

Module F

Tâches courantes



24.09

Table des matières

Sur ce document	1
Chers lectrices & lecteurs,	1
À propos de DALIBO	1
Remerciements	2
Forme de ce manuel	2
Licence Creative Commons CC-BY-NC-SA	2
Marques déposées	3
Versions de PostgreSQL couvertes	3
1/ Tâches courantes	5
1.1 Introduction	6
1.2 Bases	7
1.2.1 Liste des bases	7
1.2.2 Modèle (template)	10
1.2.3 Création d'une base	11
1.2.4 Suppression d'une base	14
1.2.5 Modification / configuration	15
1.3 Rôles	18
1.3.1 Utilisateurs et groupes	18
1.3.2 Liste des rôles	18
1.3.3 Création d'un rôle	21
1.3.4 Suppression d'un rôle	24
1.3.5 Modification d'un rôle	25
1.3.6 Mot de passe	28
1.3.7 Sécurité des mots de passe	30
1.4 Droits sur les objets	34
1.4.1 Droits sur les objets	34
1.4.2 Afficher les droits	39
1.4.3 Droits sur les métadonnées	40
1.4.4 Droits plus globaux 1/2	42
1.4.5 Droits plus globaux 2/2	43
1.4.6 Héritage des droits	44
1.4.7 Changement de rôle	45
1.5 Droits de connexion	48
1.5.1 Informations de connexion	48
1.5.2 Configuration de l'authentification : pg_hba.conf	49
1.5.3 Exemple de pg_hba.conf	51
1.5.4 pg_hba.conf : colonne type	52
1.5.5 pg_hba.conf : colonne database	52
1.5.6 pg_hba.conf : colonne user	53
1.5.7 pg_hba.conf : colonne adresse IP	54
1.5.8 pg_hba.conf : colonne méthode	55

1.5.9	pg_hba.conf : colonne options	55
1.5.10	pg_hba.conf : méthodes internes	56
1.5.11	pg_hba.conf : méthodes externes	56
1.5.12	Un (mauvais) exemple de pg_hba.conf	57
1.5.13	Mapping : pg_ident.conf	58
1.6	Synchronisation des rôles	60
1.7	Tâches de maintenance	61
1.7.1	Maintenance : VACUUM	62
1.7.2	Maintenance : VACUUM FULL	64
1.7.3	VACUUM vs VACUUM FULL	65
1.7.4	Maintenance : ANALYZE	66
1.7.5	Maintenance : REINDEX	67
1.7.6	Maintenance : CLUSTER	69
1.7.7	Maintenance : automatisation	69
1.7.8	Maintenance : autovacuum	70
1.7.9	Maintenance : Script de REINDEX	70
1.8	Sécurité	75
1.8.1	Droits par défaut	75
1.8.2	Droits par défaut (suite)	77
1.8.3	Droits par défaut (suite)	78
1.8.4	Restreindre les droits	79
1.8.5	Arrêter une requête ou une session	84
1.8.6	Chiffrements	86
1.8.7	En cas de crash	86
1.8.8	Corruption de données	87
1.9	Conclusion	89
1.9.1	Pour aller plus loin	89
1.9.2	Questions	89
1.10	Quiz	90
1.11	Travaux pratiques	91
1.11.1	Traces maximales	91
1.11.2	Méthode d'authentification	91
1.11.3	Création des bases	92
1.11.4	Mots de passe	92
1.11.5	Rôles et permissions	93
1.11.6	Autorisation d'accès distant	95
1.11.7	VACUUM, VACUUM FULL, DELETE, TRUNCATE	95
1.11.8	Statistiques	97
1.11.9	Réindexation	98
1.11.10	Traces	98
1.12	Travaux pratiques (solutions)	100
1.12.1	Traces maximales	100
1.12.2	Méthode d'authentification	101
1.12.3	Création des bases	102
1.12.4	Mots de passe	103

1.12.5 Rôles et permissions	108
1.12.6 Autorisation d'accès distant	114
1.12.7 VACUUM, VACUUM FULL, DELETE, TRUNCATE	115
1.12.8 Réactivation de l'autovacuum	124
1.12.9 Statistiques des données	126
1.12.10 Réindexation	127
1.12.11 Traces	129
Les formations Dalibo	131
Cursus des formations	131
Les livres blancs	132
Téléchargement gratuit	132

Sur ce document

Formation	Module F
Titre	Tâches courantes
Révision	24.09
PDF	https://dali.bo/f_pdf
EPUB	https://dali.bo/f_epub
HTML	https://dali.bo/f_html
Slides	https://dali.bo/f_slides
TP	https://dali.bo/f_tp
TP (solutions)	https://dali.bo/f_solutions

Vous trouverez en ligne les différentes versions complètes de ce document.

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.com¹ !

À propos de DALIBO

DALIBO est le spécialiste français de PostgreSQL. Nous proposons du support, de la formation et du conseil depuis 2005.

Retrouvez toutes nos formations sur <https://dalibo.com/formations>

¹<mailto:formation@dalibo.com>

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, Pierrick Chovelon, 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, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachaires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

Forme de ce manuel

Les versions PDF, EPUB ou HTML de ce document sont structurées autour des slides de nos formations. Le texte suivant chaque slide contient le cours et de nombreux détails qui ne peuvent être données à l'oral.

Licence Creative Commons CC-BY-NC-SA

Cette formation est sous licence **CC-BY-NC-SA**². Vous êtes libre de la redistribuer et/ou modifier aux conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre). À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web. Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre. Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible sur <http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode>

²<http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode>

Cela inclut les diapositives, les manuels eux-mêmes et les travaux pratiques. Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées³ par PostgreSQL Community Association of Canada.

Versions de PostgreSQL couvertes

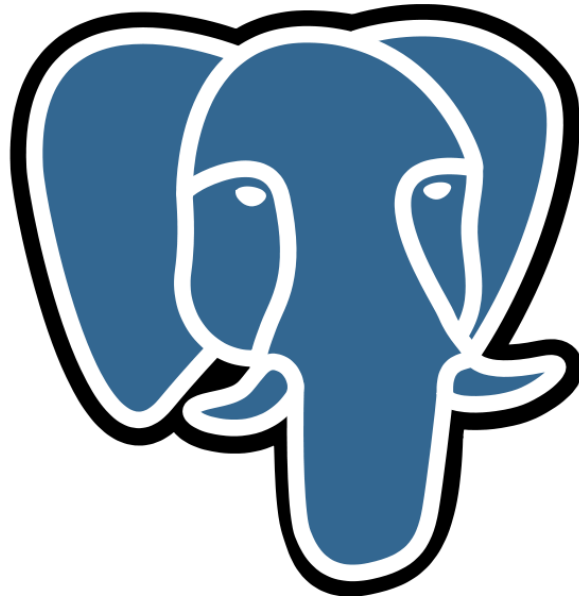
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

Sur les versions précédentes susceptibles d'être encore rencontrées en production, seuls quelques points très importants sont évoqués, en plus éventuellement de quelques éléments historiques.

