

Bases de données : PostgreSQL

- Donner des droits read-only à un utilisateur
- Migration d'une version majeure de PostgreSQL à une autre
- Utiliser Barman pour sauvegarder la base PostgreSQL d'un Gitlab Omnibus

Donner des droits read-only à un utilisateur

Tiré de <https://stackoverflow.com/questions/760210/how-do-you-create-a-read-only-user-in-postgresql/762649#762649>.

Pour donner un droit read-only sur une table :

```
GRANT SELECT ON mytable TO xxx;
```

Pour donner le droit sur toutes les tables :

```
GRANT SELECT ON ALL TABLES IN SCHEMA public TO xxx;
```

Ceci ne donne le droit que sur les tables existantes. Pour que le droit sur toutes les tables, même celles à venir :

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public  
GRANT SELECT ON TABLES TO xxx;
```

Pour supprimer le droit :

```
REVOKE SELECT ON mytable FROM xxx;  
REVOKE SELECT ON ALL TABLES IN SCHEMA public FROM xxx;  
ALTER DEFAULT PRIVILEGES IN SCHEMA public  
REVOKE SELECT ON TABLES FROM xxx;
```

Migration d'une version majeure de PostgreSQL à une autre

NB : instructions pour le passage de PostgreSQL 13 à PostgreSQL 15 (Debian Bookworm). Voir [ici](#) pour de 9.1 à 9.4 (Debian Jessie), [ici](#) pour de 9.4 à 9.6 (Debian Stretch), [ici](#) pour de 9.6 à 11 (Debian Buster) et [ici](#) pour de 11 à 13 (Debian Bullseye).

NB : mettez vous dans un `tmux` avant de commencer la procédure. Prudence est mère de sûreté.

NB : si vous utilisez l'extension PostGis, optez pour la méthode moins rapide. N'oubliez pas d'installer le paquet de l'extension `postgis` pour la nouvelle version de PostgreSQL !

Attention : si vous avez des fichiers de configuration dans `/etc/postgresql/XX/main/conf.d`, ils ne seront pas copiés dans le dossier de configuration de la nouvelle version par `pg_upgrade_cluster` ! Pensez à les copier à la main.

Pré-requis

Vérifiez les paquets PostgreSQL installés sur votre système avec la commande suivante :

```
dpkg -l | grep postgresql
```

Ça vous permettra de voir si vous avez des extensions installées, comme `postgresql-13-repack`, et donc d'installer la version qui va bien pour la nouvelle version de PostgreSQL (en l'occurrence, `postgresql-15-repack`).

Méthode rapide

On stoppe les clusters PostgreSQL

```
systemctl stop postgresql
```

On vire le cluster de la nouvelle version (normalement vide si on vient juste de l'installer : faire gaffe à ne pas laisser passer de temps entre l'installation de la nouvelle version et la migration des données, pour que personne n'utilise le nouveau cluster)

```
pg_dropcluster --stop 15 main
```

On migre les données

```
pg_upgradecluster -m upgrade 13 main
```

ATTENTION

Si vous avez mis des `shared_preload_libraries` dans la configuration de votre ancien cluster, il y a des chances que `pg_upgradecluster -m upgrade 13 main` se foire (mais pas si on utilise la méthode `dump` décrite plus bas).

La solution est simple : créez le répertoire `/etc/postgresql/15/main/conf.d` et mettez-y un fichier dont le nom se termine par `.conf` (genre `shared_preload_libraries.conf`).

Dans ce fichier, mettez la configuration de vos `shared_preload_libraries` et ça devrait être bon.

Il faut savoir que cette commande copie les données de l'ancien cluster vers le nouveau. Il vous faut donc avoir au moins une fois la place de `/var/lib/postgresql/13` de disponible. Un contournement est d'utiliser l'option `--link` qui utilisera des *hard links* plutôt qu'une copie. Par contre, si quelque chose foire, vous foirez votre ancien cluster avec, c'est donc dangereux.

On redémarre le cluster (le 15 pour le coup) :

```
systemctl start postgresql
```

On lance l'analyse du nouveau cluster :

```
sudo -u postgres /usr/lib/postgresql/15/bin/vacuumdb --all --analyze-in-stages
```

Si vous utilisiez des extensions, allez dans `/var/log/postgresql`, vous aurez un dossier qui commence par `pg_upgrade` et qui contiendra un script un autre pour supprimer l'ancien cluster et un autre pour mettre à jour vos extensions. Faites alors (**Attention** : je n'ai pas encore testé cette partie, ça vient des tutos des versions précédentes mais vu que la commande précédente a changé, il est possible que celle-ci aussi) :

