−Table des matières



# Créer une base avec PostgreSQL

*Cet article, écrit par Guillaume Lelarge, a été publié dans le hors-série 44 du magazine GNU/Linux Magazine France, hors-série dédié à PostgreSQL. Il est disponible maintenant sous licence Creative Commons.*

---


Nous allons construire une petite base, et voir quelques rudiments : création d'une base, création de quelques tables, insertion de données, sauvegarde. Le minimum pour savoir comment débuter.

## Qui suis-je ?

Nous allons tout faire en tant que super-utilisateur de la base de données. Ce sera postgres pour ceux qui ont installé PostgreSQL avec les paquets de leur distribution Linux, ou l'utilisateur qui a lancé le serveur PostgreSQL pour ceux qui l'ont compilé.

Nous ne nous occuperons donc pas de la gestion des droits.

## Créer une base de données

Une fois que le serveur est installé et qu'il est en cours d'exécution, la première chose à faire est de créer une base. Cette base va nous servir à y stocker des données. Il existe pour cela un outil utilisable sur la ligne de commande shell : createdb. Il dispose de plusieurs options, mais nous allons faire simple pour l'instant :

```
postgres@debian1:~$ createdb base1
```

Un seul argument, le nom de la base de données. Lorsque nous voudrons nous connecter à cette base, nous n'aurons que ce nom à fournir.

## Créer une table

Pour créer une table, nous allons lancer la requête SQL adéquate via la console interactive de PostgreSQL. Cet outil est nommé psql. C'est certainement l'outil le plus utilisé pour interagir avec ce SGBD. Tout simplement parce que, bien qu'il puisse sembler simpliste, il est en fait extrêmement puissant tout en restant simple à utiliser.

Nous allons donc entrer dans la console interactive :

```
postgres@debian1:~$ psql base1
psql (8.4.0)
Saisissez « help » pour l'aide.

base1=#
```

« base1=# » est l'invite (aussi appelé en anglais prompt). Il indique le nom de la base où on est connecté. Il ne nous reste plus qu'à saisir une requête ou une méta-commande. Pour créer une table, il nous faut utiliser l'instruction « CREATE TABLE ». Il est possible d'avoir de l'aide sur les instructions SQL avec la méta-commande \h. Sans plus, elle renvoie la liste des instructions SQL connues. Avec l'instruction SQL, elle renvoie la syntaxe complète de l'instruction :

```
base1=# \h CREATE TABLE
Commande :    CREATE TABLE
Description : définir une nouvelle table
Syntaxe :
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE nom_table ( [
  { nom_colonne type_données [ DEFAULT expr_par_défaut ]
    [ contrainte_colonne [ ... ] ]
    | contrainte_table
    | LIKE table_parent [ { INCLUDING | EXCLUDING }
      { DEFAULTS | CONSTRAINTS | INDEXES } ] ... }
    [, ... ]
] )
[ INHERITS ( table_parent [, ... ] ) ]
[ WITH ( paramètre_stockage [= valeur] [, ... ] )  | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace ]
```

```
(etc.)
```

Pour créer une table, il faut donc, au minimum, lui donner un nom et des colonnes. Les colonnes sont indiquées un peu comme les arguments d'une fonction : nom de la colonne puis type de données de cette colonne. Créons une table contenant une liste d'articles. Nous aurons une colonne titre et une colonne auteur.

```
base1=# CREATE TABLE articles (titre text, auteur text);
CREATE TABLE
```

text est le type de données pour les champs contenant des chaînes de caractères. Le point-virgule à la fin permet d'indiquer à psql que la requête est terminée et que nous voulons qu'il l'exécute. Après une exécution réussie, psql renvoie un message de confirmation : « CREATE TABLE » dans notre cas.

