLLBACK on
```

---

pg_restore - Fichier ou entrée standard

- Fichier à restaurer en dernier argument de la ligne de commande
- **Attention à `-f`** : fichier en sortie

Le fichier à restaurer s'indique en dernier argument sur la ligne de commande.

L'option `-f` permet d'indiquer un fichier qui contiendra un script SQL correspondant aux ordres à générer à la restauration. Il sera donc écrit. S 'il est utilisé pour indiquer le nom du fichier de sauvegarde à restaurer, ce dernier se verra vidé pour contenir les erreurs éventuels de la restauration. Autrement dit, il faut faire extrêmement attention lors de l'utilisation de l'option `-f` avec l'outil `pg_restore`.

Utilisation de l'option `-f`

Il est déconseillé d'utiliser l'option `-f` avec `pg_restore` ! Cette dernière est contre intuitive : elle indique le journal d'activité de `pg_restore` et **NON** le fichier à restaurer. Il est ainsi toujours recommandé d'utiliser simplement la redirection standard qui ne portent jamais à confusion pour récupérer les messages de `pg_restore`.

---

pg_restore - Sélection de sections

- `--section`
  - `pre-data`, la définition des objets (hors contraintes et index)
  - `data`, les données
  - `post-data`, la définition des contraintes et index

Il est possible de restaurer une base section par section. En fait, un fichier de sauvegarde complet est composé de trois sections :

- la définition des objets,
- les données,
- la définition des contraintes et index.

Il est plus intéressant de restaurer par section que de restaurer schéma et données séparément car cela permet d'avoir la partie contrainte et index dans un script à part, ce qui accélère la restauration.

---

pg_restore - Sélection d'objets

- `-n <schema>` : uniquement ce schéma
- `-N <schema>` : tous les schémas sauf ce schéma
- `-t <table>` : cette relation
- `-T <trigger>` : ce trigger
- `-I <index>` : cet index
- `-P <fonction>` : cette fonction
- En option
  - possibilité d'en mettre plusieurs
  - `--strict-names`, pour avoir une erreur si l'objet est inconnu

`pg_restore` fournit quelques options supplémentaires pour sélectionner les objets à restaurer. Il y a les options `-n` et `-t` qui ont la même signification que pour `pg_dump`. `-N` n'existe que depuis la version 10 et a la même signification que pour `pg_dump`. Par contre, `-T` a une signification différente: `-T` précise un trigger dans `pg_restore`.

Il est à noter que l'option `-t` concerne toutes les relations : tables, vues, vues matérialisées et séquences.

Il existe en plus les options `-I` et `-P` (respectivement `--index` et `--function`) pour restaurer respectivement un index et une procédure stockée spécifique.

Là-aussi, il est possible de mettre plusieurs fois les options pour restaurer plusieurs objets de même type ou de type différent.

Par défaut, si le nom de l'objet est inconnu, `pg_restore` ne dit rien, et l'opération se termine avec succès. Ajouter l'option `--strict-names` permet de s'assurer d'être averti avec une erreur sur le fait que `pg_restore` n'a pas restauré l'objet souhaité.

---

pg_restore - Sélection avancée

- `-l` : récupération de la liste des objets
- `-L <liste_objets>` : restauration uniquement des objets listés dans ce fichier

Les options précédentes sont intéressantes quand on a peu de sélection à faire. Par exemple, cela convient quand on veut restaurer deux tables ou quatre index. Quand il faut en restaurer beaucoup plus, cela devient plus difficile. `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.

---

pg_restore - Option divers

- `-j <nombre_de_threads>` : en format *custom* ou *directory*
- `-O` : ignorer le propriétaire
- `-x` : ignorer les droits
- `-1` pour tout restaurer en une seule transaction
- `-c` : pour détruire un objet avant de le restaurer
- `-C` : pour créer la base de donnée avant le rechargement

Historiquement, `pg_restore` n'utilise qu'une seule connexion à la base de données pour y exécuter en série toutes les requêtes nécessaires pour restaurer la base. Cependant, une fois que la première étape de création des objets est réalisée, l'étape de copie des données et celle de création des index peuvent être parallélisées pour profiter des nombreux processeurs disponibles sur un serveur. L'option `-j` permet de préciser le nombre de connexions réalisées vers la base de données. Chaque connexion est gérée dans `pg_restore` par un processus sous Unix et par un thread sous Windows.

Les option `-O` et `-x` permettent de ne pas restaurer respectivement le propriétaire et les droits des objets.

L'option `-1` permet d'exécuter `pg_restore` dans une seule transaction. Attention, ce mode est incompatible avec le mode `-j` car on ne peut pas avoir plusieurs sessions qui partagent la même transaction.

L'option `-c` permet d'exécuter des DROP des objets avant de les restaurer. Ce qui évite les conflits à la restauration de tables par exemple: l'ancienne est détruite avant de restaurer la nouvelle.
L'option `-C` permet de créer la base de données avant de débuter la restauration.
Si les options `-c` et `-C` sont utilisées conjointement, la base de données précisée avec l'option `-d` est uniquement utilisée pour lancer les commandes `DROP DATABASE` et `CREATE DATABASE`.

Enfin, l'option `-v` permet de voir la progression de la commande.

Enoncé

Les outils de sauvegardes et restaurations

Quel est le résultat de chacunes des commandes suivantes ?

Précisez toutes les informations en votre possession.

Outils PostgreSQL