Sauf précision contraire, le système d'exploitation utilisé est Linux.

³<https://www.postgresql.org/about/policies/trademarks/>

1/ Tâches courantes



1.1 INTRODUCTION



- Gestion des bases
- Gestion des rôles
- Gestion des droits
- Tâches du DBA
- Sécurité

1.2 BASES



- Liste des bases
- Modèle (Template)
- Création
- Suppression
- Modification / configuration

Pour gérer des bases, il faut savoir les créer, les configurer et les supprimer. Il faut surtout comprendre qui a le droit de faire quoi, et comment. Ce chapitre détaille chacune des opérations possibles concernant les bases sur une instance.

1.2.1 Liste des bases



- Catalogue système : `pg_database`
- Commande `psql` :
 - `\l`
 - `\l+`

La liste des bases de données est disponible grâce à un catalogue système appelé `pg_database`. Il suffit d'un `SELECT` pour récupérer les méta-données sur chaque base :

```
postgres=# \x
Expanded display is on.
postgres=# SELECT * FROM pg_database \gx
```

```
-[ RECORD 1 ]--+-+-----
oid           | 5
datname       | postgres
datdba        | 10
encoding      | 6
datlocprovider | c
datistemplate | f
dataallowconn | t
datconlimit   | -1
datfrozenxid  | 8885214
datminmxid    | 1
dattablespace | 1663
datcollate    | fr_FR.UTF-8
datctype      | fr_FR.UTF-8
```

DALIBO Formations

```

daticulocale |
daticurules  |
datcollversion | 2.36
datacl       |
-[ RECORD 2 ]-----
oid          | 18770
datname      | cave
datdba       | 18769
encoding     | 6
datlocprovider | c
datistemplate | f
dataallowconn | t
datconlimit  | -1
datfrozenxid | 722
datminmxid   | 1
dattablespace | 1663
datcollate   | fr_FR.UTF-8
datctype     | fr_FR.UTF-8
daticulocale |
daticurules  |
datcollversion | 2.36
datacl       |
-[ RECORD 3 ]-----
oid          | 1
datname      | template1
datdba       | 10
encoding     | 6
datlocprovider | c
datistemplate | t
dataallowconn | t
datconlimit  | -1
datfrozenxid | 722
datminmxid   | 1
dattablespace | 1663
datcollate   | fr_FR.UTF-8
datctype     | fr_FR.UTF-8
daticulocale |
daticurules  |
datcollversion | 2.36
datacl       | {=c/postgres,postgres=CTc/postgres}
-[ RECORD 4 ]-----
oid          | 4
datname      | template0
datdba       | 10
encoding     | 6
datlocprovider | c
datistemplate | t
dataallowconn | f
datconlimit  | -1
datfrozenxid | 722
datminmxid   | 1
dattablespace | 1663
datcollate   | fr_FR.UTF-8
datctype     | fr_FR.UTF-8
daticulocale |
daticurules  |

```

```
datcollversion |
datacl         | {=c/postgres,postgres=CTc/postgres}
...
```

Voici la signification des principales colonnes :

- `oid` : identifiant système de la base ;
- `datname` : nom de la base ;
- `datdba` : l'identifiant de l'utilisateur propriétaire de cette base (voir l'OID correspondant à cet identifiant dans le catalogue système `pg_roles`) ;
- `encoding` : avec la fonction `pg_encoding_to_char()`, on peut voir que les valeurs 6 correspondent à UTF8, ce qui est généralement recommandé ;
- `datlocprovider` : `c` indique que les collations sont fournies par la bibliothèque `libc` du système, un `i` indiquerait l'utilisation de la bibliothèque ICU (indépendante du système), et dans ce dernier cas les champs `daticulocale` et `daticurules` sont remplies ;
- `datistemplate` : à `true` si cette base est utilisable comme modèle ;
- `dataallowconn` : à `true` s'il est autorisé de se connecter à cette base ;
- `datconnlimit` : limite du nombre de connexions simultanées pour les utilisateurs standards sur cette base (`0` interdit toute connexion, et `-1` ne pose pas de limite, jusque `max_connections` du moins) ;
- `datfrozenxid` : plus ancien identifiant de transaction géré par cette base (cela a une importance dans le recyclage des numéros de transaction) ;
- `dattablespace` : identifiant du tablespace par défaut de cette base (1663 indique généralement le tablespace `pg_default`, de fait le répertoire PGDATA ; les emplacements des tablespaces peuvent se trouver dans la table `pg_tablespace` ou par la fonction `pg_tablespace_location()`) ;
- `datacl` : droits pour cette base (un champ vide indique qu'il n'y a pas de droits spécifiques).

Pour avoir une vue plus simple, il est préférable d'utiliser la métacommande `\l` dans `psql` (vue raccourcie pour la mise en page) :

```
postgres=# \l
                                List of databases
  Name      | Owner   | Enc. | Locale P. | Collate | ... | Access privileges
-----+-----+-----+-----+-----+-----+-----
cave        | caviste | UTF8 | libc       | fr_FR.UTF-8 | | 
pgbench     | pgbench | UTF8 | libc       | fr_FR.UTF-8 | | 
postgres    | postgres | UTF8 | libc       | fr_FR.UTF-8 | | 
template0   | postgres | UTF8 | libc       | fr_FR.UTF-8 | | =c/postgres          +
             |         |     |           |           | | postgres=CTc/postgres
template1   | postgres | UTF8 | libc       | fr_FR.UTF-8 | | =c/postgres          +
             |         |     |           |           | | postgres=CTc/postgres
tpc         | tpc_owner | UTF8 | libc       | fr_FR.UTF-8 | | 
...
```

Avec `\l+`, il est possible d'avoir plus d'informations (notamment la taille de la base ou le commentaire).

Noter que si l'on veut savoir où psql va chercher ces informations, il est possible de lui demander d'afficher la requête qu'il envoie au serveur :

```
$ psql --echo-hidden -c '\l+'

***** QUERY *****
SELECT
  d.datname as "Name",
  pg_catalog.pg_get_userbyid(d.datdba) as "Owner",
  pg_catalog.pg_encoding_to_char(d.encoding) as "Encoding",
  CASE d.datlocprovider WHEN 'c' THEN 'libc' WHEN 'i' THEN 'icu' END AS "Locale
↵ Provider",
  d.datcollate as "Collate",
  d.datctype as "Ctype",
  d.daticulocale as "ICU Locale",
  d.daticurules as "ICU Rules",
  pg_catalog.array_to_string(d.datacl, E'\n') AS "Access privileges",
  CASE WHEN pg_catalog.has_database_privilege(d.datname, 'CONNECT')
    THEN pg_catalog.pg_size_pretty(pg_catalog.pg_database_size(d.datname))
    ELSE 'No Access'
  END as "Size",
  t.spcname as "Tablespace",
  pg_catalog.shobj_description(d.oid, 'pg_database') as "Description"
FROM pg_catalog.pg_database d
  JOIN pg_catalog.pg_tablespace t on d.dattablespace = t.oid
ORDER BY 1;
*****
...
```

(suivent les résultats.)

