Module I1

Sauvegarde et restauration

DALIBO
L'expertise PostgreSQL

22.09

Dalibo SCOP

https://dalibo.com/formations

Sauvegarde et restauration

Module I1

TITRE : Sauvegarde et restauration
SOUS-TITRE : Module I1

REVISION: 22.09
DATE: 02 septembre 2022

Remerciements : Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment : Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

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

Vous êtes autorisé à :

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

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

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

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

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

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

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

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

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

[1] mailto:contact@dalibo.com

Chers lectrices & lecteurs,

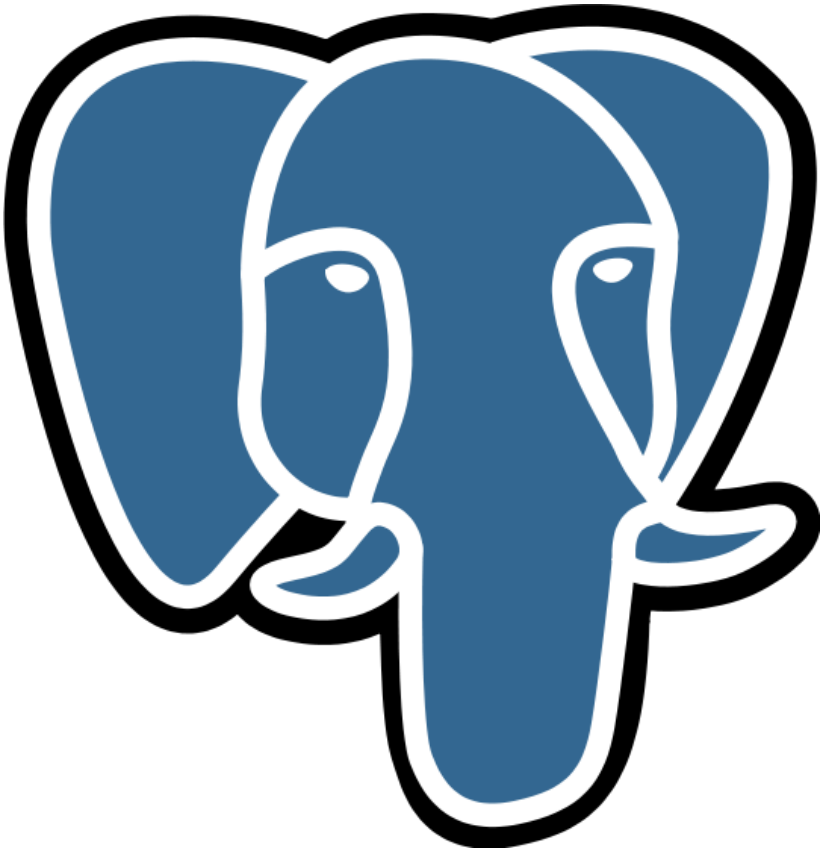
Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est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

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

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

Table des Matières

1 POSTGRESQL : SAUVEGARDE ET RESTAURATION



1.1 INTRODUCTION

- 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 bogue 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 cohérentes. 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.

1.1.1 AU MENU

- Sauvegardes logiques
- Sauvegarde physique à froid des fichiers

1.2 SAUVEGARDES LOGIQUES

- À chaud
- Cohérente
- Locale ou à distance
- 2 outils
  - `pg_dump`
  - `pg_dumpall`
- Pas d'impact sur les utilisateurs
  - sauf certaines opérations DDL
- Jamais inclus :
  - tables systèmes
  - fichiers de configuration

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.

La technique alors utilisée permet de s'assurer de la cohérence des données : lors de la sauvegarde, l'outil ne voit pas les modifications faites par les autres utilisateurs. Pour cela, quand il se connecte à la base à sauvegarder, il commence une transaction pour que sa vision des enregistrements de l'ensemble des tables soit cohérente. Cela empêche le recyclage des enregistrements par `VACUUM` pour les enregistrements dont il pourrait avoir besoin. Par conséquent, que la sauvegarde dure 10 minutes ou 10 heures, le résultat correspondra au contenu de la base telle qu'elle était au début de la transaction.

Des verrous sont placés sur chaque table, mais leur niveau est très faible (*Access Share*). Il visent juste à éviter la suppression des tables pendant la sauvegarde, ou la modification de leur structure. Les opérations habituelles sont toutes permises en lecture ou écriture, sauf quand elles réclament un verrou très invasif, comme `TRUNCATE`, `VACUUM FULL` ou certains `LOCK TABLE`. Les verrous ne sont relâchés qu'à la fin de la sauvegarde.

Par ailleurs, pour assurer une vision cohérente de la base durant toute la durée de son export, cette transaction de longue durée est de type *REPEATABLE READ* et non de type *READ COMMITED* utilisé par défaut.

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` permet d'extraire le contenu d'une seule base de données dans différents formats.

(Pour rappel, une instance PostgreSQL contient plusieurs bases de données.)

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

`psql` exécute les ordres SQL contenus dans des *dumps* (sauvegardes) au format texte.

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

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

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`.

L'export ne concerne que les données des utilisateurs : les tables systèmes ne sont jamais concernées, il n'y a pas de risque de les écraser lors d'un import. En effet, les schémas systèmes `pg_catalog` et `information_schema` et leurs objets sont gérés uniquement par PostgreSQL. Vous n'êtes d'ailleurs pas censé modifier leur contenu, ni y ajouter ou y effacer quoi que ce soit !

La configuration du serveur (fichiers `postgresql.conf`, `pg_hba.conf`…) n'est jamais incluse et doit être sauvegardée à part. Un export logique ne concerne que des données.

Les extensions ne posent pas de problème non plus : la sauvegarde contiendra une mention de l'extension, et les *données* des éventuelles tables gérées par cette extension. Il faudra que les binaires de l'extension soient installés sur le système cible.

1.2.1 PG_DUMP

- Sauvegarde une base de données
  `pg_dump nombase > nombase.dump`
- Sauvegarde complète ou partielle

`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 :

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

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

Sous Windows avec le Powershell, préférer la syntaxe `pg_dump -f b1.dump b1`, car une redirection avec `>` peut corrompre la sauvegarde.

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.

1.2.2 PG_DUMP - FORMAT DE SORTIE

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

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

`pg_dump` accepte d'enregistrer la sauvegarde suivant quatre formats :

- le format SQL, soit un fichier texte unique pour toute la base, non compressé ;
- le format tar, un fichier binaire, non compressé, comprenant un index des objets ;
- le format « personnalisé » (*custom*), un fichier binaire, compressé, avec un index des objets ;
- le format « répertoire » (*directory*), arborescence de fichiers binaires généralement compressés, comprenant aussi un index des objets.

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

- `plain` ou `p` pour un fichier SQL (texte) ;
- `tar` ou `t` pour un fichier tar ;
- `custom` ou `c` pour un fichier « personnalisé » ;
- `directory` ou `d` pour le répertoire.

Le format `plain` est lisible directement par psql. Les autres nécessitent de passer par `pg_restore` pour restaurer tout ou partie de la sauvegarde.

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

- configuration de certaines variables ;
- création des objets de la base : schémas, tables, vues, procédures stockées, etc., à l'exception des index, contraintes et triggers ;
- ajout des données aux tables (ordres `COPY` par défaut) ;
- ajout des index, contraintes et triggers ;
- définition des droits d'accès aux objets ;
- rafraîchissement des vues matérialisées.

Les index figurent vers la fin pour des raisons de performance : il est plus rapide de créer un index à partir des données finales que de le mettre à jour en permanence pendant l'ajout des données. Les contraintes et le rafraîchissement des vues matérialisées sont aussi à la fin parce qu'il faut que les données soient déjà restaurées dans leur ensemble. Les triggers ne devant pas être déclenchés pendant la restauration, ils sont aussi restaurés vers la fin. Les propriétaires sont restaurés pour chacun des objets.

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

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

$ time pg_dump -Ft b1 > b1.Ft.dump
real    0m37.478s
user    0m10.387s
sys 0m2.285s

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

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

La sauvegarde la plus longue est la sauvegarde au format personnalisée (`custom`) car elle est compressée. La sauvegarde au format répertoire se trouve entre la sauvegarde au format personnalisée et la sauvegarde au format `tar` : elle est aussi compressée mais sur des fichiers plus petits.

En terme de taille :

```
$ du -sh b1.F?.dump
```

```
116M    b1.Fc.dump
116M    b1.Fd.dump
379M    b1.Fp.dump
379M    b1.Ft.dump
```

Le format compressé est évidemment le plus petit. Le format texte et le format tar[2] sont les plus lourds à cause du manque de compression. Le format tar est même généralement un peu plus lourd que le format texte à cause de l'entête des fichiers tar.

De plus, avant la 9.5 et l'utilisation d'extensions du format tar, il n'était pas possible d'y stocker une table dont la représentation physique dans l'archive tar dépassait 8 Go, avec cette erreur :

```
$ pg_dump -Ft tar > tar.Ft.dump
pg_dump: [tar archiver] archive member too large for tar format
```

1.2.3 CHOIX DU FORMAT DE SORTIE

- Format `plain` (SQL)
  - restaurations partielles très difficiles (ou manuelles)
- Parallélisation du dump
  - format `directory`
- Objets globaux seulement :
  - `pg_dumpall -g`
- Formats binaires conseillés
  - `pg_dump -Fc` ou `-Fd`
- => combiner `pg_dump -Fc/Fd` et `pg_dumpall -g`

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

Tout d'abord, le format `tar` est à éviter. Il n'apporte aucune plus-value par rapport au format `custom` (et ajoute la limitation vue ci-dessus avant la 9.5).

Ensuite, même si c'est le plus portable (et le seul disponible avec `pg_dumpall`), le format `plain` rend les restaurations partielles difficiles car il faut extraire manuellement le SQL d'un fichier texte souvent très volumineux. Ce peut être ponctuellement pratique cependant, mais on peut aisément regénérer un fichier SQL complet à partir d'une sauvegarde binaire et `pg_restore`.

Certaines informations (notamment les commandes `ALTER DATABASE … SET` pour modifier un paramètre pour une base précise) ne sont pas générées par `pg_dump -Fp`, à moins de penser à rajouter `--create` (pour les ordres de création). Par contre, elles sont in-

[2] https://fr.wikipedia.org/wiki/Tar_(informatique)

cluses dans les entêtes des formats `custom` ou `directory`, où un `pg_restore --create` saura les retrouver.

On privilégiera donc les formats `custom` et `directory` pour plus de flexibilité à la restauration.

Le format `directory` ne doit pas être négligé : il permet d'utiliser la fonctionnalité de sauvegarde en parallèle de `pg_dump --jobs`, avec de gros gains de temps d'exécution (ou de migration) à la clé.

Enfin, l'outil `pg_dumpall`, initialement prévu pour les montées de versions majeures, permet de sauvegarder les objets globaux d'une instance : la définition des rôles et des tablespaces.

Ainsi, pour avoir la sauvegarde la plus complète possible d'une instance, il faut combiner `pg_dumpall -g` (pour la définition des objets globaux), et `pg_dump` (pour sauvegarder les bases de données une par une au format `custom` ou `directory`).

1.2.4 PG_DUMP - COMPRESSION

- `-Z` : compression par zlib
  - de 0 à 9
  - défaut 6

La compression permet de réduire énormément la volumétrie d'une sauvegarde logique par rapport à la taille physique des fichiers de données.

Par défaut, `pg_dump -Fc` ou `-Fd` utilise le niveau de compression par défaut de la zlib, à priori le meilleur compromis entre compression et vitesse, correspondant à la valeur 6. `-Z1` comprimera peu mais rapidement, et `-Z9` sera nettement plus lent mais compressera au maximum. Seuls des tests permettent de déterminer le niveau acceptable pour un cas d'utilisation particulier.

Le format `plain` (pur texte) accepte aussi l'option `-Z`, ce qui permet d'obtenir un export texte compressé en gzip. Cependant, cela ne remplace pas complètement un format `custom`, plus souple.