```
admin$ psql -d ojm3 -h 192.168.0.5 -U adb1 -p 5454
```

```
postgres@nevrast$ psql -p 5433 -c 'SELECT pg_reload_conf();' -h /tmp
```

```
postgres$ createdb -O app_user -h 10.51.32.40 -p 5446 my_app
```

```
postgres@localhost$ export PGPORT=5436 && dropdb test
```

Sauvegardes

```
postgres@dbhost$ pg_dump -Fp -d miranda -f /tmp/miranda.sql -h /tmp/v9.6/
```

```
pgadmin@srv$ pg_dump -Fc -U thibaut -t decors -f /home/thibaut/db.dump main
```

```
postgres@doriath$ /usr/pgsql-9.4/bin/pg_dump -Fd -d webapp -j 5 -f
↪  /tmp/ma_db
```

```
jacques@calvin$ pg_dumpall -U tester -h 192.168.10.38 -f /tmp/ma_db.sql -d
 ↪  jon1

jacques@hobbes$ pg_dumpall -g > ~/pg_global.sql
```

Restaurations

```
db=# \i /home/thibaut/oberon.sql
```

```
$ psql -U titania -h 11.56.99.77 -d puck -f /tmp/ma_db.sql

lysandre@athenes$ pg_restore --clean --create -j3 -d hermia
 ↪  reve_de_minuit.dump

postgres@src$ pg_restore -U google -h 8.8.8.8 -l -f e3rsf.list e3rsf

pgadmin@pghost$ pg_restore -L db.list -d sauvegarde -p 5434 ma_db.dump
```

Contrôle des accès Étudier le contenu du fichier pg_hba.conf suivant :

```
TYPE      DATABASE       USER        CIDR-ADDRESS    METHOD
local     all            postgres                    peer
local     web            web                         trust
local     sameuser       all                         peer
host      all            all         127.0.0.1/32    md5
host      all            all         89.192.0.3/8    md5
hostssl   recherche      recherche   89.192.0.4/32   passwd
local     replication    all                         peer
host      replication    all         127.0.0.1/32    md5
host      replication    all         ::1/128         md5
hostssl   replication    repli       89.192.0.0/24   scram-sha-256
```

Solutions du TD 3

Question de cours

- Quel fichier permet de configurer les droits d'accès à l'instance ?
  - Le fichier de configuration *pg_hba.conf* permet des contrôler l'authentification du client.

  - *HBA* signifie « host-based authentication » : authentification fondée sur l'hôte.

- Y a-t-il un lien entre utilisateur système et rôle PostgreSQL ?
  - Si le rôle n'est pas précisé dans la chaîne de connexion, le nom de l'utilisateur système est utilisé.
  - Lors d'une connexion en local par la méthode `peer`, l'accès est permis sans mot de passe pour la connexion d'un utilisateur à l'instance si un rôle du nom de l'utilisateur existe dans l'instance.

- Quel ordre SQL permet de donner des droits sur une table ?
  - L'ordre SQL pour donner des droits sur une table est : `GRANT (...) ON TABLE table_name TO role_specification`.
  - Les droits possibles sont : `SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER`

Outils PostgreSQL

```
admin$ psql -d ojm3 -h 192.168.0.5 -U adb1 -p 5454
```

Connexion à la base de données **ojm3** avec le rôle **adb1** sur l'instance hébergée sur le serveur d'adresse IP 192.168.0.5 au port 5454.

```
postgres@nevrast$ psql -p 5433 -c 'SELECT pg_reload_conf();' -h /tmp
```

Exécution de la requête `SELECT pg_reload_conf();` qui recharge la configuration de l'instance sur l'instance locale au port 5433 (fichier `/tmp/.s.PGSQL.5433`). La connexion se fait à la base de données **postgres** avec le rôle **postgres**.

```
postgres$ createdb -O app_user -h 10.51.32.40 -p 5446 my_app
```

Création de la base de données **my_app** sur l'instance hébergée sur le serveur d'adresse IP 192.168.0.5 au port 5446. Le propriétaire de la base de données sera **app_user**.