La requête affichée montre bien que `psql` accède au catalogue `pg_database`, ainsi qu'à des fonctions système permettant d'éviter d'avoir à faire soi-même les jointures.

1.2.2 Modèle (template)



- Toute création de base se fait à partir d'un modèle
 - `template1`
- Permet de personnaliser sa création de base
- Mais il est aussi possible d'utiliser une autre base

Toute création de base se fait à partir d'un modèle. Par défaut, PostgreSQL utilise le modèle `template1`.

Tout objet ajouté dans le modèle est copié dans la nouvelle base. Cela concerne le schéma (la structure) comme les données. Il est donc intéressant d'ajouter des objets directement dans `template1`

pour que ces derniers soient copiés dans les prochaines bases qui seront créées. Pour éviter malgré tout que cette base soit trop modifiée, il est possible de créer des bases qui seront ensuite utilisées comme modèle.

1.2.3 Création d'une base



- **SQL** : `CREATE DATABASE`
 - droit nécessaire: `SUPERUSER` ou `CREATEDB`
 - prérequis : base inexistante
- **Outil système** : `createdb`

L'ordre `CREATE DATABASE` est le seul moyen avec PostgreSQL de créer une base de données. Il suffit d'y ajouter le nom de la base à créer pour que la création se fasse. Il est néanmoins possible d'y ajouter un certain nombre d'options :

- `OWNER` , pour préciser le propriétaire de la base de données (si cette option n'est pas utilisée, le propriétaire est celui qui exécute la commande) ;
- `TEMPLATE` , pour indiquer le modèle à copier (par défaut `template1`) ;
- `ENCODING` , pour forcer un autre encodage que celui du serveur (à noter qu'il faudra utiliser le modèle `template0` dans ce cas) ;
- `LC_COLLATE` et `LC_CTYPE` , pour préciser respectivement l'ordre de tri des données textes et le jeu de caractères (par défaut, il s'agit de la locale utilisée lors de l'initialisation de l'instance) ;
- `STRATEGY` (depuis la version 15), pour indiquer la stratégie employée pour créer la base de donnée. Deux choix sont disponibles :
 - `FILE_COPY` : C'est la méthode historique, seule possible jusqu'en version 14 incluse. Le contenu des répertoires d'une base de données est intégralement copié pour initialiser la nouvelle base, avec juste une trace dans les journaux de transaction. Elle implique deux *checkpoints* parfois gênants, mais peut être intéressante pour copier de grosses bases en générant moins de journaux ;
 - `WAL_LOG` : C'est la méthode par défaut à partir de la version 15. L'opération est entièrement journalisée, et la liste des objets à copier est générée via le catalogue de PostgreSQL. Cette méthode évite les checkpoints et sécurise la copie en garantissant que toutes les opérations sont tracées, tout en évitant la copie accidentelle de fichier orphelins de la base modèle vers la base cible. Cette opération peut écrire beaucoup de journaux dans le cas où la base modèle est grosse, mais elle est idéale pour les créations de nouvelles bases presque vides ;
- `TABLESPACE` , pour stocker la base dans un autre tablespace que le répertoire des données ;
- `ALLOW_CONNECTIONS` , pour autoriser ou non les connexions à la base ;

- `CONNECTION LIMIT`, pour limiter le nombre de connexions d'utilisateurs standards simultanées sur cette base (illimité par défaut, tout en respectant le paramètre `max_connections`);
- `IS_TEMPLATE`, pour configurer ou non le mode template.

La copie se fait par clonage de la base de données modèle sélectionnée. Tous les objets et toutes les données faisant partie du modèle seront copiés sans exception. Par exemple, avant la 9.0, on ajoutait le langage PL/pgSQL dans la base de données `template1` pour que toutes les bases créées à partir de `template1` disposent directement de ce langage. Ce n'est plus nécessaire à partir de la 9.0 car le langage PL/pgSQL est activé dès la création de l'instance. Mais il est possible d'envisager d'autres usages de ce comportement (par exemple installer une extension ou une surcouche comme PostGIS dans chaque base).

À noter qu'il peut être nécessaire de sélectionner le modèle `template0` en cas de sélection d'un autre encodage que celui par défaut (comme la connexion est interdite sur `template0`, il y a peu de chances que des données textes avec un certain encodage aient été enregistrées dans cette base).

Voici l'exemple le plus simple de création d'une base :

```
CREATE DATABASE b1 ;
```

Cet ordre crée la base de données **b1**. Elle aura toutes les options par défaut. Autre exemple :

```
CREATE DATABASE b2 OWNER u1;
```

Cette commande SQL crée la base **b2** et s'assure que le propriétaire de cette base soit l'utilisateur **u1** (il faut que ce dernier existe au préalable).

Tous les utilisateurs n'ont pas le droit de créer une base de données. L'utilisateur qui exécute la commande SQL doit avoir soit l'attribut `SUPERUSER` soit l'attribut `CREATEDB`. S'il utilise un autre modèle que celui par défaut, il doit être propriétaire de ce modèle ou le modèle doit être marqué comme étant un modèle officiel (autrement dit la colonne `datistemplate` doit être à `true`).

Voici un exemple complet :

```
postgres=# CREATE DATABASE b1;
CREATE DATABASE
postgres=# CREATE USER u1;
CREATE ROLE
postgres=# CREATE DATABASE b2 OWNER u1;
CREATE DATABASE
postgres=# CREATE USER u2 CREATEDB;
CREATE ROLE
```

NB : pour que la connexion qui suit fonctionne, et sans mot de passe, il faut paramétrer `pg_hba.conf` pour autoriser la connexion de cet utilisateur. Ce sera traité plus bas.

```
postgres=# \c postgres u2
```

You are now connected to database "postgres" as user "u2".

```
postgres=> CREATE DATABASE b3;
```

CREATE DATABASE

```
postgres=> CREATE DATABASE b4 TEMPLATE b2;
ERROR: permission denied to copy database "b2"
postgres=> CREATE DATABASE b4 TEMPLATE b3;
```

CREATE DATABASE

```
postgres=> \c postgres postgres
```

You are now connected to database "postgres" as user "postgres".

```
postgres=# ALTER DATABASE b2 IS_TEMPLATE=true;
```