Pour ajouter des données, nous utilisons l'ordre INSERT.

```
base1=# \h insert
Commande :    INSERT
Description : créer de nouvelles lignes dans une table
Syntaxe :
INSERT INTO table [ ( colonne [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | requête }
    [ RETURNING * | expression_sortie [ [ AS ] nom_sortie ] [, ...] ]

base1=# INSERT INTO articles (titre, auteur) VALUES ('Ubuntu: installation dual boot sur clef USB', 'Yves Bailly');
INSERT 0 1
base1=# INSERT INTO articles (titre, auteur) VALUES ('Stockage en ligne : NBD dans un tunnel SSH', 'Fabien Germain');
INSERT 0 1
base1=# INSERT INTO articles (titre, auteur) VALUES ('Parce qu''y''en a marre', 'Jean-Pierre Troll');
INSERT 0 1
base1=# INSERT INTO articles (titre, auteur) VALUES ('Créez votre live CD Debian 5.0 Lenny', 'Denis Boor');
INSERT 0 1
```

Aie, je me suis trompé dans le nom de notre rédacteur-en-chef préféré ! Pour corriger cela, nous pouvons utiliser l'instruction SQL UPDATE. Nous allons mettre à jour la colonne auteur, mais comment faire pour sélectionner la bonne ligne ? Si je sélectionne par rapport au titre, rien ne me dit que je vais pouvoir sélectionner une ligne unique. Dans notre cas, il n'y aura pas ce problème. Mais j'aurais pu ajouter d'autres articles de Jean-Pierre Troll… tous ses articles ont pour titre « Parce qu'y'en a marre », ce qui rend leur distinction difficile. De ce fait, chaque table doit avoir une façon unique d'identifier chaque ligne. Le moyen habituel est d'associer à la table une clé primaire. Cette dernière peut se poser sur une colonne ou sur un groupe de colonne. Dans notre contexte, le meilleur moyen d'identifier un article est de connaître le numéro du magazine et le titre de l'article. Nous allons donc ajouter dans un premier temps une nouvelle colonne à la table articles:

```
base1=# ALTER TABLE articles ADD COLUMN numeromagazine integer;
ALTER TABLE
```

Nous allons ajouter la clé primaire:

```
base1=# ALTER TABLE articles ADD CONSTRAINT articles_pkey PRIMARY KEY (numeromagazine, titre);
NOTICE:  ALTER TABLE / ADD PRIMARY KEY créera un index implicite « articles_pkey » pour la table « articles »
ALTER TABLE
```

Notez que, dès maintenant, je peux ajouter deux fois un article de même nom, à condition que le numéro du magazine soit différent:

```
base1=# INSERT INTO articles (numeromagazine, titre, auteur) VALUES (116, 'Parce qu''y''en a marre', 'Jean-Pierre Troll');
INSERT 0 1
base1=# INSERT INTO articles (numeromagazine, titre, auteur) VALUES (117, 'Parce qu''y''en a marre', 'Jean-Pierre Troll');
INSERT 0 1
base1=# INSERT INTO articles (numeromagazine, titre, auteur) VALUES (117, 'Parce qu''y''en a marre', 'Jean-Pierre Troll');
ERREUR:  la valeur d'une clé dupliquée rompt la contrainte unique « articles_pkey »
base1=# INSERT INTO articles (numeromagazine, titre, auteur) VALUES (118, 'Parce qu''y''en a marre', 'Jean-Pierre Troll');
INSERT 0 1
```

Nous allons supprimer toutes les lignes, puis ajouter de nouveau les anciennes lignes en indiquant en plus le numéro. La suppression se fait avec l'instruction DELETE.

```
base1=# DELETE FROM articles;
DELETE 4
base1=# INSERT INTO articles (numeromagazine, titre, auteur) VALUES (118, 'Ubuntu: installation dual boot sur clef USB', 'Yves Bailly');
INSERT 0 1
base1=# INSERT INTO articles (numeromagazine, titre, auteur) VALUES (118, 'Stockage en ligne : NBD dans un tunnel SSH', 'Fabien Germain');
INSERT 0 1
base1=# INSERT INTO articles (numeromagazine, titre, auteur) VALUES (115, 'Parce qu''y''en a marre', 'Jean-Pierre Troll');
INSERT 0 1
base1=# INSERT INTO articles (numeromagazine, titre, auteur) VALUES (115, 'Créez votre live CD Debian 5.0 Lenny', 'Denis Boor');
```

