Administration : gestion des droits, authentification et sauvegardes (PostgreSQL)

Anne-Cécile Caron, Anne Étien, Mikaël Monet, Sylvain Salvati

Introduction

La sécurité des données comporte 2 facettes :

1. Le contrôle d'accès aux données (qui a le droit de faire quoi). Trois objectifs :
   - Maintenir la confidentialité : les informations ne doivent pas être données à des utilisateurs non autorisés.
   - Maintenir l'intégrité : seuls les utilisateurs autorisés peuvent modifier les données.
   - Maintenir l'accessibilité : les utilisateurs autorisés doivent avoir accès aux données.
2. La tolérance aux pannes. Exemple : coupure de courant ou incendie (possiblement malintentionnels). La récupération des données se fait grâce à la *journalisation des transactions* (cf. séance précédente) et aux *sauvegardes*.

Plan

Plan

Privilèges objet

Un rôle = un utilisateur ou un groupe d'utilisateurs (plus de détails dans la section suivante).

Les privilèges objet définissent de quelle manière les différents *rôles* peuvent accéder/modifier les objets (base de données, schéma, table, vue, procédure stockée, etc.) du SGDB.

Lors de la création d'un objet, le rôle ayant exécuté l'instruction `CREATE` devient son propriétaire. Le propriétaire d'un objet a tous les droits sur celui-ci et peut donner des droits à d'autres rôles (qui n'ont par défaut aucun droits sur l'objet) en utilisant la commande suivante :

```
GRANT <liste de privilèges> ON <objet>
TO <liste de rôles>
```

Note : seul le propriétaire d'un objet – ou un superutilisateur, cf. section suivante – peut modifier (`ALTER`) ou supprimer (`DROP`) cet objet.

Exemples de privilèges

- `SELECT`, `SELECT(col,...,col)` : pour les tables et vues.
- `INSERT`, `INSERT(col,...,col)` : pour les tables et vues. Dans le deuxième cas, les colonnes non autorisées prendront les valeurs par défaut.
- `DELETE`, `UPDATE`, `UPDATE(col,...,col)` : pour les tables et vues (nécessite le privilège SELECT la plupart du temps).
- `REFERENCES(col,...,col)` : permet la création d'une clé étrangère référençant une colonne.
- `EXECUTE` : pour les procédures et fonctions stockées .
- `CREATE` :
  - pour les bases de données, permet la création de schémas
  - pour les schémas, permet la création d'objets dans ce schéma (tables, vues, procédures/fonctions stockées, indexes, etc.).
- `CONNECT` : permet de se connecter à une base de données.
- `USAGE` : pour un schéma, permet d'accéder à ses objets .
- `ALL` : mot clé qui représente tous les privilèges possibles pour le type d'objet en question.

Plan

Création de rôles

Un rôle = un utilisateur ou un groupe d'utilisateurs. Syntaxe pour créer un rôle :

```
CREATE ROLE name [ [ WITH ] option [ ... ] ]
```

Quelques exemples d'options (*privilèges système*) :

- `SUPERUSER` : détermine si le rôle est un superutilisateur. Un superutilisateur a tous les droits sur tout. Seul un rôle qui est déjà superutilisateur peut créer un autre superutilisateur.
- `CREATEDB` : détermine si le rôle a le droit de créer des bases de données.
- `CREATEROLE` : détermine si le rôle a le droit de créer d'autres rôles. Permet aussi la modification (`ALTER`) et la suppression (`DROP`) de n'importe quel rôle (sauf d'un rôle superutilisateur, à moins d'être soi-même superutilisateur).

Exemples d'options (suite)

- `LOGIN` : détermine si le rôle est autorisé à se logger (c-à-d, si on peut initier une session avec ce rôle, lors d'une connexion ; cf. section suivante). Généralement, on utilise les rôles avec `LOGIN` pour représenter un utilisateur, tandis qu'un rôle sans `LOGIN` représentera plutôt un groupe d'utilisateurs.
- `CONNECTION LIMIT` : nombre maximal de connexions simultanées pour ce rôle.
- `PASSWORD` : permet de définir un mot de passe pour ce rôle (utile seulement pour les rôles ayant `LOGIN`). Par défaut, le hash MD5 du mot de passe est stocké par le SGBD.
- `VALID UNTIL` : permet de définir une date après laquelle le mot de passe du rôle ne fonctionnera plus.
- etc.