1.2.5 PG_DUMP - FICHIER OU SORTIE STANDARD

- `-f` : fichier où stocker la sauvegarde
- sinon : 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.

1.2.6 PG_DUMP - STRUCTURE OU DONNÉES ?

- `--schema-only` ou `-s` : uniquement la structure
- `--data-only` ou `-a` : uniquement les données

Il est possible de ne sauvegarder que la structure avec l'option `--schema-only` (ou `-s`). De cette façon, seules les requêtes de création d'objets seront générées. Cette sauvegarde est généralement très rapide. Cela permet de créer un serveur de tests très facilement.

Il est aussi possible de ne sauvegarder que les données pour les réinjecter dans une base préalablement créée avec l'option `--data-only` (ou `-a`).

1.2.7 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

Ces options apparaissent avec la version 9.2.

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

1.2.8 PG_DUMP - SÉLECTION D'OBJETS

- `-n <schema>` : uniquement ce schéma
- `-N <schema>` : tous les schémas sauf celui-là
- `-t <table>` : uniquement cette relation (sans dépendances !)
- `-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`.

Notez que les dépendances ne sont pas gérées. Si vous demandez à sauvegarder une vue avec `pg_dump -t unevue`, la sauvegarde ne contiendra **pas** les définitions des objets nécessaires à la construction de cette vue.

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
```

1.2.9 PG_DUMP - OPTION DE PARALLÉLISATION

- `--jobs <nombre_de_threads>`
- format directory (`-Fd`) uniquement

Par défaut, `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.

Cette option n'est compatible qu'avec le format de sortie `directory` (option `-Fd`).

La parallélisation de requêtes ne s'applique hélas pas aux ordres `COPY` utilisés par `pg_dump`. Mais ce dernier peut en lancer plusieurs simultanément avec l'option `--jobs` (`-j`). Elle permet de préciser le nombre de connexions vers la base de données, et aussi de connexions (Unix) ou threads (Windows).

Cela permet d'améliorer considérablement la vitesse de sauvegarde, à condition de pouvoir réellement paralléliser. Par exemple, si une table occupe 15 Go sur une base de données de 20 Go, il y a peu de chance que la parallélisation change fondamentalement la durée de sauvegarde.

1.2.10 PG_DUMP - OPTIONS DIVERSES

- `--create` (`-C`) : recréer la base
  - y compris paramétrage utilisateur sur base (>v11 et format `plain`)
  - inutile dans les autres formats
- `--no-owner` : ignorer le propriétaire
- `--no-privileges` : ignorer les droits
- `--no-tablespaces` : ignorer les tablespaces
- `--inserts` : remplacer `COPY` par `INSERT`
- `--rows-per-insert` , `--on-conflict-do-nothing`
- `-v` : progression

Il reste quelques options plus anecdotiques mais qui peuvent se révéler très pratiques.

--create :

`--create` (ou `-C`) n'a d'intérêt qu'en format texte (`-Fp`), pour ajouter les instructions de création de base :

```
CREATE DATABASE b1 WITH TEMPLATE = template0
                    ENCODING = 'UTF8' LC_COLLATE = 'C' LC_CTYPE = 'C';
```

```
ALTER DATABASE b1 OWNER TO postgres;
```

et éventuellement, s'il y a un paramétrage propre à la base (si le client est en version 11 minimum) :

```
ALTER DATABASE b1 SET work_mem TO '100MB' ;
ALTER ROLE chef IN DATABASE b1 SET work_mem TO '1GB';
```

À partir de la version 11, ces dernières informations sont incluses d'office aux formats `custom` ou `directory`, où `pg_restore --create` peut les retrouver.
Jusqu'en version 10, elles ne se retrouvaient que dans `pg_dumpall` (sans `-g`), ce qui n'était pas pratique quand on ne restaurait qu'une base.
Il faut bien vérifier que votre procédure de restauration reprend ce paramétrage.

Avec `--create`, la restauration se fait en précisant une autre base de connexion, généralement `postgres`. L'outil de restauration basculera automatiquement sur la base nouvellement créée dès que possible.

Masquer des droits et autres propriétés :

`--no-owner`, `--no-privileges`, `--no-comments` et `--no-tablespaces` permettent de ne pas récupérer respectivement le propriétaire, les droits, le commentaire et le tablespace de l'objet dans la sauvegarde s'ils posent problème.

Ordres INSERT au lieu de COPY :

Par défaut, `pg_dump` génère des commandes `COPY`, qui sont bien plus rapides que les `INSERT`. Cependant, notamment pour pouvoir restaurer plus facilement la sauvegarde sur un autre moteur de bases de données, il est possible d'utiliser des `INSERT` au lieu des `COPY`. Il faut forcer ce comportement avec l'option `--inserts`.

L'inconvénient des `INSERT` ligne à ligne est leur lenteur par rapport à un `COPY` massif (même avec des astuces comme `synchronous_commit=off`). Par contre, l'inconvénient de `COPY` est qu'en cas d'erreur sur une ligne, tout le `COPY` est annulé. Pour diminuer l'inconvénient des `INSERT` tout en conservant leur intérêt, il est possible d'indiquer le nombre de lignes à intégrer par `INSERT` avec l'option `--rows-per-insert` : si un `INSERT` échoue, seulement ce nombre de lignes sera annulé.

L'option `--on-conflict-do-nothing` permet d'éviter des messages d'erreur si un `INSERT` tente d'insérer une ligne violant une contrainte existante. Très souvent ce sera pour éviter des problèmes de doublons (de même clé primaire) dans une table déjà partiellement chargée, avec les contraintes déjà en place.

Ces deux dernières options sont disponibles à partir de la version 12 du client.

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

1.2.11 PG_DUMPALL

- Sauvegarde d'une instance complète
  - objets globaux (utilisateurs, tablespaces…)
  - toutes les bases de données
- Format texte (SQL) uniquement

`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.

Contrairement à `pg_dump`, `pg_dumpall` ne dispose que d'un format en sortie : des ordres SQL en texte.

1.2.12 PG_DUMPALL - FICHIER OU SORTIE STANDARD

- `-f` : fichier où est stockée la sauvegarde
- sinon : 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.

1.2.13 PG_DUMPALL - SÉLECTION DES OBJETS

- `-g` : tous les objets globaux
- `-r` : uniquement les rôles
- `-t` : uniquement les tablespaces
- `--no-role-passwords` : sans 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 décla-

ration des objets globaux, ou des rôles, ou des tablespaces. Leur versions longues sont respectivement : `--globals-only`, `--roles-only` et `--tablespaces-only`.

Par exemple, voici la commande pour ne sauvegarder que les rôles :

```
$ pg_dumpall -r
--
-- PostgreSQL database cluster dump
--

SET default_transaction_read_only = off;

SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;

--
-- Roles
--

CREATE ROLE admin;
ALTER ROLE admin WITH SUPERUSER INHERIT NOCREATEROLE NOCREATEDB NOLOGIN
                 NOREPLICATION NOBYPASSRLS;
CREATE ROLE dupont;
ALTER ROLE dupont WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN
                 NOREPLICATION NOBYPASSRLS
                 PASSWORD 'md5505548e69dafa281a5d676fe0dc7dc43';
CREATE ROLE durant;
ALTER ROLE durant WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN
                 NOREPLICATION NOBYPASSRLS
                 PASSWORD 'md56100ff994522dbc6e493faf0ee1b4f41';
CREATE ROLE martin;
ALTER ROLE martin WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN
                 NOREPLICATION NOBYPASSRLS
                 PASSWORD 'md5d27a5199d9be183ccf9368199e2b1119';
CREATE ROLE postgres;
ALTER ROLE postgres WITH SUPERUSER INHERIT CREATEROLE CREATEDB LOGIN
                 REPLICATION BYPASSRLS;
CREATE ROLE utilisateur;
ALTER ROLE utilisateur WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB NOLOGIN
                 NOREPLICATION NOBYPASSRLS;

--
-- User Configurations
--

--
-- User Config "u1"
```

```
--

ALTER ROLE u1 SET maintenance_work_mem TO '256MB';

--
-- Role memberships
--

GRANT admin TO dupont GRANTED BY postgres;
GRANT admin TO durant GRANTED BY postgres;
GRANT utilisateur TO martin GRANTED BY postgres;


--
-- PostgreSQL database cluster dump complete
--
```

On remarque que le mot de passe est sauvegardé sous forme de *hash*.

La sauvegarde des rôles se fait en lisant le catalogue système `pg_authid`. Seuls les superutilisateurs ont accès à ce catalogue système car il contient les mots de passe des utilisateurs.

À partir de PostgreSQL 10, on peut permettre d'utiliser `pg_dumpall` sans avoir un rôle superutilisateur, avec l'option `--no-role-passwords`. Celle-ci a pour effet de ne pas sauvegarder les mots de passe. Dans ce cas, `pg_dumpall` va lire le catalogue système `pg_roles` qui est accessible par tout le monde.

1.2.14 PG_DUMPALL - EXCLURE UNE BASE

- Possibilité d'exclure une base (v12)
- `--exclude-database`

Par défaut, `pg_dumpall` sauvegarde les objets globaux et toutes les bases de l'instance. Dans certains cas, il peut être intéressant d'exclure une (ou plusieurs bases). L'option `--exclude-database` a été ajoutée pour cela. Elle n'est disponible qu'à partir de la version 12.

1.2.15 PG_DUMPALL - OPTIONS DIVERSES

- Quelques options partagées avec `pg_dump`
- Très peu utilisées

Il existe d'autres options gérées par `pg_dumpall`. Elles sont déjà expliquées pour la commande `pg_dump` et sont généralement peu utilisées avec `pg_dumpall`.

1.2.16 PG_DUMP/PG_DUMPALL - OPTIONS DE CONNEXIONS

- `-h` / `$PGHOST` / socket Unix
- `-p` / `$PGPORT` / 5432
- `-U` / `$PGUSER` / utilisateur du système
- `-W`/ `$PGPASSWORD`
  - ou `.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, mais force `pg_dump` à le demander (en interactif donc, et qu'il soit vérifié ou non, ceci dépendant de la méthode d'authentification).

En général, une sauvegarde automatique est effectuée sur le serveur directement par l'utilisateur système `postgres` (connexion par `peer` sans mot de passe), ou à distance avec le mot de passe stocké dans un fichier `.pgpass`.

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 secondaire,primaire -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 (c'est-à-dire des droits 600). L'outil vérifiera cela avant d'accepter d'utiliser les informations qui s'y trouvent.

1.2.17 IMPACT DES PRIVILÈGES

- Les outils se comportent comme des clients pour PostgreSQL
- Préférer un rôle superutilisateur
- Sinon :
  - connexion à autoriser
  - le rôle doit pouvoir lire tous les objets à exporter

Même si ce n'est pas obligatoire, il est recommandé d'utiliser un rôle de connexion disposant des droits de superutilisateur pour la sauvegarde et la restauration.

En effet, pour sauvegarder, il faut pouvoir :

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

Pour restaurer, il faut pouvoir :

- se connecter à la base de données : autorisation dans `pg_hba.conf`, être propriétaire ou avoir le privilège `CONNECT` ;
- optionnellement, pouvoir créer la base de données cible et pouvoir s'y connecter (option `-C` de `pg_restore`)
- pouvoir créer des schémas : être propriétaire de la base de données ou avoir le privilège `CREATE` sur celle-ci ;
- pouvoir créer des objets dans les schémas : être propriétaire du schéma ou avoir le privilège `CREATE` sur celui-ci ;
- pouvoir écrire dans les tablespaces cibles : être propriétaire du tablespace ou avoir le privilège `CREATE` sur celui-ci ;
- avoir la capacité de donner ou retirer des privilèges : faire partie des rôles bénéficiant d'ACL dans le `dump`.

Le nombre de ces privilèges explique pourquoi il n'est parfois possible de ne restaurer qu'avec un superutilisateur.

1.2.18 TRAITER AUTOMATIQUEMENT LA SORTIE

- Pour compresser : `pg_dump | bzip2`
  - utile avec formats `plain`,`tar`,`custom`
- Outils multi-threads de compression, bien plus rapides :
  - `pbzip2`
  - `pigz`

À part pour le format `directory`, il est possible d'envoyer la sortie standard à un autre outil. Pour les formats non compressés (`tar` et `plain`), cela permet d'abord de compresser avec l'outil de son choix.

Il y a un intérêt même avec le format `custom` : celui d'utiliser des outils de compression plus performants que la zlib, comme `bzip2` ou `lzma` (compression plus forte au prix d'une exécution plus longue) ou `pigz`[3] , `pbzip2`[4] , beaucoup plus rapides grâce à l'utilisation de plusieurs threads. On met ainsi à profit les nombreux processeurs des machines récentes, au prix d'un très léger surcoût en taille. Ces outils sont présents dans les distributions habituelles.

Au format `custom`, il faut penser à désactiver la compression par défaut, comme dans cet exemple avec `pigz` :