Nous allons enfin pouvoir modifier cette ligne:

```
base1=# UPDATE articles SET auteur='Denis Bodor' WHERE numeromagazine=115 AND auteur='Denis Boor';
UPDATE 1
```

psql nous répond que nous avons bien modifié une seule ligne. Tout va bien.

SELECT est l'instruction SQL pour récupérer des informations. La clause WHERE aide à filtrer les lignes à récupérer. Par exemple :

```
base1=# SELECT auteur, titre FROM articles WHERE numeromagazine=115;
      auteur       |               titre
-------------------+-------------------------------------
 Jean-Pierre Troll | Parce qu'y'en a marre
 Denis Bodor       | Créez votre live CD Debian 5.0 Lenny
(2 lignes)
```

Cette requête récupère les lignes dont la colonne numeromagazine vaut 115 et affiche les colonnes auteur et titre.

Ajoutons une nouvelle table, celle des magazines. Nous lui donnerons deux colonnes, le numéro et la date de sortie.

```
base1=# CREATE TABLE magazines (numeromagazine integer primary key, datesortie date);
NOTICE:  CREATE TABLE / PRIMARY KEY créera un index implicite « magazines_pkey » pour la table « magazines »
CREATE TABLE
```

Remarquez que la définition de clé primaire est intégrée à la définition de la table. Maintenant, nous allons pouvoir ajouter des éléments dans cette table.

```
base1=# INSERT INTO magazines (numeromagazine, datesortie) VALUES (115, '2009-04-01');
INSERT 0 1
base1=# INSERT INTO magazines (numeromagazine, datesortie) VALUES (116, '2009-05-01');
INSERT 0 1
base1=# INSERT INTO magazines (numeromagazine, datesortie) VALUES (117, '2009-06-01');
INSERT 0 1
base1=# INSERT INTO magazines (numeromagazine, datesortie) VALUES (118, '2009-07-01');
INSERT 0 1
```

Un des gros intérêts des SGBD, c'est de pouvoir mettre la vérification des données de leur côté. Par exemple, nous pouvons très bien faire en sorte qu'il ne soit pas possible d'insérer un article pour lequel le numéro du magazine n'existe pas dans la table magazines. Le terme technique est clé étrangère. Voici sa déclaration

```
base1=# ALTER TABLE articles ADD CONSTRAINT mag FOREIGN KEY (numeromagazine) REFERENCES magazines(numeromagazine);
ALTER TABLE
```

Testons de suite :

```
base1=# INSERT INTO articles (numeromagazine, auteur, titre) VALUES (115, 'Lionel Tricon', 'Espionnez vos applications avec strace et ltrace');
INSERT 0 1
```

Le numéro 115 existe bien, donc l'insertion est acceptée.

```
base1=# insert into articles (numeromagazine, auteur, titre) values (114, 'Olivier Delhomme', 'ZFS sous GNU/Linux');
ERREUR:  une instruction insert ou update sur la table « articles » viole la contrainte de clé
étrangère « mag »
DÉTAIL : La clé (numeromagazine)=(114) n'est pas présente dans la table « magazines ».
```

Le numéro 114 n'existe pas, donc un échec a lieu lors de cette insertion.

```
base1=# INSERT INTO magazines (numeromagazine, datesortie) VALUES (114, '2009-03-01');
INSERT 0 1
base1=# INSERT INTO articles (numeromagazine, auteur, titre) VALUES (114, 'Olivier Delhomme', 'ZFS sous GNU/Linux');
INSERT 0 1
```

Après insertion du magazine 114 dans la table magazines, l'insertion de l'article est possible.