Appartenance d'un rôle

Pour faciliter la gestions des droits, il est souvent pratique de regrouper des utilisateurs dans des groupes (un peu comme dans les systèmes UNIX). On peut ainsi donner/révoquer des droits à un ensemble d'utilisateurs en une commande. Pour ajouter/supprimer l'appartenance d'un rôle à un autre, on utilise :

```
GRANT/REVOKE group_role TO role1, ... ;
```

Par défaut, `role1` héritera alors de tous les privilèges de `group_role`, et des groupes auquels `group_role` appartient.

Un groupe peut appartenir à un autre groupe (et Postgres n'autorise pas de cycle dans le graphe d'appartenance).

Exemple

```
CREATE ROLE enseignants;
GRANT SELECT ON reservations_salles TO enseignants;
CREATE ROLE enseignants_BDR_2021;
GRANT enseignants TO enseignants_BDR_2021;
GRANT SELECT, INSERT, UPDATE, DELETE
  ON notes_BDR_2021 TO enseignants_BDR_2021;
CREATE ROLE toto LOGIN PASSWORD "totopw";
CREATE ROLE titi LOGIN PASSWORD "titipw";
GRANT enseignants_BDR_2021 TO toto,titi;
```

$\rightarrow$ `toto` et `titi` ont les droits `SELECT`, `INSERT`, `UPDATE` et `DELETE` sur la table `notes_BDR_2021` et le droit `SELECT` sur la table `reservations_salles`.

Spécificités Postgres et bonnes pratiques

- Dans la plupart des distributions, lors de l'installation de Postgres un superutilisateur `Postgres` est créé ainsi qu'une base de données initiale également appelée `postgres`. Travailler en tant que superutilisateur pour les tâches qui ne le nécessitent pas est généralement une mauvaise idée ; une bonne pratique est donc de créer un utilisateur disposant des droits `CREATEDB` et `CREATEROLE` qui ne soit pas un superutilisateur.
- Par défaut et pour des raisons de compatibilité avec d'anciennes versions, toute base de donnée créée contient un shéma `public`, et n'importe quel rôle (qui peut se connecter à la base) est autorisé à y créer des objets. Ainsi il est de bonne pratique de :
  - soit supprimer le schéma `public` et d'en créer un autre (ne pas oublier de donner le droit `USAGE` aux rôles ayant besoin d'accéder aux objets du nouveau schéma) ;
  - soit de révoquer le droit `CREATE` au rôle `PUBLIC` (mot clé qui représente tous les rôles existants) via la commande `REVOKE CREATE ON SCHEMA public FROM PUBLIC;`

Plan

Connexion et authentification

Quand un client veut se connecter, il doit spécifier (1) le rôle qu'il souhaite prendre et (2) la base de données à laquelle il veut se connecter. Dans ce qui suit, nous appellerons *utilisateur* un rôle avec option `LOGIN`.

Avec le client `psql` par exemple on utilise la commande
`psql -U utilisateur -d base`

Le fichier (serveur) `pg_hba.conf` sert à contrôler les connexions : *depuis quelles machines quels utilisateurs peuvent se connecter à quelles bases, avec quelle méthode d'authentification*. Il se compose d'une suite de lignes. Dans ce cours nous ne considérerons que les lignes du type
`local database user auth-method`
et
`host database user address auth-method`

Lors d'une tentative de connexion, la première ligne qui correspond au type de connexion est utilisée pour effectuer l'authentification, et si l'authentification échoue les lignes d'après ne sont pas considérées. Si aucune ligne ne correspond la connexion est refusée également.

Connexion et authentification : options (1/2)

```
local  database  user  auth-method
```

- `local` : connexions sockets domaine Unix (donc depuis la machine sur laquelle le serveur tourne)
- `database` : liste de noms de bases de données, séparés par des virgules. Valeurs spéciales `all` (n'importe quelle base), `sameuser` (le nom de la base doit être identique au nom d'utilisateur) et `samerole` (l'utilisateur doit faire partie d'un rôle portant le même nom que la base).
- `database` : liste de rôles (correspond exactement au rôle demandé), ou de rôles précédés par "+" (accepte les rôles qui sont membres de ce rôle), le tout séparé par des virgules. Valeur spéciale `all`.
- `auth-method` :
  - `trust` : autorise la connexion sans condition
  - `password` : le client doit indiquer un mot de passe, qui est envoyé en clair sur le réseau. Si aucun mot de passe n'a été défini pour l'utilisateur, la connexion échoue.
  - `md5` : pareil, mais seulement le hash est envoyé sur le réseau
  - `reject` : rejette la connexion
  - etc.