```
sudo -u postgres psql -f /var/log/postgresql/pg_upgradecluster-13-15-main*/update_extensions.sql
```

Méthode moins rapide

Cette méthode fait un `pg_dump` et un `pg_restore`. C'est infiniment plus long quand on a de grosses bases de données, mais ça donne un cluster bien propre. Tellement propre que des fois ça foire pour cause de clés dupliquées ?

Vous aurez compris, je n'aime pas tellement cette méthode. Elle a cependant l'avantage d'éviter les problème d'index, vu que ça reconstruit les indexes (ce qui participe à la lenteur de la méthode).

```
systemctl start postgresql  
pg_upgradecluster -m dump 13 main
```

Fin de migration, partie commune aux deux méthodes

On teste les applis qui utilisent PostgreSQL.

Si ça fonctionne, on vire les anciennes données

```
pg_dropcluster 13 main --stop
```

On vire l'ancienne version de PostgreSQL

```
apt-get autoremove --purge postgresql-13 postgresql-client-13
```

[Source](#)

Utiliser Barman pour sauvegarder la base PostgreSQL d'un Gitlab Omnibus

Barman est un super logiciel de sauvegarde d'un *cluster* PostgreSQL au fil de l'eau.

Attention : ça ne sauvegarde pas les bases de données une à une, ça sauvegarde **tout** le *cluster* PostgreSQL. C'est un peu embêtant de devoir remonter un *cluster* entier pour récupérer une base ou juste quelques données mais comme c'est un outil surpuissant qui permet de récupérer ses données à la milliseconde près, il est facile de passer outre cet inconvénient.

Pour le côté « au fil de l'eau », ça veut dire que les modifications sont répliquées du *cluster* PostgreSQL à Barman en temps quasi réel par le biais des WAL.

Il est fort simple de mettre en place la sauvegarde d'un *cluster* PostgreSQL par Barman. Je vous laisse lire la documentation officielle.

Ce tutoriel vise le cas particulier de la sauvegarde du *cluster* PostgreSQL d'un serveur Gitlab installé via les paquets Omnibus. Avec cette méthode d'installation, c'est Gitlab qui installe sa version de PostgreSQL, à l'endroit qu'il a choisi, et qui le configure. Toute modification directe des fichiers de configuration de PostgreSQL serait supprimée à la mise à jour suivante. Ma méthode configure proprement PostgreSQL de façon à conserver les modifications par-delà les mises à jour.

Création des utilisateurs

Pas d'utilisateur `postgres` pour Gitlab, mais `gitlab-psql`, et les chemins habituels des outils ont changé.

On se logue :

```
su gitlab-psql -s /bin/bash
```

Et on crée les utilisateurs :

```
/opt/gitlab/embedded/bin/createuser -h /var/opt/gitlab/postgresql/ -s -P barman  
/opt/gitlab/embedded/bin/createuser -h /var/opt/gitlab/postgresql/ -P --replication streaming_barman
```

Modification de la configuration

Il faut modifier le fichier `/etc/gitlab/gitlab.rb` pour que Gitlab configure PostgreSQL pour nous.

De façon un peu bête, dès qu'on fait écouter PostgreSQL sur une interface réseau, Gitlab n'essaye plus de se connecter en *socket* unix mais par le réseau... donc on va le forcer à utiliser la *socket* :

```
gitlab_rails['db_host'] = "/var/opt/gitlab/postgresql/"
```

Ensuite, c'est l'équivalent de la documentation officielle de Barman :

```
postgresql['listen_address'] = '0.0.0.0'  
postgresql['wal_level'] = "replica"  
postgresql['max_wal_senders'] = 3  
postgresql['max_replication_slots'] = 3
```

À l'exception de la façon de créer des entrées dans

```
postgresql['custom_pg_hba_entries'] = {  
  'barman': [{  
    type: 'hostssl',  
    database: 'all',  
    user: 'barman',  
    cidr: '203.0.113.42/32',  
    method: 'md5'  
  }],  
  'streaming_barman': [{  
    type: 'hostssl',  
    database: 'replication',
```

```
user: 'streaming_barman',  
cidr: '203.0.113.42/32',  
method: 'md5'  
}  
}]  
}
```

Puis il suffit de lancer la commande suivante pour que Gitlab reconfigure PostgreSQL (et tout le reste de Gitlab, mais ce n'est pas ce qui nous intéresse) :

```
gitlab-ctl reconfigure
```