```
postgres@localhost$ export PGPORT=5436 && dropuser test
```

Destruction du rôle **test** de l'instance locale au port 5436 (sur un système utilisant systemd, fichier `/var/run/postgresql/.s.PGSQL.5436`). La connexion se fait à la base de données **postgres** avec le rôle **postgres**.

Sauvegardes

```
postgres@dbhost$ pg_dump -Fp -d miranda -f /tmp/miranda.sql -h /tmp/v9.6/
```

Sauvegarde en mode texte de la base de données miranda de l'instance hébergée sur le serveur local au port 5432 (fichier `/tmp/v9.6/.s.PGSQL.5432`). La connexion se fait à la base de données **postgres** avec le rôle **postgres**. La sauvegarde est stockée dans le fichier **/tmp/miranda.sql**.

```
pgadmin@srv$ pg_dump -Fc -U thibaut -t decors -f /home/thibaut/db.dump main
```

Sauvegarde en mode custom (binaire) de la table **decors** de la base de données **main** de l'instance hébergée sur le serveur local au port 5432 (fichier `/var/run/postgresql/.s.PGSQL.5432`). La connexion se fait à la base de données **main** avec le rôle **thibaut**. La sauvegarde est stockée dans le fichier **/tmp/db.dump**.

```
postgres@doriath$ /usr/pgsql-9.4/bin/pg_dump -Fd -d webapp -j 5 -f
↪  /tmp/ma_db
```

Sauvegarde en mode directory (binaire) de la base de données **webapp** de l'instance hébergée sur le serveur local au port 5432 (fichier `/var/run/postgresql/.s.PGSQL.5432`). La connexion se fait à la base de données **webapp** avec le rôle **postgres**. La sauvegarde est stockée dans le répertoire **/tmp/ma_db**. La sauvegarde est parallélisée en traitant 5 tables à la fois.

```
jacques@calvin$ pg_dumpall -U tester -h 192.168.10.38 -f /tmp/ma_db.sql -d
↪  jon1
```

Tentative de sauvegarde en mode texte de l'instance hébergée sur le serveur d'adresse IP 192.168.10.38 au port 5432. La connexion se ferait avec le rôle **tester**. La sauvegarde serait stockée sur le serveur d'où la commande est lancée, calvin, dans le fichier **/tmp/ma_db.sql**.
Le problème vient du dernier paramètre **jon1**. Pour les outils *psql* ou *pg_dump*, le dernier paramètre précisera la base de données à laquelle se connecter. *pg_dumpall* n'accepte pas de tel paramètre. On pourra préciser avec l'option `-l` la base de données à laquelle se connecter pour effectuer la sauvegarde (par défaut *postgres*).

```
jacques@hobbes$ pg_dumpall -g > ~/pg_global.sql
```

Sauvegarde en mode texte des objets globaux (rôles et tablespaces) de l'instance hébergée sur le serveur local au port 5432 (sur un système utilisant systemd, fichier `/var/run/postgresql/.s.PGSQL.5432`). La connexion se fait à la base de données **jacques** avec le rôle **jacques**. La sauvegarde est redirigé de la sortie standard vers le fichier **/home/jacques/pg_global.sql**.

Restaurations

```
db=# \i /home/thibaut/oberon.sql
```

Nous sommes dans la console *psql*. On charge le fichier texte **/home/thibaut/oberon.sql** dans la base de données **db**.

Si l'accès à l'instance se fait de façon distante (*host*), le script chargé est situé sur le disque du client. *psql* ne donne en effet pas accès au système de fichier du serveur hébergeant l'instance.

```
$ psql -U titania -h 11.56.99.77 -d puck -f /tmp/ma_db.sql
```

Chargement du fichier texte **/tmp/ma_db.sql** dans la base de données **puck** de l'instance hébergée sur le serveur d'adresse IP 11.56.99.77 au port 5432. La connexion se fait avec le rôle **titania**. Si le fichier contient des ordres DDL sans précision sur le possesseur, tous les objets créés seront possédés par le rôle **titania**.

```
lysandre@athenes$ pg_restore --clean --create -j3 -d hermia
↪  reve_de_minuit.dump
```

Restauration du contenu de l'archive **reve_de_minuit.dump** sur l'instance hébergée sur le serveur local au port 5432 (fichier `/var/run/postgresql/.s.PGSQL.5432`). La connexion se fait à la base de données **hermia** avec le rôle **lysandre**. La base de données restaurée n'est pas **hermia** qui est juste utilisée pour la connexion et les ordres `DROP DATABASE` et `CREATE DATABASE`. La base de donnée restaurée figure dans le fichier dump. Elle sera préalablement détruite si existante, puis recréée.

```
postgres@src$ pg_restore -U google -h 8.8.8.8 -l -f e3rsf.list e3rsf
```

Sauvegarde dans le fichier **e3rsf.list** de la liste du contenu de l'archive **e3rsf**. Les options de l'adresse de l'hôte et du rôle de connexion ne sont pas utilisées.