Connexion et authentification : options (2/2)

```
host  database  user  address  auth-method
```

Même fonctionnement, mais on doit indiquer une adresse ou une plage d'adresses IP, ou un nom de domaine (suppose d'avoir bien configuré son DNS...). Voir la doc pour les détails syntaxiques.

Note 1 : pour qu'un rôle puisse se connecter à une base, il faut qu'il passe le test `pg_hba.conf`, et aussi qu'il ait le privilège `CONNECT` pour la base !
Note 2 : le fichier `pg_hba.conf` est lu au lancement du serveur. Quand on le modifie, on peut par exemple éteindre et redémarrer le serveur pour qu'il soit pris en compte.

Plan

Sauvegardes

Les sauvegardes sont là pour limiter les dégâts en cas de pannes matériel, incendies, vols de matériel, etc..

Évidemment, on évitera de stocker les sauvegardes sur le (les) même support que là où est stockée la base : disques différents, bâtiment différent, sauvegardes non connectées au réseau pour plus de sécurité en cas d'intrusion, etc.

Trois approches pour sauvegarder des bases de données :

- sauvegarde SQL
- sauvegarde au niveau du système de fichiers
- archivage continu

Sauvegarde SQL (SQL dump) (1/2)

La commande `pg_dump base_de_donnees > fichier_sauvegarde` produit un fichier texte contenant des commandes SQL telles que, si on les redonne au SGBD, cela va recréer la base dans le même état.

`pg_dump` est un client PostgreSQL comme les autres ; il faudra donc lancer la commande avec un utilisateur qui a des droits suffisants pour lire la base entièrement (par exemple `pg_dump -U postgres base_de_donnees`). Sinon, on peut préciser des parties de la base à sauvegarder, avec les options `-n schema` ou `-t table`.

Sauvegarde SQL (SQL dump) (2/2)

Restauration avec

```
psql dbname < fichier_sauvegarde
```

Attention ! Les rôles propriétaires ou ayant des privilèges sur les objets de la base doivent exister.

La commande `pg_dumpall` permet de sauvegarder le cluster de bases en entier (avec les rôles, entre autres).

Avantages de l'approche à base de dumps :

- Restauration possible dans des versions de PostgreSQL plus récentes (ce qui peut ne pas être le cas pour des sauvegardes au niveau du système de fichiers).
- `pg_dump` peut être exécuté alors que le serveur tourne, et il renverra une version consistante de la base (cf. les techniques du cours précédents).
- La sauvegarde prend souvent moins de place que les fichiers utilisés pour stocker la base. (pourquoi ?)

Sauvegarde des fichiers

On peut sauvegarder les fichiers du SGBD (typiquement, dans `\usr/local/pgsql/data`) comme n'importe quels autres fichiers (`cp`, `tar`, `rsync`, etc.).

Inconvénients :

- Le serveur doit être éteint pour assurer la cohérence
- Impossible de ne sauvegarder qu'une partie du cluster : on doit tout sauvegarder
- Prend plus de place d'un dump

Avantage : selon les cas, cette méthode pourrait être plus rapide

Archivage continu (1/2)

PostgreSQL maintient des fichiers journaux qui enregistrent chaque modification faite sur les fichier des bases. L'idée est de sauvegarder périodiquement la base, et de sauvegarder au fur et à mesure ces fichiers de log.

La restauration peut alors se faire à partir de la dernière sauvegarde en rejouant les entrées des fichiers de log.

Archivage continu (2/2)

Avantages :

- On peut restaurer la base à n'importe quel moment depuis la dernière sauvegarde complète.
- Si la base de très volumineuse, il peut être plus intéressant de sauvegarder fréquemment les log et rarement la base en entier, plutôt que de faire des sauvegardes complètes à chaque fois.
- Les fichiers journaux peuvent être fournis en continu à une autre machine qui est chargé avec la même base sauvegardée, ce qui permet de faire de la *reprise à chaud*.

Inconvénients :

- Plus complexe à mettre en place.
- Comme pour les sauvegardes par fichiers, on est forcé de sauvegarder tout le cluster.