ALTER DATABASE

```
postgres=# \c postgres u2
```

You are now connected to database "postgres" as user "u2".

```
postgres=> CREATE DATABASE b5 TEMPLATE b2;
```

CREATE DATABASE

```
postgres=> \c postgres postgres
postgres=# \l
```

```
postgres=# \l
```

List of databases						
Name	Owner	Enc...	Collate	Ctype	...	Access privileges
b1	postgres	UTF8	en_US.UTF-8	en_US.UTF-8		
b2	u1	UTF8	en_US.UTF-8	en_US.UTF-8		
b3	u2	UTF8	en_US.UTF-8	en_US.UTF-8		
b4	u2	UTF8	en_US.UTF-8	en_US.UTF-8		
b5	u2	UTF8	en_US.UTF-8	en_US.UTF-8		
postgres	postgres	UTF8	en_US.UTF-8	en_US.UTF-8		
template0	postgres	UTF8	en_US.UTF-8	en_US.UTF-8		=c/postgres + postgres=Ctc/postgres
template1	postgres	UTF8	en_US.UTF-8	en_US.UTF-8		=c/postgres + postgres=Ctc/postgres

L'outil système `createdb` se connecte à la base de données `postgres` et exécute la commande `CREATE DATABASE`, exactement comme ci-dessus. Appelée sans aucun argument, `createdb` crée une base de donnée portant le nom de l'utilisateur connecté (si cette dernière n'existe pas). L'option `--echo` de cette commande permet de voir exactement ce que `createdb` exécute :

```
$ createdb --echo --owner u1 b6
```

```
SELECT pg_catalog.set_config('search_path', '', false)
CREATE DATABASE b6 OWNER u1;
```

Avec une configuration judicieuse des traces de PostgreSQL (`log_min_duration_statement = 0`, `log_connections = on`, `log_disconnections = on`), il est possible de voir cela complètement du point de vue du serveur :

```
[unknown] - LOG:  connection received: host=[local]
[unknown] - LOG:  connection authorized: user=postgres database=postgres
createdb - LOG:  duration: 1.018 ms
                statement: SELECT pg_catalog.set_config('search_path', '', false)
createdb - CREATE DATABASE b6 OWNER u1;
createdb - LOG:  disconnection: session time: 0:00:00.277 user=postgres
                database=postgres
                host=[local]
```

1.2.4 Suppression d'une base



- **SQL :** `DROP DATABASE`
 - droit nécessaire : `SUPERUSER` ou propriétaire de la base
 - prérequis : aucun utilisateur connecté sur la base
 - ou déconnexion forcée (v13)
- **Outil système :** `dropdb`

Supprimer une base de données supprime tous les objets et toutes les données contenues dans la base. La destruction d'une base de données ne peut pas être annulée.

La suppression se fait uniquement avec l'ordre `DROP DATABASE`. Seuls les superutilisateurs et le propriétaire d'une base peuvent supprimer cette base. Cependant, pour que cela fonctionne, il faut qu'aucun utilisateur ne soit connecté à cette base. Si quelqu'un est connecté, un message d'erreur apparaîtra :

```
postgres=# DROP DATABASE b6;
```

```
ERROR:  database "b6" is being accessed by other users
DETAIL:  There are 1 other session(s) using the database.
```

Il faut donc attendre que les utilisateurs se déconnectent, ou leur demander de le faire, voire les déconnecter autoritairement :

```
postgres=# SELECT pg_terminate_backend(pid)
           FROM pg_stat_activity
           WHERE datname='b6';
```

```
pg_terminate_backend
-----
t
```

```
postgres=# DROP DATABASE b6;
```

DROP DATABASE

Là-aussi, PostgreSQL propose un outil système appelé `dropdb` pour faciliter la suppression des bases. Cet outil se comporte comme `createdb`. Il se connecte à la base `postgres` et exécute l'ordre SQL correspondant à la suppression de la base :

```
$ dropdb --echo b5
```

```
SELECT pg_catalog.set_config('search_path', '', false)
DROP DATABASE b5;
```

Contrairement à `createdb`, sans nom de base, `dropdb` ne fait rien.

À partir de la version 13, il est possible d'utiliser la clause `WITH (FORCE)` de l'ordre `DROP DATABASE` ou l'option en ligne de commande `--force` de l'outil `dropdb` pour forcer la déconnexion des utilisateurs.

1.2.5 Modification / configuration



- `ALTER DATABASE`
 - pour modifier quelques méta-données
 - pour ajouter, modifier ou supprimer une configuration
- Catalogue système `pg_db_role_setting`

Avec la commande `ALTER DATABASE`, il est possible de modifier quelques méta-données :

- le nom de la base ;
- son propriétaire ;
- la limite de connexions ;
- le tablespace de la base.

Dans le cas d'un changement de nom ou de tablespace, aucun utilisateur ne doit être connecté à la base pendant l'opération.

Il est aussi possible d'ajouter, de modifier ou de supprimer une configuration spécifique pour une base de données en utilisant la syntaxe suivante :

```
ALTER DATABASE base SET paramètre TO valeur;
```

La configuration spécifique de chaque base de données surcharge toute configuration reçue sur la ligne de commande du processus `postgres` père ou du fichier de configuration `postgresql.conf`. L'ajout d'une configuration avec `ALTER DATABASE` sauvegarde le paramétrage mais ne l'applique pas immédiatement. Il n'est appliqué que pour les prochaines connexions. Notez que les utilisateurs

peuvent cependant modifier ce réglage pendant la session ; il s'agit seulement d'un réglage par défaut, pas d'un réglage forcé.

Voici un exemple complet :

```
b1=# SHOW work_mem;
```

```
work_mem
-----
4MB
```

```
b1=# ALTER DATABASE b1 SET work_mem TO '2MB';
```

```
ALTER DATABASE
```

```
b1=# SHOW work_mem;
```

```
work_mem
-----
4MB
```

```
b1=# \c b1
```

You are now connected to database "b1" as user "postgres".

```
b1=# SHOW work_mem;
```

```
work_mem
-----
2MB
```

Cette configuration est présente même après un redémarrage du serveur. Elle n'est pas enregistrée dans le fichier de configuration `postgresql.conf`, mais dans un catalogue système appelé

`pg_db_role_setting` :

```
b1=# ALTER DATABASE b2 SET work_mem TO '32MB';
```

```
ALTER DATABASE
```

```
b1=# ALTER USER u1 SET maintenance_work_mem TO '256MB';
```

```
ALTER ROLE
```

```
b1=# SELECT * FROM pg_db_role_setting;
```

setdatabase	setrole	setconfig
16384	0	{work_mem=2MB}
16386	0	{work_mem=32MB}
0	16385	{maintenance_work_mem=256MB}

```
b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
setconfig AS "Configuration"
FROM pg_db_role_setting
LEFT JOIN pg_database d ON d.oid=setdatabase
LEFT JOIN pg_roles r ON r.oid=setrole
ORDER BY 1, 2;
```