```
$ pg_dump -Fc -Z0 -v foobar | pigz > sauvegarde.dump.gzip
```

On peut aussi utiliser n'importe quel autre outil Unix. Par exemple, pour répartir sur plusieurs fichiers :

```
$ pg_dump | split
```

Nous verrons plus loin que la sortie de `pg_dump` peut même être fournie directement à `pg_restore` ou `psql`, ce qui est fort utile dans certains cas.

---

[3] https://www.zlib.net/pigz/
[4] http://compression.ca/pbzip2/

1.2.19 OBJETS BINAIRES

- Deux types dans PostgreSQL : bytea et Large Objects
- Option `-b`
  - uniquement si utilisation des options `-n`/`-N` et/ou `-t`/`-T`
- Option `--no-blobs`
  - pour ne pas sauvegarder les Large Objects
- Option `bytea_output`
  - `escape`
  - `hex`

Il existe deux types d'objets binaires dans PostgreSQL : les Large Objects et les bytea.

Les Larges Objects sont stockées dans une table système appelé `pg_largeobjects`, et non pas dans les tables utilisateurs. Du coup, en cas d'utilisation des options `-n`/`-N` et/ou `-t`/`-T`, la table système contenant les Large Objects sera généralement exclue. Pour être sûr que les Large Objects soient inclus, il faut en plus ajouter l'option `-b`. Cette option ne concerne pas les données binaires stockées dans des colonnes de type bytea, ces dernières étant réellement stockées dans les tables utilisateurs.

Il est possible d'indiquer le format de sortie des données binaires, grâce au paramètre `bytea_output` qui se trouve dans le fichier `postgresql.conf`. Le défaut est `hex`.

Si vous restaurez une sauvegarde d'une base antérieure à la 9.0, notez que la valeur par défaut était différente (`escape`).

Le format `hex` utilise deux octets pour enregistrer un octet binaire de la base alors que le format `escape` utilise un nombre variable d'octets. Dans le cas de données ASCII, ce format n'utilisera qu'un octet. Dans les autres cas, il en utilisera quatre pour afficher textuellement la valeur octale de la donnée (un caractère d'échappement suivi des trois caractères du codage octal). La taille de la sauvegarde s'en ressent, sa durée de création aussi (surtout en activant la compression).

1.2.20 EXTENSIONS

- Option `--extension` (`-e`)
  - uniquement si sélection/exclusion (`-n`/`-N` et/ou `-t`/`-T`)

Les extensions sont sauvegardées par défaut.

Cependant, dans le cas où les options `-n`/`-N` (sélection/exclusion de schéma) et/ou `-t`/`-T` (sélection/exclusion de tables) sont utilisées, les extensions ne sont pas sauvegardées. Or, elles pourraient être nécessaires pour les schémas et tables sélectionnées. L'option `-e` permet de forcer la sauvegarde des extensions précisées.

1.3 RESTAURATION D'UNE SAUVEGARDE LOGIQUE

- `psql` : restauration de SQL (option `-Fp`) :
- `pg_restore` : restauration binaire (`-Ft`/`-Fc`/`-Fd`)

`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.

1.3.1 PSQL

- Client standard PostgreSQL
- Capable d'exécuter des requêtes
  - donc de restaurer une sauvegarde 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`.

1.3.2 PSQL - OPTIONS

- `-f` pour indiquer le fichier contenant la sauvegarde
  - sans `-f` : lit l'entrée standard
- `-1` (`--single-transaction`) : pour tout restaurer en une seule transaction
- `-e` pour afficher les ordres SQL exécutés
- `ON_ERROR_ROLLBACK`/`ON_ERROR_STOP`

`psql` étant le client standard de PostgreSQL, le dump au format `plain` se trouve être un script SQL qui peut également contenir des commandes psql, comme `\connect` pour se connecter à une base de données (ce que fait `pg_dumpall` pour changer de base de données).

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

On peut restaurer avec `psql` de plusieurs manières :

- 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 `psql` fonctionne par défaut en autocommit : après une erreur, les requêtes précédentes sont déjà 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 souvent conseillé 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 existent, parfois 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 il est à `off`, et toute erreur dans **une transaction** entraîne le `ROLLBACK` de toute la transaction. Les commandes suivantes échouent toutes. Activer `ON_ERROR_ROLLBACK` permet de n'annuler que la commande en erreur. psql effectue des *savepoints* avant chaque

ordre, et y retourne en cas d'erreur, avant de continuer le script, toujours dans la même transaction.

Cette option n'est donc **pas** destinée à tout arrêter en cas de problème, au contraire. Mais elle peut être utile pour passer outre à une erreur quand on utilise `-1` pour enrober le script dans une transaction.

`ON_ERROR_ROLLBACK` 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.

C'est l'option à privilégier quand on veut arrêter un script au moindre problème. Si `-1` est utilisé, et que `ON_ERROR_ROLLBACK` est resté à `off`, le script entier est bien sûr annulé, et on évite les nombreux messages de type :

```
ERROR: current transaction is aborted,
commands ignored until end of transaction block
```

après la première requête en erreur.

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'
psql -v ON_ERROR_ROLLBACK='on'
```

- de façon interactive dans psql:

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

1.3.3 PG_RESTORE

- Restaure uniquement les sauvegardes au format binaire
  - format autodétecté (`-F` inutile)
- 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.

Si l'option `-C` (`--create`) est utilisée pour demander la création de la base, l'option `-d` indique une base existante, utilisée uniquement pour la connexion initiale nécessaire à la création de la base. Ensuite, une connexion est initiée vers cette base de restauration.

1.3.4 PG_RESTORE - BASE DE DONNÉES

- `-d` : base de données de connexion
- `-C` (`--create`) :
  - connexion (`-d`) et `CREATE DATABASE`
  - connexion à la nouvelle base et exécute le SQL

Avec `pg_restore`, il est indispensable de fournir le nom de la base de données de connexion avec l'option `-d`.

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

L'option `--create` (ou `C`) permet de créer la base de données cible. Dans ce cas l'option `-d` doit indiquer une base de données existante afin que `pg_restore` se connecte pour exécuter l'ordre `CREATE DATABASE`. Après cela, il se connecte à la base nouvellement créée pour exécuter les ordres SQL de restauration. Pour vérifier cela, on peut lancer la commande sans l'option `-d`. En observant le code SQL renvoyé on remarque un `\connect` :

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

```
--
-- PostgreSQL database dump
--

SET statement_timeout = 0;
SET lock_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SET check_function_bodies = false;
SET client_min_messages = warning;

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

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


ALTER DATABASE b1 OWNER TO postgres;

\connect b1

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

Rappelons que cette option ne récupère la configuration spécifique de la base (paramétrage et droits) que pour une sauvegarde effectuée par un outil client à partir de la version 11.

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

1.3.5 PG_RESTORE - FICHIER OU ENTRÉE STANDARD

- `-f` : fichier SQL
- Ou sortie standard : défaut si < v12, sinon `-f -`
- Fichier à restaurer en dernier argument de la ligne de commande
- Attention à ne pas écraser la sauvegarde !

L'option `-f` envoie le SQL généré dans un script, qui sera donc du SQL parfaitement lisible :

```
$ pg_restore base.dump -f script_restauration.sql
```

`-f` n'indique **pas** le fichier de sauvegarde, mais bien la sortie de `pg_restore` quand on ne restaure pas vers une base. N'écrasez pas votre sauvegarde !

Avec `-f -`, le SQL transmis au serveur est affiché sur la sortie standard, ce qui est très pratique pour voir ce qui va être restauré, et par exemple valider les options d'une restauration partielle, récupérer des définitions d'index ou de table, voire « piper » le contenu vers un autre outil.

Avant la version 12, c'était même le défaut, si ni `-d` ni `-f` n'étaient précisés. Il était alors conseillé de rediriger la sortie standard plutôt que d'utiliser `-f` pour éviter toute ambiguïté.

À partir de la version 12, `pg_restore` exige soit `-d`, soit `-f` : soit on restaure dans une base, soit on génère un fichier SQL.

Pour obtenir le journal d'activité complet d'une restauration, il suffit classiquement de rediriger la sortie :

```
$ pg_restore -d cible --verbose base.dump > restauration.log 2>&1
```

1.3.6 PG_RESTORE - STRUCTURE OU DONNÉES ?

- `--schema-only` : uniquement la structure
- `--data-only` : uniquement les données

ou :

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

Comme pour `pg_dump`, il est possible de ne restaurer que la structure, ou que les données.

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.

Dans les cas un peu délicats (modification des fichiers, imports partiels), on peut vouloir traiter séparément chaque étape. Par exemple, si l'on veut modifier le SQL (modifier des noms de champs, renommer des index…) tout en tenant à compresser ou paralléliser la sauvegarde pour des raisons de volume :

```
$ mkdir data.dump
$ pg_dump -d source --section=pre-data -f predata.sql
$ pg_dump -d source --section=data -Fd --jobs=8 -f data.dump
$ pg_dump -d source --section=post-data -f postdata.sql
```

Après modification, on réimporte :

```
$ psql -d cible < predata.sql
$ pg_restore -d cible --jobs=8  data.dump
$ psql -d cible < postdata.sql
```

Le script issu de `--section=pre-data` (ci-dessous, allégé des commentaires) contient les `CREATE TABLE`, les contraintes de colonne, les attributions de droits mais aussi les fonctions, les extensions, etc. :

```
SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;

CREATE EXTENSION IF NOT EXISTS plperl WITH SCHEMA pg_catalog;
COMMENT ON EXTENSION plperl IS 'PL/Perl procedural language';

CREATE EXTENSION IF NOT EXISTS pg_stat_statements WITH SCHEMA public;
COMMENT ON EXTENSION pg_stat_statements
 IS 'track execution statistics of all SQL statements executed';

CREATE FUNCTION public.impair(i integer) RETURNS boolean
    LANGUAGE sql IMMUTABLE
    AS $$
select mod(i,2)=1 ;
$$;

ALTER FUNCTION public.impair(i integer) OWNER TO postgres;

SET default_tablespace = '';
SET default_table_access_method = heap;

CREATE TABLE public.fils (
    i integer,
    CONSTRAINT impair_ck CHECK ((public.impair(i) IS TRUE)),
```

```
    CONSTRAINT nonzero_ck CHECK ((i > 0))
);
ALTER TABLE public.fils OWNER TO postgres;

CREATE TABLE public.pere (
    i integer NOT NULL
);

ALTER TABLE public.pere OWNER TO postgres;
```

La partie `--section=data`, compressée ou non, ne contient que des ordres `COPY` :

```
# lecture du fichier data.dump sur la sortie standard (-)
$ pg_restore -f - data.dump

SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;

COPY public.fils (i) FROM stdin;
1
2
…
\.
COPY public.pere (i) FROM stdin;
1
2
…
\.
```

Quant au résultat de `--section=pre-data`, il regroupe notamment les contraintes de clés primaire, de clés étrangères, et les créations d'index. Il est nettement plus rapide de charger la table avant de poser contraintes et index que l'inverse.