```
pgadmin@pghost$ pg_restore -L db.list -d sauvegarde -p 5434 ma_db.dump
```

Restauration des objets SQL contenus dans le fichier **ma_db.dump** et listés dans le fichier **db.list** dans la base de données **sauvegarde** sur l'instance hébergée sur le serveur local au port 5434 (fichier `/var/run/postgresql/.s.PGSQL.5434`). La connexion se fait à la base de données **sauvegarde** avec le rôle **pgadmin**.

Contrôle des accès Ce fichier comporte plusieurs erreurs :

```
host    all         all          127.0.0.1/32    md5
```

autorise tous les utilisateurs, en IP, en local (127.0.0.1) à se connecter à **toutes** les bases, ce qui est en contradiction avec :

```
local   sameuser  all                          peer
```

Le masque CIDR de :

```
host    all       all         89.192.0.3/8    md5
```

est incorrect, ce qui fait qu'au lieu d'autoriser *89.192.0.3* à se connecter, on autorise tout le réseau 89.*.

Les entrées :

```
hostssl   recherche      recherche   89.192.0.4/32   md5
hostssl   replication    repli       89.192.0.0/24   scram-sha-256
```

sont bonnes, mais inutiles, car masquées par la ligne précédente : toute ligne correspondant à cette entrée correspondra aussi à la ligne précédente. Le fichier étant lu séquentiellement, cette dernière entrée ne sert à rien.

Côté sécurité, le mode d'authentification *trust* est à proscrire. Il est conseillé de déclarer un mot de passe au rôle *web* et de passer par le mode d'authentification *md5* ou *scram-sha-256*.

Travaux Pratiques 3

- Sauvegarde / restauration avec PostgreSQL

Rappel

Durant ces travaux pratiques, nous allons travailler sur notre serveur de base de données PostgreSQL via la machine virtuelle du TP 1.

Effectuez les manipulations nécessaires pour réaliser les actions listées dans la section *Énoncés*.

Vous pouvez vous aider des annexes de ce TP, des derniers TP, des sections Administration via bash_ et *psql* ainsi que de l'aide en ligne ou les pages de manuels (`man`).

Énoncés

Installation d'une nouvelle version majeure

- Installer la version 16 de PostgreSQL à partir du PGDG (aide : https://wiki.postgresql.org/wiki/Apt).

Sauvegardes logiques Créer un répertoire `backups` dans le répertoire `HOME` de `postgres` pour y déposer les fichiers d'export.

Les sauvegardes seront à effectuer depuis l'instance en version 13 ou 15.

Sauvegarde logique de toutes les bases

Sauvegarder toutes les bases de données de l'instance PostgreSQL à l'aide de `pg_dumpall` dans le fichier *~postgres/backups/base_all.sql.gz*.

La sauvegarde doit être compressée.

Sauvegarde logique en mode directory d'une base

Sauvegarder de façon parallélisée de la base de données `mondial` au format *directory* à l'aide de `pg_dump` dans le répertoire *~postgres/backups/base_mondial*.

Sauvegarde logique en mode custom d'une base

Sauvegarder de la base de données `cave` au format *custom* à l'aide de `pg_dump` dans le répertoire *~postgres/backups/base_cave.dump*.

Export des objets globaux

Exporter uniquement les objets globaux de l'instance (rôles et définitions de tablespaces) à l'aide de `pg_dumpall` dans le fichier *~postgres/backups/base_globals.sql*.

Sauvegarde logique de tables

Sauvegarder la table `city` dans le fichier *~postgres/backups/table_city.sql* au format *plain text*.

Restaurations logiques Toutes les restaurations sont à effectuer sur la nouvelle instance en version 16.

Restaurations logiques

Restauration des données globales

Restaurer les données globales de l'instance sauvegardée dans le fichier *~postgres/backups/base_globals.sql*.

Restauration d'une base de données

Restaurer le schéma de la base de données `mondial` (pas les données) dans une nouvelle base de données nommée `mondial2` en utilisant le répertoire de sauvegarde `base_mondial`. Le possesseur en sera le rôle *caviste*.

Restauration d'une table

À partir du fichier de sauvegarde *~postgres/backups/table_city.sql*, restaurer la table `city` dans la base de données `mondial2`.

Migration de données

Copier le schéma et les données de la base `mondial` de l'ancienne instance dans une nouvelle base `mondial_test` sans passer par un fichier de sauvegarde.

Restauration partielle

Restaurer dans une base `mondial3` tout le contenu de la sauvegarde de la base `mondial`, sauf les données de la table `city`.

La définition de la table `city` et toutes les contraintes s'y rapportant doivent être restaurées.

Restauration de la base de données cave

À partir du fichier de sauvegarde *~postgres/backups/base_cave.dump*, restaurer de façon parallélisée la base de données `cave`.

Réalisation de sauvegardes physiques à chaud en local

- Créer une sauvegarde physique de l'instance en version 16 avec l'outil `pg_basebackup`.
- Créer une nouvelle instance `copie` avec le commande bash `pg_createcluster`.
- Restaurer la sauvegarde dans cette nouvelle instance et vérifier son bon fonctionnement.

Réalisation de sauvegardes PITR en local

- Configurer l'instance PostgreSQL en version 16 pour réaliser l'archivage des journaux de transactions grâce au paramètre `archive_command`.
- Vérifier le bon archivage des journaux de transactions avec la fonction `pg_switch_wal`.
- Créer une sauvegarde physique avec l'outil `pg_basebackup`.
- Effectuer des actions dans l'instance d'origine (création d'une nouvelle base de données, de nouvelles tables, insertion et suppression de données…). Forcez régulièrement l'archivage des journaux de transactions.
- Créer une nouvelle instance `pitr` avec le commande bash `pg_createcluster`.
- Restaurer la sauvegarde dans cette nouvelle instance et appliquez tous les journaux de transactions archivés (paramètre `restore_command`). Vérifiez que tous les changements sont présents dans la nouvelle instance.

Mise en place de pgBackRest Réalisation de sauvegardes physiques à chaud avec le logiciel pgBackRest :

- Installer le logiciel pgBackRest.
- Configurer le logiciel et l'instance PostgreSQL en version 16 pour réaliser des sauvegardes physique en ligne de commande.
- Créer une sauvegarde physique.
- Restaurer une des sauvegardes dans une nouvelle instance.
- Mettre en place des sauvegardes et purges journalières (par *cron* ou *systemd*).

Solutions du TP 3

Installation d'une nouvelle version majeure Installer la version 16 de PostgreSQL à partir du PGDG :