Base	Utilisateur	Configuration
b1		{work_mem=2MB}
b2		{work_mem=32MB}
	u1	{maintenance_work_mem=256MB}

```
b1=# ALTER DATABASE b3 SET work_mem to '10MB';
```

```
ALTER DATABASE
```

```
b1=# ALTER DATABASE b3 SET maintenance_work_mem to '128MB';
```

```
ALTER DATABASE
```

```
b1=# ALTER DATABASE b3 SET random_page_cost to 3;
```

```
ALTER DATABASE
```

```
b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
       setconfig AS "Configuration"
       FROM pg_db_role_setting
       LEFT JOIN pg_database d ON d.oid=setdatabase
       LEFT JOIN pg_roles r ON r.oid=setrole
       ORDER BY 1, 2;
```

Base	Utilisateur	Configuration
b1		{work_mem=2MB}
b2		{work_mem=32MB}
b3		{work_mem=10MB,maintenance_work_mem=128MB,random_page_cost=3}
	u1	{maintenance_work_mem=256MB}

Pour annuler la configuration d'un paramètre, utilisez :

```
ALTER DATABASE base RESET paramètre;
```

Par exemple :

```
b1=# ALTER DATABASE b3 RESET random_page_cost;
```

```
ALTER DATABASE
```

```
b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
       setconfig AS "Configuration"
       FROM pg_db_role_setting
       LEFT JOIN pg_database d ON d.oid=setdatabase
       LEFT JOIN pg_roles r ON r.oid=setrole
       ORDER BY 1, 2;
```

Base	Utilisateur	Configuration
b1		{work_mem=2MB}
b2		{work_mem=32MB}
b3		{work_mem=10MB,maintenance_work_mem=128MB}
	u1	{maintenance_work_mem=256MB}

Si vous copiez avec `CREATE DATABASE ... TEMPLATE` une base dont certains paramètres ont été configurés spécifiquement pour elle, ces paramètres ne sont pas appliqués à la nouvelle base de données.

1.3 RÔLES



- Utilisateur/groupe
- Liste des rôles
- Création
- Suppression
- Modification
- Gestion des mots de passe

Un rôle peut être vu soit comme un utilisateur de la base de données, soit comme un groupe d'utilisateurs de la base de données, suivant la façon dont le rôle est conçu et configuré. Les rôles peuvent être propriétaires d'objets de la base de données (par exemple des tables) et peuvent affecter des droits sur ces objets à d'autres rôles pour contrôler l'accès à ces objets. De plus, il est possible de donner l'appartenance d'un rôle à un autre rôle, l'autorisant ainsi à utiliser les droits affectés au rôle dont il est membre.

Nous allons voir dans cette partie comment gérer les rôles, en allant de leur création à leur suppression, en passant par leur configuration.

1.3.1 Utilisateurs et groupes



- « Rôle » = utilisateurs et groupes
- Ordres SQL
 - `CREATE/DROP/ALTER USER`
 - `CREATE/DROP/ALTER GROUP`

Les rôles sont disponibles depuis la version 8.1. Auparavant, PostgreSQL avait la notion d'utilisateur et de groupe. Pour conserver la compatibilité avec les anciennes applications, les ordres SQL pour les utilisateurs et les groupes ont été conservés. Il est donc toujours possible de les utiliser mais il est actuellement conseillé de passer par les ordres SQL pour les rôles.

1.3.2 Liste des rôles



- Catalogue système : `pg_roles`
- Dans `psql` : `\du`

La liste des rôles est disponible grâce à un catalogue système appelé `pg_roles`. Il suffit d'un `SELECT` pour récupérer les méta-données sur chaque rôle :

```
postgres=# \x
```

Expanded display is on.

```
postgres=# SELECT * FROM pg_roles LIMIT 3;
```

```
-[ RECORD 1 ]--+-+-----
rolname       | postgres
rolsuper      | t
rolinherit    | t
rolcreatorole | t
rolcreatedb   | t
rolcanlogin   | t
rolreplication| t
rolconnlimit  | -1
rolpassword   | *****
rolvaliduntil |
rolbypassrsls| t
rolconfig     |
oid           | 10
-[ RECORD 2 ]--+-+-----
rolname       | pg_monitor
rolsuper      | f
rolinherit    | t
rolcreatorole | f
rolcreatedb   | f
rolcanlogin   | f
rolreplication| f
rolconnlimit  | -1
rolpassword   | *****
rolvaliduntil |
rolbypassrsls| f
rolconfig     |
oid           | 3373
-[ RECORD 3 ]--+-+-----
rolname       | pg_read_all_settings
rolsuper      | f
rolinherit    | t
rolcreatorole | f
rolcreatedb   | f
rolcanlogin   | f
rolreplication| f
rolconnlimit  | -1
rolpassword   | *****
rolvaliduntil |
rolbypassrsls| f
rolconfig     |
oid           | 3374
```

Voici la signification des différentes colonnes :

- `rolname`, le nom du rôle ;
- `rolsuper`, le rôle a-t-il l'attribut `SUPERUSER` ? ;

- `rolinherit`, le rôle hérite-t-il automatiquement des droits des rôles dont il est membre ? ;
- `rolcreatorole`, le rôle a-t-il le droit de créer des rôles ? ;
- `rolcreatedb`, le rôle a-t-il le droit de créer des bases ? ;
- `rolcanlogin`, le rôle a-t-il le droit de se connecter ? ;
- `rolreplication`, le rôle peut-il être utilisé dans une connexion de réplication ? ;
- `rolconndeflimit`, limite du nombre de connexions simultanées pour ce rôle (`0` indiquant « pas de connexions possibles », `-1` permet d'indiquer qu'il n'y a pas de limite en dehors de la valeur du paramètre `max_connections`) ;
- `rolpassword`, mot de passe du rôle (non affiché) ;
- `rolvaliduntil`, date limite de validité du mot de passe ;
- `rolbypassrls`, le rôle court-circuite-t-il les droits sur les lignes ? ;
- `rolconfig`, configuration spécifique du rôle ;
- `oid`, identifiant système du rôle.

Pour avoir une vue plus simple, il est préférable d'utiliser la métacommande `\du` dans `psql` :

```
postgres=# \du

List of roles
-[ RECORD 1 ]-----
Role name | postgres
Attributes | Superuser, Create role, Create DB, Replication, Bypass RLS
Member of | {}
-[ RECORD 2 ]-----
Role name | u1
Attributes |
Member of | {}
-[ RECORD 3 ]-----
Role name | u2
Attributes | Create DB
Member of | {}
```

Il est à noter que les rôles systèmes ne sont pas affichés. Les rôles systèmes sont tous ceux commençant par `pg_`.