```
SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
```

```
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;
SET default_tablespace = '';

ALTER TABLE ONLY public.pere
    ADD CONSTRAINT pere_pkey PRIMARY KEY (i);

CREATE UNIQUE INDEX fils_i_idx ON public.fils USING btree (i);

ALTER TABLE ONLY public.fils
    ADD CONSTRAINT fk FOREIGN KEY (i) REFERENCES public.pere(i);
```

1.3.7 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 existe en plus les options `-I` et `-P` (respectivement `--index` et `--function`) pour restaurer respectivement un index et une routine 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. En voici un exemple :

```
$ pg_restore -t t2 -d postgres pouet.dump
$ echo $?
0
```

```
$ pg_restore -t t2 --strict-names -d postgres pouet.dump
pg_restore: [archiver] table "t2" not found
$ echo $?
1
```

1.3.8 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 2020-09-16 15:44:35 CET
;     dbname: b1
;     TOC Entries: 15
;     Compression: -1
;     Dump Version: 1.14-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 13.0
;     Dumped by pg_dump version: 13.0
;
;
; Selected TOC Entries:
;
200; 1255 24625 FUNCTION public f1() postgres
201; 1255 24626 PROCEDURE public p1() postgres
197; 1259 24630 TABLE public t2 postgres
199; 1259 24637 MATERIALIZED VIEW public mv1 postgres
196; 1259 24627 TABLE public t1 postgres
198; 1259 24633 VIEW public v1 postgres
3902; 0 24627 TABLE DATA public t1 postgres
3903; 0 24630 TABLE DATA public t2 postgres
3778; 2606 24642 CONSTRAINT public t2 t2_pkey postgres
3776; 1259 24643 INDEX public t1_c1_idx postgres
```

```
3904; 0 24637 MATERIALIZED VIEW DATA public mv1 postgres
```

Toutes les lignes qui commencent avec un point-virgule sont des commentaires. Le reste indique les objets à créer : un schéma public, le langage `plpgsql`, la procédure stockée `f1`, les tables `t1` et `t2`, la vue `v1`, la clé primaire sur `t2` et l'index sur `t1`. Il indique aussi les données à restaurer avec des lignes du type `TABLE DATA`. Donc, dans cette sauvegarde, il y a les données pour les tables `t1` et `t2`. Enfin, il y a le rafraîchissement des données de la vue matérialisée mv1.

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 PROCEDURE "public.p1()"
pg_restore: creating TABLE "public.t2"
pg_restore: creating MATERIALIZED VIEW "public.mv1"
pg_restore: creating TABLE "public.t1"
pg_restore: creating VIEW "public.v1"
pg_restore: processing data for table "public.t1"
pg_restore: creating CONSTRAINT "public.t2 t2_pkey"
pg_restore: creating MATERIALIZED VIEW DATA "public.mv1"
```

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

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

1.3.9 PG_RESTORE - OPTION DE PARALLÉLISATION

- `-j <nombre_de_threads>`

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.

Cela permet d'améliorer considérablement la vitesse de restauration. Un test effectué a montré qu'une restauration d'une base de 150 Go prenait 5 h avec une seule connexion, mais seulement 3 h avec plusieurs connexions.

Il est possible qu'il n'y ait pas de gain. Par exemple, si une table occupe 15 Go sur une sauvegarde de 20 Go, la parallélisation ne changera pas fondamentalement la durée de restauration, car la table ne sera importée que par une seule connexion.

Il est à noter que, même si la version 9.6 de PostgreSQL apporte la parallélisation de certains types de requêtes, cela ne concerne pas la commande `COPY` de `pg_restore`.

1.3.10 PG_RESTORE - OPTIONS DIVERSES

- `-O` : ignorer le propriétaire
- `-x` : ignorer les droits
- `--no-comments` : ignorer les commentaires
- `--no-tablespaces` : ignorer le tablespace
- `-1` pour tout restaurer en une seule transaction
- `-c` : pour détruire un objet avant de le restaurer

Il reste quelques options plus anecdotiques mais qui peuvent se révéler très pratiques.

Les quatre suivantes (`--no-owner`, `--no-privileges`, `--no-comments` et `--no-tablespaces`) permettent de ne pas restaurer respectivement le propriétaire, les droits, le commentaire et le tablespace 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 `--disable-triggers` est très dangereuse mais peut servir dans certaines situations graves : elle inhibe la vérification des contraintes lors d'un import et peut donc mener à une base incohérente !

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

1.4 AUTRES CONSIDÉRATIONS SUR LA SAUVEGARDE LOGIQUE

- Versions des outils & version du serveur
- Script de sauvegarde
- Sauvegarder sans passer par un fichier
- Statistiques et maintenance après import
- Durée d'exécution d'une sauvegarde
- Taille d'une sauvegarde

La sauvegarde logique est très simple à mettre en place. Mais certaines considérations sont à prendre en compte lors du choix de cette solution : comment gérer les statistiques, quelle est la durée d'exécution d'une sauvegarde, quelle est la taille d'une sauvegarde, etc.

1.4.1 VERSIONS DES OUTILS CLIENTS ET VERSION DE L'INSTANCE

- `pg_dump` : reconnaît les versions de PG antérieures
- `pg_restore`
  - minimum la version du `pg_dump` utilisé
  - si possible celle du serveur cible
- Pas de problème entre OS différents

Il est fréquent de générer une sauvegarde sur une version de PostgreSQL pour l'importer sur un serveur de version différente, en général plus récente. Une différence de version mineure n'est d'habitude pas un problème, ce sont les versions majeures (9.6, 10, 11…) qui importent.

Il faut bien distinguer les versions des instances source et cible, et les versions des outils `pg_dump` et `pg_restore`. Il peut y avoir plusieurs de ces dernières sur un poste. La version sur le poste où l'on sauvegarde peut différer de la version du serveur. Or, leurs formats de sauvegarde `custom` ou `directory` diffèrent.

`pg_dump` sait sauvegarder depuis une instance de version antérieure à la sienne (du moins à partir de PostgreSQL 8.0). Il refusera de tenter une sauvegarde d'une instance de version postérieure. Donc, pour sauvegarder une base PostgreSQL 12, utilisez un `pg_dump` de version 12, 13 ou supérieure.

`pg_restore` sait lire les sauvegardes des versions antérieures à la sienne. Il peut restaurer vers une instance de version supérieure à la sienne, même s'il vaut mieux utiliser le `pg_restore` de la même version que l'instance. La restauration vers une version antérieure a de bonnes chances d'échouer sur une évolution de la syntaxe.

Bien sûr, le format `plain` (SQL pur) est toujours lisible par `psql`, et `pg_restore -f dump.sql` permet toujours d'en regénérer un depuis une sauvegarde `plain` ou `directory`. Il peut même être envoyé directement sans fichier intermédiaire, par exemple ainsi :

```
pg_restore -f dump.sql | psql -h serveurcible -d basecible
```

S'il y a une nouveauté ou une régression que l'instance cible ne sait pas interpréter, il est possible de modifier ce SQL. Dans beaucoup de cas, il suffira d'adapter le SQL dans les parties générées par `--section=pre-data` et `--section=post-data`, et de charger directement les données avec `pg_restore --section=data`.

Enfin, puisqu'il s'agit de sauvegardes logiques, des différences de système d'exploitation ne devraient pas poser de problème de compatibilité supplémentaire.

1.4.2 SCRIPT DE SAUVEGARDE IDÉAL

- `pg_dumpall -g`
- Chaque base : `pg_dump -Fc` ou `-Fd`
- Outils client >= v11
  - sinon reprendre les paramètres des rôles sur les bases
    `ALTER role xxx IN DATABASE xxx SET param=valeur;`
- Bien tester !

`pg_dumpall` n'est intéressant que pour récupérer les objets globaux. Le fait qu'il ne supporte pas les formats binaires entraîne que sa sauvegarde n'est utilisable que dans un cas : la restauration de toute une instance. C'est donc une sauvegarde très spécialisée, ce qui ne conviendra pas à la majorité des cas.

Le mieux est donc d'utiliser `pg_dumpall` avec l'option `-g`, puis d'utiliser `pg_dump` pour sauvegarder chaque base dans un format binaire.

Attention :

- Avant la version 11, les paramètres sur les bases (`ALTER DATABASE xxx SET param=valeur;`) ne seront pas du tout dans les sauvegardes : `pg_dumpall -g` n'exporte pas les définitions des bases (voir `pg_dump --create` plus haut).
- Les paramètres sur les rôles dans les bases figureront dans l'export de `pg_dumpall -g` ainsi :

```
ALTER role xxx IN DATABASE xxx SET param=valeur;
```

mais les bases n'existeront pas forcément au moment où l'`ALTER ROLE` sera exécuté ! Il faudra donc penser à les restaurer à la main…

Voici un exemple de script minimaliste :

```
#!/bin/sh
# Script de sauvegarde pour PostgreSQL

REQ="SELECT datname FROM pg_database WHERE datallowconn ORDER BY datname"

pg_dumpall -g > globals.dump
psql -XAtc "$REQ" postgres | while read base
do
  pg_dump -Fc $base > ${base}.dump
done
```

Évidemment, il ne conviendra pas à tous les cas, mais donne une idée de ce qu'il est possible de faire. (Voir plus bas `pg_back` pour un outil plus complet.)

Exemple de script de sauvegarde adapté pour un serveur Windows :

```
@echo off

SET PGPASSWORD=super_password
SET PATH=%PATH%;C:\Progra~1\PostgreSQL\11\bin\

pg_dumpall -g -U postgres postgres > c:\pg-globals.sql

for /F %%v in ('psql -XAt -U postgres -d cave
             -c "SELECT datname FROM pg_database WHERE NOT datistemplate"') do (
    echo "dump %%v"
    pg_dump -U postgres -Fc %%v > c:\pg-%%v.dump
)
pause
```

Autre exemple plus complet de script de sauvegarde totale de toutes les bases, avec une période de rétention :

```
#!/bin/sh
#------------------------------------------------------------------
#
# Script used to perform a full backup of all databases from a
# PostgreSQL Cluster. The pg_dump use the custom format is done
# into one file per database. There's also a backup of all global
# objects using pg_dumpall -g.
#
# Backup are preserved following the given retention days (default
# to 7 days).
#
# This script should be run daily as a postgres user cron job:
#
# 0 23 * * * /path/to/pg_fullbackup.sh >/tmp/fullbackup.log 2>&1
```

```
#
#------------------------------------------------------------------

# Backup user who must run this script, most of the time it should be postgres
BKUPUSER=postgres
# Number of days you want to preserve your backup
RETENTION_DAYS=7
# Where the backup files should be saved
#BKUPDIR=/var/lib/pgsql/backup
BKUPDIR=/var/lib/postgresql/backup_bases
# Prefix used to prefixe the name of all backup file
PREFIX=bkup
# Set it to save a remote server, default to unix socket
HOSTNAME=""
# Set it to set the remote user to login
USERNAME=""

WHO=`whoami`
if [ "${WHO}" != "${BKUPUSER}" ]; then
        echo "FATAL: you must run this script as ${BKUPUSER} user."
        exit 1;
fi

# Testing backup directory
if [ ! -e "${BKUPDIR}" ]; then
        mkdir -p "${BKUPDIR}"
fi
echo "Begining backup at "`date`

# Set the query to list the available database
REQ="SELECT datname FROM pg_database WHERE datistemplate = 'f'
     AND datallowconn ORDER BY datname"

# Set the date variable to be used in the backup file names
DATE=$(date +%Y-%m-%d_%Hh%M)

# Define the addition pg program options
PG_OPTION=""
if [ $HOSTNAME != "" ]; then
        PG_OPTION="${PG_OPTION} -h $HOSTNAME"
fi;
if [ $USERNAME != "" ]; then
        PG_OPTION="${PG_OPTION} -U $USERNAME"
fi;