```
root# sudo apt install curl ca-certificates gnupg
root# curl https://www.postgresql.org/media/keys/ACCC4CF8.asc \
          | gpg --dearmor \
          | sudo tee /etc/apt/trusted.gpg.d/apt.postgresql.org.gpg
           ↪ >/dev/null
root# apt update
root# apt install postgresql-16
```

Une instance *main* sera créée automatiquement. Vérifier son port avec la commande `pg_lsclusters`.

Sauvegardes logiques Créer un répertoire `backups` dans le répertoire home de `postgres` pour y déposer les fichiers d'export.

Se logguer avec l'utilisateur `postgres`, puis exécuter les commandes suivantes :

```
postgres$ cd ~postgres
postgres$ mkdir backups
postgres$ chmod 700 backups
```

Sauvegarde logique de toutes les bases

Sauvegarder toutes les bases de données du cluster PostgreSQL à l'aide de `pg_dumpall` dans le fichier : *~postgres/backups/base_all.sql.gz*

Se logguer avec l'utilisateur `postgres`, puis exécuter la commande suivante :

```
postgres$ pg_dumpall | gzip > ~postgres/backups/base_all.sql.gz
```

Sauvegarde logique en mode directory d'une base

Sauvegarder de façon parallélisée de la base de données `mondial` au format *directory* à l'aide de `pg_dump` dans le répertoire *~postgres/backups/base_mondial*.

Se connecter avec l'utilisateur `postgres`, puis exécuter la commande suivante :

```
postgres$ pg_dump -Fd -j3 -f ~postgres/backups/base_mondial mondial
```

Sauvegarde logique en mode custom d'une base

Sauvegarder de la base de données `cave` au format *custom* à l'aide de `pg_dump` dans le répertoire *~postgres/backups/base_cave.dump*.

```
postgres$ pg_dump -Fc -f ~postgres/backups/base_cave.dump cave
```

Export des objets globaux

Exporter uniquement les objets globaux de l'instance (rôles et définitions de tablespaces) à l'aide de `pg_dumpall` dans le fichier *~postgres/backups/base_globals.sql*.

```
postgres$ pg_dumpall -g > ~postgres/backups/base_globals.sql
```

Sauvegarde logique de tables

Sauvegarder la table `city` dans le fichier *~postgres/backups/table_city.sql* :

```
postgres$ pg_dump -t city mondial > ~postgres/backups/table_city.sql
```

Restaurations logiques On supposera que le port de la nouvelle instance est le 5435.

Restauration des données globales

Restaurer les données globales de l'instance sauvegardée dans le fichier *~postgres/backups/base_globals.sql* :

```
postgres$ psql -p 5435 -f ~postgres/backups/base_globals.sql
```

Restauration d'une base de données

Restaurer le schéma de la base de données `mondial` dans une nouvelle base de données nommée `mondial2` en utilisant le répertoire de sauvegarde `base_mondial`. Le possesseur en sera le rôle *caviste*.