La métacommande `\du` ne fait qu'accéder aux tables systèmes. Par exemple :

```
$ psql -E postgres

psql (13.0)
Type "help" for help.

postgres=# \du
***** QUERY *****
SELECT r.rolname, r.rolsuper, r.rolinherit,
       r.rolcreatorole, r.rolcreatedb, r.rolcanlogin,
       r.rolconndeflimit, r.rolvaliduntil,
       ARRAY(SELECT b.rolname
             FROM pg_catalog.pg_auth_members m
             JOIN pg_catalog.pg_roles b ON (m.roleid = b.oid)
```

```

WHERE m.member = r.oid) as memberof
, r.rolreplication
, r.rolbypassrls
FROM pg_catalog.pg_roles r
WHERE r.rolname !~ '^pg_'
ORDER BY 1;
*****

```

List of roles

```

-[ RECORD 1 ]-----
Role name   | postgres
Attributes | Superuser, Create role, Create DB, Replication, Bypass RLS
Member of  | {}
[...]

```

La requête affichée montre bien que `psql` accède aux catalogues `pg_roles` et `pg_auth_members`.

1.3.3 Création d'un rôle



- **SQL :** `CREATE ROLE`
 - droit nécessaire : `SUPERUSER` ou `CREATEROLE`
 - prérequis : utilisateur inexistant
- **Outil système :** `createuser`
 - attribut `LOGIN` par défaut

L'ordre `CREATE ROLE` est le seul moyen avec PostgreSQL de créer un rôle. Il suffit d'y ajouter le nom du rôle à créer pour que la création se fasse. Il est néanmoins possible d'y ajouter un certain nombre d'options :

- `SUPERUSER`, pour que le nouveau rôle soit superutilisateur (autrement dit, ce rôle a le droit de tout faire une fois connecté à une base de données) ;
- `CREATEDB`, pour que le nouveau rôle ait le droit de créer des bases de données ;
- `CREATEROLE`, pour que le nouveau rôle ait le droit de créer un rôle ;
- `INHERIT`, pour que le nouveau rôle hérite automatiquement des droits des rôles dont il est membre ;
- `LOGIN`, pour que le nouveau rôle ait le droit de se connecter ;
- `REPLICATION`, pour que le nouveau rôle puisse se connecter en mode répliation ;
- `BYPASSRLS`, pour que le nouveau rôle puisse ne pas être vérifié pour les sécurités au niveau ligne ;
- `CONNECTION LIMIT`, pour limiter le nombre de connexions simultanées pour ce rôle ;
- `PASSWORD`, pour préciser le mot de passe de ce rôle ;

- `VALID UNTIL`, pour indiquer la date limite de validité du mot de passe ;
- `IN ROLE`, pour indiquer à quel rôle ce rôle appartient ;
- `IN GROUP`, pour indiquer à quel groupe ce rôle appartient ;
- `ROLE`, pour indiquer les membres de ce rôle ;
- `ADMIN`, pour indiquer les membres de ce rôles (les nouveaux membres ayant en plus la possibilité d'ajouter d'autres membres à ce rôle) ;
- `USER`, pour indiquer les membres de ce rôle ;
- `SYSID`, pour préciser l'identifiant système, mais est ignoré.

Par défaut, un rôle n'a aucun attribut (ni superutilisateur, ni le droit de créer des rôles ou des bases, ni la possibilité de se connecter en mode réplication, ni la possibilité de se connecter).

Voici quelques exemples simples :

```
postgres=# CREATE ROLE u3;
```

```
CREATE ROLE
```

```
postgres=# CREATE ROLE u4 CREATEROLE;
```

```
CREATE ROLE
```

```
postgres=# CREATE ROLE u5 LOGIN IN ROLE u2;
```

```
CREATE ROLE
```

```
postgres=# CREATE ROLE u6 ROLE u5;
```

```
CREATE ROLE
```

```
postgres=# \du
```

```
List of roles
```

```
-[ RECORD 1 ]-----
Role name | postgres
Attributes | Superuser, Create role, Create DB, Replication, Bypass RLS
Member of | {}
-[ RECORD 2 ]-----
Role name | u1
Attributes |
Member of | {}
-[ RECORD 3 ]-----
Role name | u2
Attributes | Create DB
Member of | {}
-[ RECORD 4 ]-----
Role name | u3
Attributes | Cannot login
Member of | {}
-[ RECORD 5 ]-----
Role name | u4
Attributes | Create role, Cannot login
Member of | {}
-[ RECORD 6 ]-----
Role name | u5
```

```

Attributes |
Member of  | {u2,u6}
-[ RECORD 7 ]-----
Role name  | u6
Attributes | Cannot login
Member of  | {}

```

Tous les rôles n'ont pas le droit de créer un rôle. Le rôle qui exécute la commande SQL doit avoir soit l'attribut `SUPERUSER` soit l'attribut `CREATEROLE`. Un utilisateur qui a l'attribut `CREATEROLE` pourra créer tout type de rôles sauf des superutilisateurs.

Voici un exemple complet :

```

postgres=# CREATE ROLE u7 LOGIN CREATEROLE;
CREATE ROLE
postgres=# \c postgres u7
You are now connected to database "postgres" as user "u7".
postgres=> CREATE ROLE u8 LOGIN;
CREATE ROLE
postgres=> CREATE ROLE u9 LOGIN CREATEDB;
CREATE ROLE
postgres=> CREATE ROLE u10 LOGIN SUPERUSER;
ERROR:  must be superuser to create superusers
postgres=> \du

```

Role name	List of roles Attributes	Member of
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS	{}
u1		{}
u2	Create DB	{}
u3	Cannot login	{}
u4	Create role, Cannot login	{}
u5		{u2,u6}
u6	Cannot login	{}
u7	Create role	{}
u8		{}
u9	Create DB	{}

Il est toujours possible d'utiliser les ordres SQL `CREATE USER` et `CREATE GROUP`. PostgreSQL les comprend comme étant l'ordre `CREATE ROLE`. Dans le premier cas (`CREATE USER`), il ajoute automatiquement l'option `LOGIN`.

Il est possible de créer un utilisateur (dans le sens, rôle avec l'attribut `LOGIN`) sans avoir à se rappeler de la commande SQL. Le plus simple est certainement l'outil `createuser`, livré avec PostgreSQL, mais c'est aussi possible avec n'importe quel autre outil d'administration de bases de données PostgreSQL.