# Dumping PostgreSQL Cluster global objects
```

```
echo "Dumping global object into ${BKUPDIR}/${PREFIX}_globals_${DATE}.dump"
pg_dumpall ${PG_OPTION} -g > "${BKUPDIR}/${PREFIX}_globals_${DATE}.dump"
if [ $? -ne 0 ]; then
        echo "FATAL: Can't dump global objects with pg_dumpall."
        exit 2;
fi

# Extract the list of database
psql ${PG_OPTION} -Atc "$REQ" postgres | while read base
# Dumping content of all databases
do
  echo "Dumping database $base into ${BKUPDIR}/${PREFIX}_${base}_${DATE}.dump"
  pg_dump ${PG_OPTION} -Fc $base > "${BKUPDIR}/${PREFIX}_${base}_${DATE}.dump"
  if [ $? -ne 0 ]; then
          echo "FATAL: Can't dump database ${base} with pg_dump."
          exit 3;
  fi
done
if [ $? -ne 0 ]; then
        echo "FATAL: Can't list database with psql query."
        exit 4;
fi

# Starting deletion of obsolete backup files
if [ ${RETENTION_DAYS} -gt 0 ]; then
  echo "Removing obsolete backup files older than ${RETENTION_DAYS} day(s)."
  find ${BKUPDIR}/ -maxdepth 1 -name "${PREFIX}_*" -mtime ${RETENTION_DAYS} \
                   -exec rm -rf '{}' ';'
fi

echo "Backup ending at "`date`

exit 0
```

1.4.3 PG_BACK - PRÉSENTATION

- https://github.com/orgrim/pg_back
- Type de sauvegardes : **logiques** (pg_dump)
- Langage : **bash** (v1) / **go** (v2)
- Licence : **BSD** (libre)
- Type de stockage : **local** + export cloud
- Planification : **crontab**
- **Unix/Linux** (v1 & 2) / **Windows (v2)**

- Compression : via pg_dump
- Versions compatibles : **toutes**
- Rétention : **durée**

pg_back[5] a été écrit par Nicolas Thauvin, consultant de Dalibo, également auteur original de pitrery[6] .

Ce programme assez complet vise à gérer le plus simplement possible des sauvegardes logiques (`pg_dump`, `pg_dumpall`), y compris au niveau de la rétention.

La version 1 est en bash, directement utilisable et éprouvée, mais ne sera plus maintenue à terme.

La version 2, parue en 2021, a été réécrite en go. Le binaire est directement utilisable, et permet notamment une configuration différente par base, une meilleure gestion des paramètres de connexion à PostgreSQL et le support de Windows. Pour les versions avant la 11, le script pg_dumpacl[7] est intégré (v2) ou supporté (v1) pour sauvegarder le paramétrage au niveau des bases.

Vous pouvez les obtenir sur le site du projet[8] , tout comme le source[9] .

Les sauvegardes sont aux formats gérés par `pg_dump` : SQL, custom, par répertoire, compressées ou non…

Les anciennes sauvegardes sont automatiquement purgées, et l'on peut en conserver un nombre minimum. La version 2.1 permet en plus de chiffrer les sauvegardes avec une phrase de passe, de générer des sommes de contrôle, et d'exporter vers Azure, Google Cloud ou Amazon S3, ou n'importe quel serveur distant accessible avec `ssh` en SFTP.

L'outil ne propose pas d'options pour restaurer les données : il faut utiliser ceux de PostgreSQL (`pg_restore`, `psql`).

---

[5] https://github.com/orgrim/pg_back
[6] http://dalibo.github.io/pitrery/
[7] https://github.com/dalibo/pg_dumpacl
[8] https://github.com/orgrim/pg_back/tags
[9] https://github.com/orgrim/pg_back/

1.4.4 SAUVEGARDE ET RESTAURATION SANS FICHIER INTERMÉDIAIRE

- `pg_dump -Fp | psql`
- `pg_dump -Ft | pg_restore`
- `pg_dump -Fc | pg_restore`
- Utilisation des options `-h`, `-p`, `-d`
- Attention à la gestion des erreurs !

La duplication d'une base ne demande pas forcément de passer par un fichier intermédiaire. Il est possible de fournir la sortie de `pg_dump` (format `plain` implicite) à `psql` ou `pg_restore`. Par exemple :

```
$ createdb nouvelleb1
$ pg_dump -Fp b1 | psql nouvelleb1
```

Ces deux commandes permettent de dupliquer `b1` dans `nouvelleb1`.

L'utilisation des options `-h`, `-p`, `-d` permet de sauvegarder et restaurer une base sur des instances différentes, qu'elles soient locales ou à distance.

Le gain de temps est appréciable : l'import peut commencer avant la fin de l'export. Comme toujours avec `pg_dump`, les données sont telles qu'au début de la sauvegarde. On épargne aussi la place nécessaire au stockage du backup. Par contre on ne peut pas paralléliser l'export.

Avec `-Fp`, le flux circule en pur texte : il est possible d'intercaler des appels à des outils comme `awk` ou `sed` pour effectuer certaines opérations à la volée (renommage…).

Cette version peut être intéressante :

```
$ pg_dump -Fc  | pg_restore
```

car elle permet de profiter des options propres au format `custom` à commencer par la compression. Il n'y a alors pas besoin d'intercaler des commandes comme `gzip`/`gunzip` pour alléger la charge réseau.

Le point délicat est la gestion des erreurs puisqu'il y a deux processus à surveiller : par exemple dans un script `bash`, on testera le contenu de `${PIPESTATUS[@]}` (et pas seulement `$?`) pour vérifier que tout s'est bien déroulé, et l'on ajoutera éventuellement `set -o pipefail` en début de script.

1.4.5 STATISTIQUES ET MAINTENANCE APRÈS IMPORT

- Statistiques non sauvegardées
  - `ANALYZE` impérativement après une restauration !
- Pour les performances :
  - `VACUUM` (ou `VACUUM ANALYZE`)
- À plus long terme :
  - `VACUUM FREEZE`

Les statistiques sur les données, utilisées par le planificateur pour sélectionner le meilleur plan d'exécution possible, ne font pas partie de la sauvegarde. Il faut donc exécuter un `ANALYZE` après avoir restauré une sauvegarde. Sans cela, les premières requêtes pourraient s'exécuter très mal du fait de statistiques non à jour.

Lancer un `VACUUM` sur toutes les tables restaurées est également conseillé. S'il n'y a pas besoin de les défragmenter, certaines opérations de maintenance effectuées par un `VACUUM` ont un impact sur les performances. En premier lieu, les *hint bits* (« bits d'indice ») de chaque enregistrement seront mis à jour au plus tard à la relecture suivante, et généreront de nombreuses écritures. Un `VACUUM` explicite forcera ces écritures, si possible avant la mise en production de la base. Il créera aussi la *visibility map* des tables, ce qui autorisera les *Index Only Scans*, une optimisation extrêmement puissante, sans laquelle certaines requêtes seront beaucoup plus lentes.

L'`autovacuum` se chargera bien sûr progressivement de tout cela. Il est cependant par défaut bridé pour ne pas gêner les opérations, et pas toujours assez réactif. Jusqu'en PostgreSQL 12 compris, les insertions ne provoquent que l'`ANALYZE`, pas le `VACUUM`, qui peut donc tarder sur les tables un peu statiques.

Il est donc préférable de lancer un `VACUUM ANALYZE` manuel à la fin de la restauration, afin de procéder immédiatement au passage des statistiques et aux opérations de maintenance.

Il est possible de séparer les deux étapes. L'`ANALYZE` est impératif et rapide ; le `VACUUM` est beaucoup plus lent mais peut avoir lieu durant la production si le temps presse.

À plus long terme, dans le cas d'un gros import ou d'une restauration de base, existe un autre danger : le `VACUUM FREEZE`. Les numéros de transaction étant cycliques, l'autovacuum les « nettoie » des tables quand il sont assez vieux. Les lignes ayant été importées en même temps, cela peut générer l'écriture de gros volumes de manière assez soudaine. Pour prévenir cela, lancez un `VACUUM FREEZE` dans une période calme quelques temps après l'import.

1.4.6 DURÉE D'EXÉCUTION

- Difficile à chiffrer
- Dépend de l'activité sur le serveur
- Option `-v`
- Suivre les `COPY`
  - vue `pg_stat_progress_copy` (v14+)

La durée d'exécution d'une sauvegarde et d'une restauration est difficile à estimer. Cela dépend beaucoup de l'activité présente sur le serveur, de la volumétrie, du matériel, etc.

L'option `-v` de `pg_dump` et de `pg_restore` permet de suivre les opérations, action par action (donc la création des objets, mais aussi leur remplissage). Cependant, sans connaître la base sauvegardée ou restaurée, il est difficile de prédire le temps restant pour la fin de l'opération. Nous savons seulement qu'il a fini de traiter tel objet.

Depuis la version 14, il est possible de suivre individuellement les opérations de copie des données, si ces dernières passent par l'instruction `COPY`. Là encore, il est difficile d'en tirer beaucoup d'informations sans bien connaître la base en cours de traitement. Cependant, cela permet de savoir si une table a bientôt fini d'être traitée par le `COPY` en cours. Pour les tables volumineuses, c'est intéressant.

1.4.7 TAILLE D'UNE SAUVEGARDE LOGIQUE

- Difficile à évaluer
- Contenu des index non sauvegardé
  - donc sauvegarde plus petite
- Objets binaires :
  - entre 2 et 4 fois plus gros
  - donc sauvegarde plus grosse

Il est très difficile de corréler la taille d'une base avec celle de sa sauvegarde.

Le contenu des index n'est pas sauvegardé. Donc, sur une base contenant 10 Go de tables et 10 Go d'index, avoir une sauvegarde de 10 Go ne serait pas étonnant. Le contenu des index est généré lors de la restauration.

Par contre, les données des tables prennent généralement plus de place. Un objet binaire est stocké sous la forme d'un texte, soit de l'octal (donc 4 octets), soit de l'hexadécimal (2 octets). Donc un objet binaire prend 2 à 4 fois plus de place dans la sauvegarde que dans la base. Mais même un entier ne va pas avoir la même occupation disque. La valeur 10 ne prend que 2 octets dans la sauvegarde, alors qu'il en prend quatre dans la base. Et

la valeur 1 000 000 prend 7 octets dans la sauvegarde alors qu'il en prend toujours 4 dans la base.

Tout ceci permet de comprendre que la taille d'une sauvegarde n'a pas tellement de lien avec la taille de la base. Il est par contre plus intéressant de comparer la taille de la sauvegarde de la veille avec celle du jour. Tout gros changement peut être annonciateur d'un changement de volumétrie de la base, changement voulu ou non.

1.4.8 AVANTAGES DE LA SAUVEGARDE LOGIQUE

- Simple et rapide
- Sans interruption de service
- Indépendante de la version de PostgreSQL
- Granularité de sélection à l'objet
- Taille réduite
- Ne conserve pas la fragmentation des tables et des index
- Éventuellement depuis un serveur secondaire

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

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

Une restauration de sauvegarde logique est d'ailleurs la méthode officielle de montée de version majeure. Même s'il existe d'autres méthodes de migration de version majeure, elle reste le moyen le plus sûr parce que le plus éprouvé.

Une sauvegarde logique ne contenant que les données utiles, sa taille est généralement beaucoup plus faible que la base de données source, sans parler de la compression qui peut encore réduire l'occupation sur disque. Par exemple, seuls les ordres DDL permettant de créer les index sont stockés, pas leur contenu. Ils sont alors créés de zéro à la restauration.

Les outils permettent de sélectionner très finement les objets sur lesquels on travaille, à la sauvegarde comme à la restauration.

Il est assez courant d'effectuer une sauvegarde logique à partir d'un serveur secondaire (réplica de la production), pour ne pas charger les disques du primaire ; quitte à mettre la réplication du secondaire en pause le temps de la sauvegarde.

1.4.9 INCONVÉNIENTS DE LA SAUVEGARDE LOGIQUE

- Durée : dépendante des données et de l'activité
- Restauration : uniquement au démarrage de l'export
- Efficace si < 200 Go
- Plusieurs outils pour sauvegarder une instance complète
- `ANALYZE`, `VACUUM ANALYZE`, `VACUUM FREEZE` après import

L'un des principaux inconvénients d'une sauvegarde et restauration porte sur la durée d'exécution. Elle est proportionnelle à la taille de la base de données, ou à la taille des objets choisis pour une sauvegarde partielle.

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

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

Comme les objets et leur contenu sont effectivement recréés à partir d'ordres SQL lors de la restauration, on se débarrasse de la fragmentation des tables et index, mais on perd aussi les statistiques de l'optimiseur, ainsi que certaines méta-données des tables. Il est donc nécessaire de lancer ces opérations de maintenance après l'import.

1.5 SAUVEGARDE PHYSIQUE À FROID DES FICHIERS

- Instance arrêtée : sauvegarde cohérente
- Ne pas oublier : journaux, tablespaces, configuration !
- Outils : système, aucun spécifique à PostgreSQL
  - `cp`, `tar`...
  - souvent : `rsync` en 2 étapes : à chaud puis à froid
  - snapshots SAN/LVM (attention à la cohérence)

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 donc arrêter PostgreSQL pour disposer d'une sauvegarde cohérente si la sauvegarde se fait au niveau du système de fichiers.

Il est cependant essentiel d'être attentif aux données qui ne se trouvent pas directement dans le répertoire des données (`PGDATA`), notamment :

- le répertoire des journaux de transactions (`pg_wal`ou `pg_xlog`), qui est souvent placé dans un autre système de fichiers pour gagner en performances : sans lui, la sauvegarde ne sera pas utilisable ;
- les répertoires des tablespaces s'il y en a, sinon une partie des données manquera ;
- les fichiers de configuration (sous `/etc` sous Debian, notamment), y compris ceux des outils annexes à PostgreSQL.

Voici un exemple de sauvegarde (Cent OS 7) :