```
postgres$ createdb -p 5435 -O caviste mondial2
postgres$ pg_restore -p 5435 -d mondial2 -s -O --role caviste
 ↪  backups/base_mondial
```

Restauration d'une table

À partir du fichier de sauvegarde *~postgres/backups/table_city.sql*, restaurer la table `city` dans la base de données `mondial2`.

La table city existe déjà, on peut au choix la supprimer ou éditer le fichier de sauvegarde texte. Puis on peut recharger la sauvegarde :

```
postgres$ psql -p 5435 -f ~postgres/backups/table_city.sql mondial2
```

Migration de données

Copier les données de la base `mondial` dans une nouvelle base `mondial_test` sans passer par un fichier de sauvegarde :

```
postgres$ createdb -p 5435 -O caviste mondial_test
postgres$ pg_dump -Fc mondial | pg_restore -p 5435 -d mondial_test
```

Restauration partielle

Lister le contenu de l'archive dans un fichier:

```
postgres$ pg_restore -l ~postgres/backups/base_mondial >
 ↪  /tmp/contenu_archive.lst
```

Éditer */tmp/contenu_archive.lst*, et supprimer ou commenter la ligne `TABLE DATA` de la table `city`.

Créer la base de données `mondial3`, puis restaurer en utilisant ce fichier :

```
postgres$ createdb -p 5435 -O caviste mondial3
postgres$ pg_restore -p 5435 -L /tmp/contenu_archive -d mondial3 \
  ~postgres/backups/base_mondial.dump
```

Restauration de la base de données cave

À partir du fichier de sauvegarde *~postgres/backups/base_cave.dump*, restaurer de façon parallélisée la base de données `cave`.

```
postgres$ pg_restore -p 5435 ~postgres/backups/base_cave.dump
```

Réalisation de sauvegardes physiques à chaud en local

- **Créer une sauvegarde physique avec l'outil `pg_basebackup`.**
- Créer, avec l'utilisateur PostgreSQL, le répertoire pour héberger les sauvegardes physiques :

```
postgres$ mkdir -p ~postgres/backups/16/main/backup
```

- Réalisez la sauvegarde

```
postgres$ pg_basebackup -D ~postgres/backups/16/main/backup/copie_1
```

- L'outil `pg_basebackup` ne sauvegarde pas les fichiers de configurations situés hors du répertoire PGDATA. Il ne faut pas oublier de copier ces fichiers :

```
postgres$ cp -rf /etc/postgresql/16/main/
 ↪  ~postgres/backups/16/main/backup/copie_1/conf_files
```

Créer une nouvelle instance avec le commande bash `pg_createcluster`

Pour restaurer l'instance sauvegardée à la question précédente, nous devons créer une nouvelle instance *copie* :

```
postgres$ pg_createcluster 16 copie
```

Restaurer la sauvegarde dans cette nouvelle instance et vérifier son bon fonctionnement

```
postgres$ rm -rf /var/lib/postgresql/16/copie/*
postgres$ cp -rf ~postgres/backups/16/main/backup/copie_1/*
 ↪  /var/lib/postgresql/16/copie/
```

Recopier les fichiers de configurations (en les adaptant si besoin, par exemple le numéro de port) :

```
postgres$ cp -rf ~postgres/backups/16/main/backup/copie_1/conf_files/*
 ↪  etc/postgresql/16/copy/
postgres$ $ sed -i 's/^port =.*$/port = 5434/'
 ↪  /etc/postgresql/16/copie/postgresql.conf
```

Nous pouvons maintenant démarrer notre copie :

```
postgres$ sudo systemctl restart postgresql@16-copie.service
postgres$ pg_lsclusters 16 copie
Ver Cluster Port Status Owner    Data directory              Log file
16  copie   5435 online postgres /var/lib/postgresql/16/copie
 ↪  /var/log/pos(...)
```

Réalisation de sauvegardes PITR en local

- **Configurer l'instance PostgreSQL pour réaliser l'archivage des journaux de transactions grâce au paramètre `archive_command`**
- Créer, avec l'utilisateur PostgreSQL, le répertoire pour héberger les archives des journaux de transactions :

```
postgres$ mkdir -p ~postgres/backups/16/main/wals
```

- mettre en place l'archivage des journaux de transactions en mettant à jour les paramètres `archive_mode` et `archive_command` :

```
archive_mode = on
archive_command = 'cp "%p" "/var/lib/postgresql/backups/16/main/wals/%f"'
```

Redémarrer l'instance PostgreSQL :

```
# systemctl restart postgresql@16-main.service
```

- **Vérifier le bon archivage des journaux de transactions avec la fonction `pg_switch_wal`**