L'outil système `createuser` se connecte à la base de données `postgres` et exécute la commande `CREATE ROLE`, exactement comme ci-dessus, avec par défaut l'option `LOGIN`. L'option `--echo` de cette commande nous permet de voir exactement ce que `createuser` exécute :

```
$ createuser --echo u10 --superuser
SELECT pg_catalog.set_config('search_path', '', false)
CREATE ROLE u10 SUPERUSER CREATEDB CREATEROLE INHERIT LOGIN;
```

Il est à noter que `createuser` est un programme interactif. Avant la version 9.2, si le nom du rôle n'est pas indiqué, l'outil demandera le nom du rôle à créer. De même, si au moins un attribut n'est pas explicitement indiqué, il demandera les attributs à associer à ce rôle :

```
$ createuser u11
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) y
Shall the new role be allowed to create more new roles? (y/n) n
```

Depuis la version 9.2, il crée un utilisateur avec les valeurs par défaut (équivalent à une réponse `n` à toutes les questions). Pour retrouver le mode interactif, il faut utiliser l'option `--interactive`.

1.3.4 Suppression d'un rôle



- **SQL :** `DROP ROLE`
 - droit nécessaire : `SUPERUSER` ou `CREATEROLE`
 - prérequis : rôle existant, rôle ne possédant pas d'objet
- **Outil système :** `dropuser`

La suppression d'un rôle se fait uniquement avec l'ordre `DROP ROLE`. Seuls les utilisateurs disposant des attributs `SUPERUSER` et `CREATEROLE` peuvent supprimer des rôles. Cependant, pour que cela fonctionne, il faut que le rôle à supprimer ne soit pas propriétaire d'objets dans l'instance. S'il est propriétaire, un message d'erreur apparaîtra :

```
postgres=> DROP ROLE u1;
ERROR:  role "u1" cannot be dropped because some objects depend on it
DETAIL:  owner of database b2
```

Il faut donc changer le propriétaire des objets en question ou supprimer les objets. Vous pouvez utiliser respectivement les ordres `REASSIGN OWNED` et `DROP OWNED` pour cela.

Un rôle qui n'a pas l'attribut `SUPERUSER` ne peut pas supprimer un rôle qui a cet attribut :

```
postgres=> DROP ROLE u10;
```

```
ERROR: must be superuser to drop superusers
```

Par contre, il est possible de supprimer un rôle qui est connecté. Le rôle connecté aura des possibilités limitées après sa suppression. Par exemple, il peut toujours lire quelques tables systèmes mais il ne peut plus créer d'objets.

Là-aussi, PostgreSQL propose un outil système appelé `dropuser` pour faciliter la suppression des rôles. Cet outil se comporte comme `createrole` : il se connecte à la base PostgreSQL et exécute l'ordre SQL correspondant à la suppression du rôle :

```
$ dropuser --echo u10
```

```
SELECT pg_catalog.set_config('search_path', '', false)
DROP ROLE u10;
```

Sans spécifier le nom de rôle sur la ligne de commande, `dropuser` demande le nom du rôle à supprimer.

1.3.5 Modification d'un rôle



- `ALTER ROLE`
 - pour modifier quelques méta-données
 - pour ajouter, modifier ou supprimer une configuration
- Catalogue système : `pg_db_role_setting`

Avec la commande `ALTER ROLE`, il est possible de modifier quelques méta-données du rôle :

- son nom ;
- son mot de passe ;
- sa limite de validité ;
- ses attributs :
 - `SUPERUSER` ;
 - `CREATEDB` ;
 - `CREATEROLE` ;
 - `CREATEUSER` ;
 - `INHERIT` ;
 - `LOGIN` ;
 - `REPLICATION` ;
 - `BYPASSRLS` .

Toutes ces opérations peuvent s'effectuer alors que le rôle est connecté à la base.

Il est aussi possible d'ajouter, de modifier ou de supprimer une configuration spécifique pour un rôle en utilisant la syntaxe suivante :

```
ALTER ROLE rôle SET paramètre TO valeur;
```

La configuration spécifique de chaque rôle surcharge toute configuration reçue sur la ligne de commande du processus postgres père ou du fichier de configuration `postgresql.conf`, mais aussi la configuration spécifique de la base de données où le rôle est connecté. L'ajout d'une configuration avec `ALTER ROLE` sauvegarde le paramétrage mais ne l'applique pas immédiatement. Il n'est appliqué que pour les prochaines connexions. Notez que les rôles peuvent cependant modifier ce réglage pendant la session ; il s'agit seulement d'un réglage par défaut, pas d'un réglage forcé.

Voici un exemple complet :

```
$ psql -U u2 postgres
psql (13.0)
Type "help" for help.

postgres=> SHOW work_mem;

 work_mem
-----
 4MB

postgres=> ALTER ROLE u2 SET work_mem TO '20MB';

ALTER ROLE

postgres=> SHOW work_mem;

 work_mem
-----
 4MB

postgres=> \c - u2

You are now connected to database "postgres" as user "u2".

postgres=> SHOW work_mem;

 work_mem
-----
 20MB
```

Cette configuration est présente même après un redémarrage du serveur. Elle n'est pas enregistrée dans le fichier de configuration `postgresql.conf` mais dans un catalogue système appelé `pg_db_role_setting` :

```
b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
      setconfig AS "Configuration"
FROM pg_db_role_setting
LEFT JOIN pg_database d ON d.oid=setdatabase
LEFT JOIN pg_roles r ON r.oid=setrole
ORDER BY 1, 2;
```


Base	Utilisateur	Configuration
b1		{work_mem=2MB}
b2		{work_mem=32MB}
b3		{work_mem=10MB,maintenance_work_mem=128MB}
	u1	{maintenance_work_mem=256MB}
	u2	{work_mem=20MB}

Il est aussi possible de configurer un paramétrage spécifique pour un utilisateur et une base données :

```
postgres=# ALTER ROLE u2 IN DATABASE b1 SET work_mem to '10MB';
```

```
ALTER ROLE
```

```
postgres=# \c postgres u2
```

```
You are now connected to database "postgres" as user "u2".
```

```
postgres=> SHOW work_mem;
```

```
work_mem
-----
20MB
```

```
postgres=> \c b1 u2
```

```
You are now connected to database "b1" as user "u2".
```

```
b1=> SHOW work_mem;
```

```
work_mem
-----
10MB
```

```
b1=> \c b1 u1
```

```
You are now connected to database "b1" as user "u1".
```

```
b1=> SHOW work_mem;
```

```
work_mem
-----
2MB
```

```
b1=> \c postgres u1
```

```
You are now connected to database "postgres" as user "u1".
```

```
postgres=> SHOW work_mem;
```

```
work_mem
-----
4MB
```

```
b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
      setconfig AS "Configuration"
      FROM pg_db_role_setting
      LEFT JOIN pg_database d ON d.oid=setdatabase
      LEFT JOIN pg_roles r ON r.oid=setrole
      ORDER BY 1, 2;
```

Base	Utilisateur	Configuration
b1	u2	{work_mem=10MB}
b1		{work_mem=2MB}
b2		{work_mem=32MB}
b3		{work_mem=10MB,maintenance_work_mem=128MB}
	u1	{maintenance_work_mem=256MB}
	u2	{work_mem=20MB}

Pour annuler la configuration d'un paramètre pour un rôle, utilisez :

```
ALTER ROLE rôle RESET paramètre;
```



Attention : la prise en compte de ces options dans les sauvegardes est un point délicat. Il est détaillé dans notre module de formation sur les sauvegardes logiques.