```
$ systemctl stop postgresql-12
$ tar cvfj data.tar.bz2 /var/lib/pgsql/12/data
$ systemctl start postgresql-12
```

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. et tout outil de sauvegarde utilisant lui-même ces outils.

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

Pour utiliser des snapshots, il faut soit disposer d'un SAN offrant cette possibilité ou bien utiliser la fonctionnalité équivalente de LVM voire du système de fichier. Dans ce cas, la

procédure est la suivante :

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

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

- `rsync` de l'ensemble des fichiers de l'instance PostgreSQL en fonctionnement pour obtenir une première copie (incohérente) ;
- arrêt de l'instance PostgreSQL ;
- `rsync` de l'ensemble des fichiers de l'instance pour ne transférer que les différences ;
- redémarrage de l'instance PostgreSQL.

1.5.1 AVANTAGES DES SAUVEGARDES À FROID

- Simple
- 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.

1.5.2 INCONVÉNIENTS DES SAUVEGARDES À FROID

- 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
  - Réindexation si changement OS

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 *little endian/big endian* doit être respectée.

De plus, des différences entre deux systèmes, et même entre deux versions d'une même distribution, peuvent mener à devoir réindexer toute l'instance[10] .

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.

1.5.3 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. Notez que l'utilisation des options `--delete` et `--checksum` est *fortement conseillée* lors de la deuxième passe, pour rendre la copie totalement fiable.

Voici un exemple de ce cas d'utilisation :

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

[10] https://blog-postgresql.verite.pro/2018/08/30/glibc-upgrade.html

```
$ rsync -a --delete --checksum /var/lib/postgresql /var/lib/postgresql2
$ /etc/init.d/postgresql start
```

1.6 SAUVEGARDE À CHAUD DES FICHIERS PAR SNAPSHOT DE PARTITION

- Avec certains systèmes de fichiers
- Avec LVM
- Avec la majorité des SAN
- Attention : cohérence entre partitions

Certains systèmes de fichiers (principalement les systèmes de fichiers ZFS et 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.

1.7 SAUVEGARDE À CHAUD DES FICHIERS AVEC POSTGRESQL

- PITR : *Point In Time Recovery*
  - nécessite d'avoir activé l'archivage des WAL
  - technique avancée, complexe à mettre en place et à maintenir
  - pas de coupure de service
  - outils dédiés
- `pg_basebackup`
  - sauvegarde ponctuelle

Il est possible de réaliser les sauvegardes de fichiers sans arrêter l'instance (à chaud), sans

perte ou presque de données, et même avec une possible restauration à un point précis dans le temps. Il s'agit cependant 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 journaux de transaction. En général on la mettra en place avec des outils dédiés et éprouvés comme ,pgBackRest[11] barman[12] ou pitrery[13] .

Pour une sauvegarde à chaud ponctuelle au niveau du système de fichiers, on peut utiliser `pg_basebackup`, fourni avec PostgreSQL. Cet outil se base sur le protocole de réplication par flux (*streaming*) et les slots de réplications pour créer une copie des répertoires de données. Son maniement s'est assez simplifié depuis les dernières versions. Il dispose en version 13 d'un outil de vérification de la sauvegarde (`pg_verifybackup`).

En raison de la complexité de ces méthodes, on testera d'autant plus soigneusement la procédure de restauration.

---

[11] https://pgbackrest.org/
[12] https://www.pgbarman.org/
[13] https://dalibo.github.io/pitrery/

1.8 RECOMMANDATIONS GÉNÉRALES

- Prendre le temps de bien choisir sa méthode
- Bien la tester
- Bien tester la restauration
- Et tester régulièrement !
- Ne pas oublier de sauvegarder les fichiers de configuration

---

1.9 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.

1.10 CONCLUSION

- Plusieurs solutions pour la sauvegarde et la restauration
- Sauvegarde/Restauration complète ou partielle
- Toutes cohérentes
- La plupart à chaud
- Méthode de sauvegarde avancée : PITR

PostgreSQL propose plusieurs méthodes de sauvegardes et de restaurations. Elles ont chacune leurs avantages et leurs inconvénients. Cependant, elles couvrent à peu près tous les besoins : sauvegardes et restaurations complètes ou partielles, sauvegardes cohérentes, sauvegardes à chaud comme à froid.

La méthode de sauvegarde avancée dite *Point In Time Recovery*, ainsi que les dernières sophistications du moteur en matière de réplication par streaming, permettent d'offrir les meilleures garanties afin de minimiser les pertes de données, tout en évitant toute coupure de service liée à la sauvegarde. Il s'agit de techniques avancées, beaucoup plus complexes à mettre en place et à maintenir que les méthodes évoquées précédemment.

1.10.1 QUESTIONS

N'hésitez pas, c'est le moment !

1.11 QUIZ

https://dali.bo/i1_quiz

1.12 TRAVAUX PRATIQUES

1.12.1 SAUVEGARDES LOGIQUES

But : Sauvegarder des bases

> Créer un répertoire `backups` dans le répertoire `home` de **postgres** pour y déposer les fichiers d'export. À l'aide de `pg_dumpall`, sauvegarder toutes les bases de données de l'instance PostgreSQL dans le fichier `~postgres/backups/base_all.sql`.

> Quelle est la taille de la sauvegarde ?

> Avec quel outil consulter le contenu du fichier de sauvegarde ?

> Ouvrir le fichier de sauvegarde. Quelles en sont les principales parties ?

> Recommencer la sauvegarde en compressant la sauvegarde avec `gzip`. Quelle en est la taille ?

> Qu'affiche un `pg_dump -d entreprise` ?

> À l'aide de `pg_dump`, sauvegarder la base de données **pgbench** au format *custom* dans le fichier `~postgres/backups/base_pgbench.dump`. Quelle est sa taille ?

> Avec `pg_restore` et son option `-l`, consulter le contenu de `~postgres/backups/base_pgbench.dump`.

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

Sauvegarder la table `pgbench_accounts` dans le fichier `~postgres/backups/table_pgbench_acco` au format *plain text*. Que contient ce fichier ?

(Optionnel) Créer un script nommé `sauvegarde_tables.sh` qui sauvegarde toutes les tables `pgbench_*` du schéma `public` au format *plain text* de la base **pgbench** dans des fichiers séparés nommés `table_<nom_table>.sql`. La première étape consiste à lister les tables du schéma. Ensuite on boucle sur chacune. Quel est le problème de principe d'un tel script s'il est exécuté sur une base active ?

1.12.2 RESTAURATIONS LOGIQUES

But : Restaurer des sauvegardes

Que se passe-t-il si l'on exécute
`pg_restore ~postgres/backups/base_pgbench.dump -f -` ?

Créer une nouvelle base **pgbench2** appartenant au rôle **testperf**. Ajuster `.pgpass` et `pg_hba.conf` pour que **testperf** puisse se connecter à toute base par son mot de passe.

Restaurer le contenu de la base de donnée **pgbench** dans **pgbench2** en utilisant le fichier de sauvegarde `base_pgbench.dump`. Toutes les tables sont-elles restaurées à l'identique ?

Puis renommer la base de données **pgbench** en **pgbench_old** et la base de données **pgbench2** en **pgbench**.

Dans la base **pgbench**, détruire la table `pgbench_history`.
À partir de la sauvegarde de la base **pgbench** au format *custom* faite précédemment, restaurer uniquement la table `pgbench_history` dans la nouvelle base **pgbench**, avec `pg_restore -t`. La table est-elle complète ?

> Rechercher les morceaux manquants dans la section `post-data` de la sauvegarde.

> Dans une fenêtre, lancer une instance `pgbench` sur la base **pgbench** :
> `/usr/pgsql-14/bin/pgbench -U testperf -T 600 pgbench`

> Créer une nouvelle base **pgbench3** appartenant à **testperf**.
> Avec `pg_dump`, copier les données de la base **pgbench** dans cette nouvelle base **pgbench3** sans passer par un fichier de sauvegarde.

1.12.3 SAUVEGARDE ET RESTAURATION PARTIELLE

But : Choisir les données à restaurer

> Il faut restaurer dans une base **pgbench4** tout le contenu de la sauvegarde de la base **pgbench**, sauf les *données* de la table `pgbench_accounts` ni les tables `copie*`. La définition de la table `pgbench_accounts` et toutes les contraintes s'y rapportant doivent être restaurées.
> Pour cela, utiliser les options `-l` de `pg_restore` pour créer le fichier des objets à charger, fichier que l'on modifiera, et `-L` pour l'utiliser.

1.12.4 (OPTIONNEL) SAUVEGARDE ET RESTAURATION PAR PARTIES AVEC MODIFICATION

But : Manipuler les données à restaurer

> Effectuer une sauvegarde de **pgbench_old**, puis une restauration vers une nouvelle base **test**, lors de laquelle on ne rechargera que les tables `pgbench_*` en les renommant en `test_*`. Pour cela, créer plusieurs fichiers (format *plain*) grâce aux options `--section=` de `pg_dump`, modifier les fichiers obtenus, et les charger.
> Vérifier les tables sont bien là avec les nouveaux noms.

> Recommencer avec une sauvegarde des données en format *custom* en scriptant les modifications à la volée.

1.13 TRAVAUX PRATIQUES (SOLUTIONS)

1.13.1 SAUVEGARDES LOGIQUES

Répertoire cible

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

En tant qu'utilisateur **postgres** :

```
$ cd ~postgres
$ mkdir backups
```

Sauvegarde logique de toutes les bases

> À l'aide de `pg_dumpall`, sauvegarder toutes les bases de données de l'instance PostgreSQL dans le fichier `~postgres/backups/base_all.sql`.

En tant qu'utilisateur **postgres**, puis exécuter la commande suivante :

```
$ pg_dumpall > ~postgres/backups/base_all.sql
```

> Quelle est la taille de la sauvegarde ?