Avec ces tables, nous allons pouvoir faire des requêtes un peu plus complexes. Par exemple, trouver la date de sortie de chaque article. Le lien est la colonne numeromagazine, nous allons récupérer une colonne d'une table et une colonne d'une autre table et nous allons les joindre comme ceci:

```
base1=# SELECT datesortie, titre FROM articles, magazines
base1-# WHERE articles.numeromagazine = magazines.numeromagazine
base1-# ORDER BY datesortie;
 datesortie |                    titre
------------+---------------------------------------------
 2009-03-01 | ZFS sous GNU/Linux
 2009-04-01 | Créez votre live CD Debian 5.0 Lenny
 2009-04-01 | Parce qu'y'en a marre
 2009-04-01 | Espionnez vos applications avec strace et ltrace
 2009-05-01 | Parce qu'y'en a marre
 2009-06-01 | Parce qu'y'en a marre
 2009-07-01 | Parce qu'y'en a marre
 2009-07-01 | Ubuntu: installation dual boot sur clef USB
 2009-07-01 | Stockage en ligne : NBD dans un tunnel SSH
(9 lignes)
```

Pour éviter d'avoir à saisir des grosses requêtes, nous pouvons en enregistrer certaines dans des vues:

```
base1=# CREATE VIEW date_et_titre_article AS
base1-# SELECT datesortie, titre FROM articles, magazines
base1-# WHERE articles.numeromagazine = magazines.numeromagazine
base1-# ORDER BY datesortie;
CREATE VIEW
base1=# SELECT * FROM date_et_titre_article;
 datesortie |                    titre
------------+---------------------------------------------
 2009-03-01 | ZFS sous GNU/Linux
 2009-04-01 | Créez votre live CD Debian 5.0 Lenny
 2009-04-01 | Parce qu'y'en a marre
 2009-04-01 | Espionnez vos applications avec strace et ltrace
 2009-05-01 | Parce qu'y'en a marre
 2009-06-01 | Parce qu'y'en a marre
 2009-07-01 | Parce qu'y'en a marre
 2009-07-01 | Ubuntu: installation dual boot sur clef USB
 2009-07-01 | Stockage en ligne : NBD dans un tunnel SSH
(9 lignes)
```

Cette vue se comporte maintenant comme une table. Par exemple, nous pouvons filtrer sur la date:

```
base1=# SELECT * FROM date_et_titre_article
base1-# WHERE datesortie BETWEEN '2009-05-01' AND '2009-06-01';
 datesortie |                    titre
------------+---------------------------------------------
 2009-05-01 | Parce qu'y'en a marre
 2009-06-01 | Parce qu'y'en a marre
(2 lignes)
```

Bon, maintenant, nous savons insérer, supprimer et modifier des données grâce aux instructions INSERT, DELETE et UPDATE. Nous savons les récupérer grâce à l'instruction SELECT. Tout ceci est très bien. Mais encore faut-il pouvoir les sauvegarder car on n'est jamais à l'abri d'un disque défectueux. L'outil pour PostgreSQL s'appelle pg_dump. Son utilisation basique est des plus simples:

```
postgres@debian1:~$ pg_dump base1 > base1.dump
```

Et voilà. Autant dire qu'il est difficile de trouver plus simple. Le résultat est un fichier texte SQL capable de générer la base base1, avec la même structure et les mêmes données. Pour la restaurer, psql est l'outil à connaître. Créons une nouvelle base et restaurons notre sauvegarde sur cette nouvelle base:

```
postgres@debian1:~$ createdb base2
postgres@debian1:~$ psql -f base1.dump base2
postgres@debian1:~$ psql base1
psql (8.4.0)
Saisissez « help » pour l'aide.

base1=# SELECT count(*) FROM articles;
  count
 -------
  9
(1 ligne)
```

## Conclusion

Ainsi se conclut notre entrée dans le monde de PostgreSQL. Évidemment, il y a plein d'autres choses à découvrir mais ceci devrait vous permettre de commencer plus simplement.