```
postgres$ psql -c 'SELECT pg_switch_wal()'
postgres$ ls -l ~postgres/backups/16/main/wals/
```

- **Créer une nouvelle sauvegarde physique avec l'outil `pg_basebackup`.**

```
postgres$ mkdir -p ~postgres/backups/16/main/backup
postgres$ pg_basebackup -p 5433 -D ~postgres/backups/16/main/backup/copie_1
```

- L'outil `pg_basebackup` ne sauvegarde pas les fichiers de configurations situés hors du répertoire PGDATA. Il ne faut pas oublier de copier ces fichiers :

```
postgres$ cp -rf /etc/postgresql/16/main/
 ↪  ~postgres/backups/16/main/backup/copie_1/conf_files
```

- **Effectuer des actions dans l'instance d'origine. Forcez régulièrement l'archivage des journaux de transactions.**

```
postgres$ createdb nouvelle
postgres$ psql -d nouvelle

# select pg_switch_wal();
# create table ma_table (i int);
# select pg_switch_wal();
# insert into ma_table (i) select generate_series(1, 10000000);
# select pg_switch_wal();
# delete from ma_table where i%4 = 0;
# select pg_switch_wal();
```

Créer une nouvelle instance avec le commande bash `pg_createcluster`

Pour restaurer l'instance sauvegardée à la question précédente, nous devons créer une nouvelle instance *pitr* :

```
postgres$ pg_createcluster 16 pitr
```

Restaurer la sauvegarde dans cette nouvelle instance et vérifier son bon fonctionnement

Copie des fichiers :

```
postgres$ rm -rf /var/lib/postgresql/16/pitr/*
postgres$ cp -rf ~postgres/backups/16/main/backup/copie_1/*
 ↪  /var/lib/postgresql/16/pitr/
```

Recopier les fichiers de configurations (en les adaptant si besoin, par exemple le numéro de port) :

```
postgres$ cp -rf ~postgres/backups/16/main/backup/copie_1/conf_files/*
 ↪  /etc/postgresql/16/pitr/
postgres$ $ sed -i 's/^port =.*$/port = 5435/'
 ↪  /etc/postgresql/16/pitr/postgresql.conf
```

Désactivation de l'archivage sur la nouvelle instance dans `/etc/postgresql/16/pitr/postgresql.conf` :

```
archive_command = '/bin/true'
```

Configuration de la nouvelle instance pour rejouer les journaux de transactions, édition du paramètre `restore_command` dans `/etc/postgresql/16/pitr/postgresql.conf` :

```
restore_command = 'cp /var/lib/postgresql/backups/16/main/wals/%f "%p"'
```

Créer un fichier `recovery.signal` dans *PGDATA* pour indiquer qu'on créé une nouvelle instance (et non une instance secondaire en réplication) :

```
postgres$ touch /var/lib/postgresql/16/pitr/recovery.signal
```

Restaurer la sauvegarde dans cette nouvelle instance et vérifier son bon fonctionnement

Copie des fichiers :

```
postgres$ rm -rf /var/lib/postgresql/16/copie/*
postgres$ cp -rf ~postgres/backups/16/main/backup/first_copie/*
 ↪  /var/lib/postgresql/16/copie/
```

Nous pouvons maintenant démarrer notre copie :

```
postgres$ sudo systemctl restart postgresql@16-copie.service
postgres$ pg_lsclusters 16 copie
Ver Cluster Port Status Owner    Data directory              Log file
16  copie   5435 online postgres /var/lib/postgresql/16/copie
 ↪  /var/log/pos(...)
```

Mise en place de pgBackRest Réalisation de sauvegardes physiques à chaud avec le logiciel pgBackRest :

- **Installer le logiciel pgBackRest**

```
# apt install pgbackrest -y
```

- **Configurer le logiciel et l'instance PostgreSQL pour réaliser des sauvegardes physique en ligne de commande**

Une sauvegarde PITR comprend 2 parties : une copie physique de l'instance, ainsi que tous les fichiers WAL depuis le début de la copie de l'instance. Ces fichiers WAL peuvent être archiver par PostgreSQL en lui précisant la ligne de commande à exécuter pour chaque fichier WAL rempli.

Nous allons créer une *stanza* pgbackrest (la configuration nécessaire pour sauvegarder notre instance).

- Créer le répertoire de configuration de pgbackrest :

```
# mkdir -p /etc/pgbackrest
# chown postgres: /etc/pgbackrest
```

- Éditez les fichier `/etc/pgbackrest/pgbackrest.conf` avec l'utilisateur *postgres* :

```
[global]
repo1-path=/var/lib/postgresql/backups/pgbackrest
repo1-retention-full=2
start-fast=y
```

```
[db]
pg1-path=/var/lib/postgresql/16/main
pg1-port=5434
```

- Initier la *stanza*:

```
postgres$ pgbackrest --stanza=db --log-level-console=info stanza-create
```

Nous allons ensuite configurer PostgreSQL pour activer l'archivage des WAL.

- Éditer le fichier de configuration de PostgreSQL :

```
archive_command = 'pgbackrest --stanza=db archive-push %p'
archive_mode = on
```

- Redémarrer l'instance

```
# systemctl restart postgresql@16-main.service
```

- Vérifions l'archivage des WAL :

```
postgres$ ls -l /var/lib/postgresql/backups/pgbackrest/archive/db/16-1/*
postgres$ psql -c 'CREATE DATABASE test'
postgres$ psql -c 'SELECT pg_switch_wal()'
postgres$ ls -l /var/lib/postgresql/backups/pgbackrest/archive/db/16-1/*
```

On doit avoir un répertoire vide lors du premier appel à *ls* et avoir un fichier du type `000000010000000000000000E-fde16e3a71f0b78d42a486a2d4bad23b588022d5.gz` au deuxième appel.

Si aucun fichier n'est présent, il faut vérifier dans les logs de PostgreSQL pour trouver la cause du problème.

```
postgres$ less /var/log/postgresql/postgresql-16-main.log
```

On peut également vérifier la configuration de notre *stanza* :

```
postgres$ pgbackrest --stanza=db --log-level-console=info check
```

- **Créer une sauvegarde physique.**

Exécutez les sauvegarde :

```
postgres$ pgbackrest --stanza=db --log-level-console=info backup
```

Vérifier le contenu :

```
postgres$ ls ~/backups/pgbackrest/backup/db/latest/pg_data/
backup_label.gz  pg_dynshmem   pg_replslot   pg_stat_tmp  PG_VERSION.gz
base             pg_logical    pg_serial     pg_subtrans  pg_wal
global           pg_multixact  pg_snapshots  pg_tblspc    pg_xact
pg_commit_ts     pg_notify     pg_stat       pg_twophase
 ↪  postgresql.auto.conf.gz
```

- **Restaurer une sauvegarde PITR dans une nouvelle instance.**

Création de la nouvelle instance :

```
# pg_createcluster 16 restoration
# rm -rf /var/lib/postgresql/16/restoration/*
```

Restauration de la sauvegarde dans la nouvelle instance:

```
postgres$ pgbackrest --stanza=db \
    --pg1-path /var/lib/postgresql/16/restoration/ restore
```

Configuration de la nouvelle instance pour rejouer les journaux de transactions en utilisant pgBackRest, édition du paramètre `restore_command` dans `/etc/postgresql/16/restoration/postgresql.c`

```
restore_command = 'pgbackrest --stanza=db archive-get %f "%p"'
```

Créer un fichier `recovery.signal` dans *PGDATA* pour indiquer qu'on créé une nouvelle instance (et non une instance secondaire en réplication) :

```
postgres$ touch /var/lib/postgresql/16/restoration/recovery.signal
```

- Démarrer l'instance :

```
# systemctl restart postgresql@16-restoration.service
```

- Vérifier le bon fonctionnement en étudiant les logs et en se connectant sur l'instance.

```
postgresql$ psql -p 5435
psql (16.1 (Debian 16.1-1.pgdg120+1))
Saisissez « help » pour l'aide.

postgres=# \conninfo
Vous êtes connecté à la base de données « postgres » en tant qu'utilisateur
 ↪  « postgres » via le socket dans « /var/run/postgresql » via le port «
 ↪  5435 ».
```

- **Mettre en place des sauvegardes et purges journalières (par *cron* ou *systemd*).**

Mettre en place des sauvegardes et purges journalières

Ajout dans le crontab de l'utilisateur *postgres* avec la commande `crontab -e` :

```
30 00 * * * /usr/bin/pgbackrest --stanza=db --log-level-console=info backup
```

Avec systemd, création des deux fichiers suivants :

- /etc/systemd/system/pitr_backrest.service :

```
[Unit]
Description=Service de sauvegarde PITR de 16-main par pitr
After=local-fs.target

[Service]
Type=oneshot
RemainAfterExit=no
User=postgres
Group=postgres
ExecStart=/usr/bin/pgbackrest --stanza=db --log-level-console=info backup

[Install]
WantedBy=multi-user.target
```

- /etc/systemd/system/pitr_backrest.timer :

```
[Unit]
Description=lancement journalier de la sauvegarde PITR de 16-main

[Timer]
OnCalendar=00:30:00
Persistent=true

[Install]
WantedBy=timers.target
```

Puis activation des services :

```
# systemctl enable pitr_backrest.timer
# systemctl enable pitr_backrest.service
```

On pourra choisir une heure précise permettant de vérifier durant le TP du bon fonctionnement de la sauvegarde.