```
$ ls -lh ~postgres/backups/
total 37M
-rw-r--r-- 1 postgres postgres 37M 1 août 17:17 base_all.sql
```

> Avec quel outil consulter le contenu du fichier de sauvegarde ?

Ne jamais ouvrir un fichier potentiellement volumineux avec un éditeur comme `vi` car il pourrait alors consommer des gigaoctets de mémoire. Sous Unix, `less` convient en général :

```
$ less ~postgres/backups/base_all.sql
```

> Ouvrir le fichier de sauvegarde. Quelles en sont les principales parties ?

Le fichier contient en clair des ordres SQL :

- d'abord du paramétrage destiné à la restauration :

```
SET default_transaction_read_only = off;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
```

- la déclaration des rôles avec des mots de passe hachés, et les affectations des groupes :

```
CREATE ROLE boulier;
ALTER ROLE boulier WITH NOSUPERUSER INHERIT
NOCREATEROLE NOCREATEDB LOGIN NOREPLICATION NOBYPASSRLS
PASSWORD 'SCRAM-SHA-256$4096:ksvTos26fH+0qUov4zssLQ==$un1…';
…
GRANT secretariat TO boulier GRANTED BY postgres;
GRANT secretariat TO tina GRANTED BY postgres;
```

- des ordres de création des bases :

```
CREATE DATABASE entreprise WITH TEMPLATE = template0 ENCODING = 'UTF8'
LOCALE = 'en_GB.UTF-8';

ALTER DATABASE entreprise OWNER TO patron;
```

- les ordres pour se connecter à chaque base et y déclarer des objets :

```
\connect entreprise
…
CREATE TABLE public.brouillon (
    id integer,
    objet text,
    creationts timestamp without time zone
);
ALTER TABLE public.brouillon OWNER TO boulier;
…
```

- les données elles-mêmes, sous formes d'ordres `COPY` :

```
COPY public.produit (appellation) FROM stdin;
Gewurtzraminer vendanges tardives
Cognac
Eau plate
Eau gazeuse
Jus de framboise
\.
```

- ainsi que divers ordres comme des contraintes ou des créations d'index :

```
ALTER TABLE ONLY public.pgbench_accounts
    ADD CONSTRAINT pgbench_accounts_pkey PRIMARY KEY (aid);
```

> Recommencer la sauvegarde en compressant la sauvegarde avec `gzip`. Quelle en est la taille ?

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

$ ls -lh base_all.sql.gz
-rw-r--r-- 1 postgres postgres 6,0M  1 août  17:39 base_all.sql.gz
```

Le taux de compression de 80 % n'est pas inhabituel, mais il dépend bien entendu énormément des données.

Sauvegarde logique d'une base

> Qu'affiche un `pg_dump -d entreprise` ?

Le résultat est un script, en clair, de création des objets de la base **entreprise**, sans ordre de création de la base ni d'utilisateur.

Notamment, les `GRANT` suppose la pré-existence des rôles adéquats.

> À l'aide de `pg_dump`, sauvegarder la base de données **pgbench** au format *custom* dans le fichier `~postgres/backups/base_pgbench.dump`. Quelle est sa taille ?

En tant qu'utilisateur **postgres**, puis exécuter la commande suivante :

```
$ pg_dump -Fc -f ~postgres/backups/base_pgbench.dump bench
```

La taille de 6 Mo environ indique que cette base contenait l'essentiel de la volumétrie de l'instance, et que la sauvegarde *custom* est bien compressée.

> Avec `pg_restore` et son option `-l`, consulter le contenu de `~postgres/backups/base_pgbench.dump`.

```
$ pg_restore -l ~postgres/backups/base_pgbench.dump
;
; Archive created at 2021-11-22 16:36:31 CET
;     dbname: pgbench
;     TOC Entries: 27
;     Compression: -1
;     Dump Version: 1.14-0
;     Format: CUSTOM
```

```
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 14.1
;     Dumped by pg_dump version: 14.1
;
;
; Selected TOC Entries:
;
200; 1259 18920 TABLE public copie1 postgres
201; 1259 18923 TABLE public copie2 postgres
202; 1259 18928 TABLE public copie3 postgres
198; 1259 18873 TABLE public pgbench_accounts testperf
199; 1259 18876 TABLE public pgbench_branches testperf
196; 1259 18867 TABLE public pgbench_history testperf
197; 1259 18870 TABLE public pgbench_tellers testperf
3712; 0 18920 TABLE DATA public copie1 postgres
3713; 0 18923 TABLE DATA public copie2 postgres
3714; 0 18928 TABLE DATA public copie3 postgres
3710; 0 18873 TABLE DATA public pgbench_accounts testperf
3711; 0 18876 TABLE DATA public pgbench_branches testperf
3708; 0 18867 TABLE DATA public pgbench_history testperf
3709; 0 18870 TABLE DATA public pgbench_tellers testperf
3579; 2606 18888 CONSTRAINT public pgbench_accounts pgbench_accounts_pkey testperf
3581; 2606 18884 CONSTRAINT public pgbench_branches pgbench_branches_pkey testperf
3577; 2606 18886 CONSTRAINT public pgbench_tellers pgbench_tellers_pkey testperf
3586; 2606 18894 FK CONSTRAINT public pgbench_accounts pgbench_accounts_bid_fkey…
3584; 2606 18909 FK CONSTRAINT public pgbench_history pgbench_history_aid_fkey…
3582; 2606 18899 FK CONSTRAINT public pgbench_history pgbench_history_bid_fkey…
3583; 2606 18904 FK CONSTRAINT public pgbench_history pgbench_history_tid_fkey…
3585; 2606 18889 FK CONSTRAINT public pgbench_tellers pgbench_tellers_bid_fkey…
```

Export des objets globaux

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

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

Le contenu du fichier est très réduit et ne reprend que les premiers éléments du fichier créé plus haut.

Sauvegarde logique de tables

> Sauvegarder la table `pgbench_accounts` dans le fichier `~postgres/backups/table_pgbench_acco` au format *plain text*. Que contient ce fichier ?

```
$ pg_dump -t pgbench_accounts pgbench > ~postgres/backups/table_pgbench_accounts.sql
```

Le contenu se consulte avec :

```
less  ~postgres/backups/table_pgbench_accounts.sql
```

On y trouvera entre autre les ordres de création :

```
CREATE TABLE public.pgbench_accounts (
    aid integer NOT NULL,
    ...
)
WITH (fillfactor='100');
```

l'affectation du propriétaire :

```
ALTER TABLE public.pgbench_accounts OWNER TO testperf;
```

les données :

```
COPY public.pgbench_accounts (aid, bid, abalance, filler) FROM stdin;
35      1       0
38      1       -11388
18      1       7416
7       1       -1124
...
```

Tout à la fin, les contraintes :

```
ALTER TABLE ONLY public.pgbench_accounts
    ADD CONSTRAINT pgbench_accounts_pkey PRIMARY KEY (aid);

ALTER TABLE ONLY public.pgbench_accounts
    ADD CONSTRAINT pgbench_accounts_bid_fkey FOREIGN KEY (bid)
    REFERENCES public.pgbench_branches(bid);
```

> (Optionnel) Créer un script nommé `sauvegarde_tables.sh` qui sauvegarde toutes les tables `pgbench_*` du schéma `public` au format *plain text* de la base **pgbench** dans des fichiers séparés nommés `table_<nom_table>.sql`. La première étape consiste à lister les tables du schéma. Ensuite on boucle sur chacune. Quel est le problème de principe d'un tel script s'il est exécuté sur une base active ?

En tant qu'utilisateur système **postgres**, créer le script avec un éditeur de texte comme `vi` :

```
$ vi ~postgres/backups/sauvegarde_tables.sh
```

Ce script pourrait être le suivant :

```
#!/bin/bash
set -xue  # pour débogage et arrêt immédiat en cas d'erreur
LIST_TABLES=$(psql -At -c '\dt public.pgbench_*' -d pgbench | \
              awk -F'|' '{print $2}')
for t in $LIST_TABLES ; do
  echo "Sauvegarde $t..."
  pg_dump -t $t pgbench > ~postgres/backups/table_${t}.sql
done
```

Autre solution possible, plus lisible, plus flexible et incluant le schéma dans le nom des fichiers :

```
#!/bin/bash
set -xue
export PGDATABASE=pgbench
export PGUSER=postgres
export PGPORT=5432
export PGHOST=/var/run/postgresql
for t in $(psql -At -F'.' -c "SELECT schemaname, tablename
                              FROM pg_tables
                              WHERE schemaname = 'public'
                              AND tablename LIKE 'pgbench_%' " )
do
  pg_dump -t "$t" pgbench > ~postgres/backups/table_$t.sql
done
```

Enfin, il faut donner les droits d'exécution au script :

```
$ chmod +x ~postgres/backups/sauvegarde_tables.sh
```

Et on teste :

```
$ ~postgres/backups/sauvegarde_tables.sh
```

Ce genre de script possède un problème fondamental : les sauvegardes sont faites séparément et la cohérence entre les tables exportées n'est plus du tout garantie. Cela peut réserver de très mauvaises surprises à la restauration. Il faudrait pour le moins utiliser un même ordre `pg_dump` avec plusieurs paramètres `t`. `pg_dump` pare à cela en utilisant une transaction en *repeatable read*.

1.13.2 RESTAURATIONS LOGIQUES

Restauration d'une base

> Que se passe-t-il si l'on exécute
> `pg_restore ~postgres/backups/base_pgbench.dump -f -` ?

La sortie affiche les ordres qui seront envoyés lors de la restauration. C'est très pratique pour consulter le contenu d'une sauvegarde compressée avant de l'exécuter.

`-f -` envoie sur la sortie standard. Pour envoyer vers un fichier, demander `-f dump.sql`, et l'on obtiendra le SQL contenu dans la sauvegarde.

À partir de la version 12, `pg_restore` exige d'avoir soit un fichier (`-f`) soit une base (`-d`) en destination.

> Créer une nouvelle base **pgbench2** appartenant au rôle **testperf**. Ajuster `.pgpass` et `pg_hba.conf` pour que **testperf** puisse se connecter à toute base par son mot de passe.

La connexion se fait par exemple en ajoutant le mot de passe dans le `.pgpass` si ce n'est déjà fait :

```
localhost:5432:*:testperf:860e74ea6eba6fdee4574c54aadf4f98
```

ainsi que ceci en tête du `pg_hba.conf` (ne pas oublier de recharger la configuration) :

```
local   all             testperf                                scram-sha-256
```

En tant qu'utilisateur **postgres** :

```
$ createdb -O testperf pgbench2
$ psql -U testperf -d pgbench2 -c "SELECT 'test de connexion' "
```

> Restaurer le contenu de la base de donnée **pgbench** dans **pgbench2** en utilisant le fichier de sauvegarde `base_pgbench.dump`. Toutes les tables sont-elles restaurées à l'identique ?

```
$ pg_restore -v -U testperf -d pgbench2 ~postgres/backups/base_pgbench.dump
```

L'option `-v` est optionnelle. Elle permet d'avoir le détail suivant :

```
pg_restore: connecting to database for restore
pg_restore: creating TABLE "public.copie1"
```

```
pg_restore: [archiver (db)] Error while PROCESSING TOC:
pg_restore: [archiver (db)] Error from TOC entry 200;1259 18920
                            TABLE copie1 postgres
pg_restore: [archiver (db)] could not execute query:
    ERROR:  must be member of role "postgres"
    Command was: ALTER TABLE public.copie1 OWNER TO postgres;
…
pg_restore: creating TABLE "public.pgbench_accounts"
pg_restore: creating TABLE "public.pgbench_branches"
pg_restore: creating TABLE "public.pgbench_history"
pg_restore: creating TABLE "public.pgbench_tellers"
…
pg_restore: processing data for table "public.pgbench_history"
pg_restore: processing data for table "public.pgbench_tellers"
pg_restore: creating CONSTRAINT "public.pgbench_accounts pgbench_accounts_pkey"
pg_restore: creating FK CONSTRAINT "public.pgbench_accounts
                                    pgbench_accounts_bid_fkey"
…
```

Un simple `\d+` sur les bases **pgbench** et **pgbench2** montre deux différences :

- la taille des tables n'est pas forcément la même (bien que les données soient identiques) car l'organisation sur le disque est plus compacte sur une base qui n'a pas de vécu ;
- les propriétaires des tables `copie*` ne sont pas les mêmes : en effet, on voit ci-dessus que les ordres `ALTER TABLE` échouent : **testperf** n'a pas les droits suffisants pour changer le propriétaire des tables. L'option `--no-owner` permet d'éviter ces messages.

> Puis renommer la base de données **pgbench** en **pgbench_old** et la base de données **pgbench2** en **pgbench**.

En tant qu'utilisateur **postgres** connecté à la base de même nom :

```
postgres@postgres=# ALTER DATABASE pgbench RENAME TO pgbench_old;
postgres@postgres=# ALTER DATABASE pgbench2 RENAME TO pgbench;
```

Les ordres peuvent échouer si une connexion est toujours en cours à la base.

```
postgres@postgres=# \l
                                  List of databases
   Name    |  Owner   | Encoding |  Collate   |   Ctype    | Access privileges
-----------+----------+----------+------------+------------+-------------------
entreprise | patron   | UTF8     |en_GB.UTF-8|en_GB.UTF-8|
pgbench    | testperf | UTF8     |en_GB.UTF-8|en_GB.UTF-8|
```

```
pgbench_old| postgres | UTF8     |en_GB.UTF-8|en_GB.UTF-8|
postgres   | postgres | UTF8     |en_GB.UTF-8|en_GB.UTF-8|
template0  | postgres | UTF8     |en_GB.UTF-8|en_GB.UTF-8|=c/postgres         +
           |          |          |           |           |postgres=CTc/postgres
template1  | postgres | UTF8     |en_GB.UTF-8|en_GB.UTF-8|=c/postgres         +
           |          |          |           |           |postgres=CTc/postgres
```

Restauration d'une table

> Dans la base **pgbench**, détruire la table `pgbench_history`.
> À partir de la sauvegarde de la base **pgbench** au format *custom* faite précédemment, restaurer uniquement la table `pgbench_history` dans la nouvelle base **pgbench**, avec `pg_restore -t`. La table est-elle complète ?
> Rechercher les morceaux manquants dans la section `post-data` de la sauvegarde.

```
$ psql -U testperf -d pgbench
testperf@pgbench=> \d pgbench_history
                        Table « public.pgbench_history »
 Colonne |            Type             | Collationnement | NULL-able | Par défaut
---------+-----------------------------+-----------------+-----------+------------
 tid     | integer                     |                 |           |
 bid     | integer                     |                 |           |
 aid     | integer                     |                 |           |
 delta   | integer                     |                 |           |
 mtime   | timestamp without time zone |                 |           |
 filler  | character(22)               |                 |           |
Contraintes de clés étrangères :
    "pgbench_history_aid_fkey" FOREIGN KEY (aid) REFERENCES pgbench_accounts(aid)
    "pgbench_history_bid_fkey" FOREIGN KEY (bid) REFERENCES pgbench_branches(bid)
    "pgbench_history_tid_fkey" FOREIGN KEY (tid) REFERENCES pgbench_tellers(tid)


testperf@pgbench=> DROP TABLE pgbench_history ;
DROP TABLE

$ pg_restore -d pgbench -U testperf -t pgbench_history \
                                    ~postgres/backups/base_pgbench.dump
```

Les données sont bien là mais les contraintes n'ont *pas* été restaurées : en effet, `pg_restore -t` ne considère que les tables proprement dites.

```
postgres@pgbench=# \d pgbench_history
                        Table « public.pgbench_history »
 Colonne |            Type             | Collationnement | NULL-able | Par défaut
```

```
---------+-----------------------------+----------------+-----------+------------
 tid     | integer                     |                |           |
 bid     | integer                     |                |           |
 aid     | integer                     |                |           |
 delta   | integer                     |                |           |
 mtime   | timestamp without time zone |                |           |
 filler  | character(22)               |                |           |
```

La manière la plus propre de récupérer une table depuis une sauvegarde est la technique avec `-L` décrite plus loin.

Une autre possibilité est d'avoir précédemment écrit un script de sauvegarde table à table comme ci-dessus. `pg_dump -t`, lui, sauvegarde les contraintes des tables.

Une autre possibilité est de retrouver les ordres manquants dans la sauvegarde. Ils sont dans la section `post-data` :

```
$ pg_restore --section=post-data  ~postgres/backups/base_pgbench.dump
```

On a donc une version en SQL de la fin d'une sauvegarde, et on peut y piocher les trois ordres `ALTER TABLE ... ADD CONSTRAINT ...` concernant la table.

Migration de données

> Dans une fenêtre, lancer une instance `pgbench` sur la base **pgbench** :
>
> `/usr/pgsql-14/bin/pgbench -U testperf -T 600 pgbench`

Cet ordre va modifier la base durant la sauvegarde suivante.

> Créer une nouvelle base **pgbench3** appartenant à **testperf**.
> Avec `pg_dump`, copier les données de la base **pgbench** dans cette nouvelle base **pgbench3** sans passer par un fichier de sauvegarde.

```
$ createdb -O testperf  pgbench3
$ pg_dump -Fp  -U testperf pgbench | psql -U testperf -d pgbench3
```

Cet ordre ne doit pas générer d'erreur bien que les données soient modifiées pendant le backup. L'état des données restaurées sera celui au moment du début du `pg_dump`.

Le format *custom* est possible aussi, mais il utilise une compression ici inutile :

```
$ pg_dump -Fc  -U testperf pgbench | pg_restore -U testperf -d pgbench3
```

1.13.3 SAUVEGARDE ET RESTAURATION PARTIELLE

> Il faut restaurer dans une base **pgbench4** tout le contenu de la sauvegarde de la base **pgbench**, sauf les *données* de la table `pgbench_accounts` ni les tables `copie*`. La définition de la table `pgbench_accounts` et toutes les contraintes s'y rapportant doivent être restaurées.
> Pour cela, utiliser les options `-l` de `pg_restore` pour créer le fichier des objets à charger, fichier que l'on modifiera, et `-L` pour l'utiliser.

On procède en deux étapes :

- lister le contenu de l'archive dans un fichier :

```
$ pg_restore -l ~postgres/backups/base_pgbench.dump > /tmp/contenu_archive
```

- dans `/tmp/contenu_archive`, supprimer ou commenter la ligne `TABLE DATA` de la table `pgbench_accounts` et tout ce qui peut concerner des tables `copie*` :

```
;200; 1259 18920 TABLE public copie1 postgres
;201; 1259 18923 TABLE public copie2 postgres
;202; 1259 18928 TABLE public copie3 postgres
...
;3712; 0 18920 TABLE DATA public copie1 postgres
;3713; 0 18923 TABLE DATA public copie2 postgres
;3714; 0 18928 TABLE DATA public copie3 postgres
;3710; 0 18873 TABLE DATA public pgbench_accounts testperf
```

- Créer la base de données **pgbench4** appartenant à **testperf**, puis restaurer en utilisant ce fichier :

```
$ createdb -O testperf  pgbench4
$ pg_restore -U testperf -d pgbench4 -L /tmp/contenu_archive \
                                      ~postgres/backups/base_pgbench.dump
```

S'il y a une contrainte vers la table `pgbench_accounts`, il est alors normal qu'elle échoue. Il y aura simplement un message d'erreur. Le plus propre aurait été de l'exclure d'entrée dans le fichier plus haut.

1.13.4 (OPTIONNEL) SAUVEGARDE ET RESTAURATION PAR PARTIES AVEC MODIFICATION

> Effectuer une sauvegarde de **pgbench_old**, puis une restauration vers une nouvelle base **test**, lors de laquelle on ne rechargera que les tables `pgbench_*` en les renommant en `test_*`. Pour cela, créer plusieurs fichiers (format *plain*) grâce aux options `--section=` de `pg_dump`, modifier les fichiers obtenus, et les charger.
> Vérifier les tables sont bien là avec les nouveaux noms.

On peut séparer la sauvegarde en 3 fichiers texte :

```
$ pg_dump -Fp -U testperf -d pgbench_old -t 'pgbench_*' --section=pre-data \
                                    -f ~postgres/backups/base_pgbench_old-1.sql
$ pg_dump -Fp -U testperf -d pgbench_old -t 'pgbench_*' --section=data \
                                    -f ~postgres/backups/base_pgbench_old-2.sql
$ pg_dump -Fp -U testperf -d pgbench_old -t 'pgbench_*' --section=post-data \
                                    -f ~postgres/backups/base_pgbench_old-3.sql
```

Le premier contient les déclarations d'objets. Supprimer les tables non voulues s'il y en a. Renommer les tables `pgbench_*` en `test_*`.

Procéder de même dans le troisième fichier, qui contient les contraintes et les index.

Le second contient les données elles-mêmes. Modifier les noms de table ici aussi. Heureusement le fichier est encore assez peu volumineux pour permettre une modification manuelle.

Noter que ces fichiers forment une sauvegarde cohérente pour peu qu'il n'y ait pas eu d'ordres DDL pendant leur création.

Restaurer les fichiers l'un après l'autre :

```
$ createdb -O testperf test

$ psql -U testperf -d test < ~postgres/backups/base_pgbench_old-1.sql
...
$ psql -U testperf -d test < ~postgres/backups/base_pgbench_old-2.sql
...
COPY 100000
COPY 1
COPY 38051
COPY 10

$ psql -U testperf -d test < ~postgres/backups/base_pgbench_old-3.sql
...
```

```
ALTER TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
```

Vérifier le résultat :

```
$ psql -d test -c '\d+'
                          List of relations
 Schema |     Name      | Type  |  Owner   |    Size    | Description
--------+---------------+-------+----------+------------+-------------
 public | test_accounts | table | testperf | 13 MB      |
 public | test_branches | table | testperf | 8192 bytes |
 public | test_history  | table | testperf | 1968 kB    |
 public | test_tellers  | table | testperf | 8192 bytes |
(4 rows)
```

> Recommencer avec une sauvegarde des données en format *custom* en scriptant les modifications à la volée.

Jusque là il n'y a pas une grande différence avec une restauration depuis un fichier de sauvegarde unique. Les sections sont plus intéressantes s'il y a plus de données et qu'il n'est pas réaliste de les stocker sur le disque dans un fichier texte. Le deuxième fichier sera donc au format *custom* ou *tar* :

```
$ pg_dump -Fc -U testperf -d pgbench_old -t 'pgbench_*' --section=data \
                                -f ~postgres/backups/base_pgbench_old-2.dump
```

Pour le restaurer, il faudra modifier les ordres `COPY` à la volée en sortie de `pg_restore`, sortie qui est réinjectée ensuite dans la cible avec `psql`. Le script ressemble par exemple à ceci :

```
$ pg_restore ~postgres/backups/base_pgbench_old-2.dump -f - | \
    awk '/^COPY/ {gsub("COPY public.pgbench_", "COPY public.test_") ; print }
         ! /^COPY/ {print} ' | \
    psql -U testperf -d test
```

La technique est utilisable pour toute manipulation des données par script. Pour éviter tout fichier intermédiaire, on peut même appliquer la technique à la sortie d'un `pg_dump -Fp --section=data`.

NOTES

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

- **DBA1 : Administration PostgreSQL**
  https://dali.bo/dba1
- **DBA2 : Administration PostgreSQL avancé**
  https://dali.bo/dba2
- **DBA3 : Sauvegarde et réplication avec PostgreSQL**
  https://dali.bo/dba3
- **DEVPG : Développer avec PostgreSQL**
  https://dali.bo/devpg
- **PERF1 : PostgreSQL Performances**
  https://dali.bo/perf1
- **PERF2 : Indexation et SQL avancés**
  https://dali.bo/perf2
- **MIGORPG : Migrer d'Oracle à PostgreSQL**
  https://dali.bo/migorpg
- **HAPAT : Haute disponibilité avec PostgreSQL**
  https://dali.bo/hapat

LIVRES BLANCS

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

TÉLÉCHARGEMENT GRATUIT

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

DALIBO, L'EXPERTISE POSTGRESQL

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

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

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