Après sa création, il est toujours possible d'ajouter et de supprimer un rôle dans un autre rôle. Pour cela, il est possible d'utiliser les ordres `GRANT` et `REVOKE` :

```
GRANT role_groupe TO role_utilisateur;
```

Il est aussi possible de passer par la commande `ALTER GROUP` de cette façon :

```
ALTER GROUP role_groupe ADD USER role_utilisateur;
```

1.3.6 Mot de passe



Le mot de passe ne concerne pas toutes les méthodes d'authentification

- Par défaut : l'utilisateur n'a pas de mot de passe
 - donc pas de connexion possible
- Modification :

```
ALTER ROLE u1 PASSWORD 'supersecret'; -- dangereux
```

- Attention à ne pas l'afficher dans les traces
 - fournir un mot de passe déjà chiffré
 - utiliser `\password` ou `createuser`

```
createuser --login --echo --pwprompt u1
```

Certaines méthodes d'authentification n'ont pas besoin de mot de passe (comme `peer` pour les connexions depuis le serveur même), ou délèguent l'authentification à un système extérieur (comme

`ldap`). Par défaut, les utilisateurs n'ont pas de mot de passe. Si la méthode en exige un, ils ne pourront pas se connecter.



Il est très fortement conseillé d'utiliser une méthode d'authentification avec saisie du mot de passe.

On peut le créer ainsi :

```
ALTER ROLE u1 PASSWORD 'supersecret';
```

À partir de là, avec une méthode d'authentification bien configurée, le mot de passe sera demandé. Il faudra, dans cet exemple, saisir « supersecret » pour que la connexion se fasse.



ATTENTION ! Le mot de passe peut apparaître en clair dans les traces ! Notamment si `log_min_duration_statement` vaut 0.

```
$ grep PASSWORD $PGDATA/log/traces.log
```

```
psql - LOG: duration: 1.865 ms
      statement: ALTER ROLE u1 PASSWORD 'supersecret';
```

La vue système `pg_stat_activity` ou l'extension `pg_stat_statements`, voire d'autres outils, sont aussi susceptibles d'afficher la commande et donc le mot de passe en clair.

Il est donc essentiel de s'arranger pour que seules des personnes de confiance aient accès aux traces et vues systèmes. Il est certes possible de demander temporairement la désactivation des traces pendant le changement de mot de passe (si l'on est superutilisateur) :

```
SET log_min_duration_statement TO -1;
SET log_statement TO none;
ALTER ROLE u1 PASSWORD 'supersecret';
```

```
$ grep PASSWORD $PGDATA/log/postgresql-Mon.log
[rien]
```

Cependant, cela ne règle pas le cas de `pg_stat_statements` et `pg_stat_activity`.

De manière générale, il est donc chaudement conseillé de ne renseigner que des mots de passe chiffrés. C'est très simple en mode interactif avec `psql`, la métacommande `\password` opère le chiffrement :

```
\password u1
```

Saisissez le nouveau mot de passe :
Saisissez-le à nouveau :

L'ordre effectivement envoyé au serveur et éventuellement visible dans les traces sera :

```
ALTER USER u1 PASSWORD 'SCRAM-SHA-256$4096:KcHoLSZE...Hj81r3w='
```

De même si on crée le rôle depuis le shell avec `createuser` :

```
createuser --login --echo --pwprompt u1
```

Saisir le mot de passe pour le nouveau rôle :

Le saisir de nouveau :

```
SELECT pg_catalog.set_config('search_path', '', false);
```

```
CREATE ROLE u1 PASSWORD
```

```
↪ 'SCRAM-SHA-256$4096:/LVaGESxmDwyF92urT...hS0k0xIpwWAbTpW1i9peGIg='
```

```
NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN NOREPLICATION NOBYPASSRLS;
```



Lors des changements de mots de passe, ne pas oublier qu'il reste un risque de fuite aussi au niveau des outils système, par exemple l'historique du shell, l'affichage de la liste des processus ou les traces système !

1.3.7 Sécurité des mots de passe



```
password_encryption = "scram-sha-256"
```

- SCRAM-SHA-256 : défaut (≥ v14)
- MD5 : dépassé, mais défaut jusque v13 !
- Sécurité :
 - date limite sur le mot de passe (pas le rôle)
 - pas de vérification de la force du mot de passe
 - pas de limite de tentatives échouées (tester `fail2ban`)
 - itérations SCRAM (v16)

Le mot de passe est chiffré en interne, et visible dans les sauvegardes avec `pg_dumpall -g`, ou dans la vue système `pg_authid`.

Il existe deux méthodes de chiffrement : SCRAM-SHA-256 (fortement conseillée) et l'historique MD5. Elles sont éventuellement utilisables simultanément pour des utilisateurs différents.

L'exemple suivant montre que la méthode de chiffrement peut différer selon les rôles, en fonction de la valeur du paramètre `password_encryption` au moment de la mise en place du mot de passe :

```
SET password_encryption TO "scram-sha-256";
```

```
CREATE ROLE u12 LOGIN PASSWORD 'supersecret';
```

```
SELECT * FROM pg_authid WHERE rolname IN ('u1', 'u12') ORDER BY 1;
```

```

-[ RECORD 1 ]-----
rolname      | u1
rolsuper     | f
rolinherit   | t
rolcreatorole | f
rolcreatedb  | f
rolcanlogin  | t
rolreplication | f
rolbypassrpls | f
rolconnlimit | -1
rolpassword  | md5fb75f17111cea61e62b54ab950dd1268
rolvaliduntil |
-[ RECORD 2 ]-----
rolname      | u12
rolsuper     | f
rolinherit   | t
rolcreatorole | f
rolcreatedb  | f
rolcanlogin  | t
rolreplication | f
rolbypassrpls | f
rolconnlimit | -1
rolpassword  | SCRAM-SHA-256$4096:0/uC6oDNuQW08H9pMaVg8g==$nDUpGSeFH0ZMd
              | TcbWR13NPELJubGg7PduiJjX/Hyt/M=:PSUzE+rP5g4f6mb5sFDRq/Hds
              | OrLvfyYew9ZIdz0/GDw=
rolvaliduntil |

```

Chiffrement SCRAM-SHA-256 :

SCRAM-SHA-256 n'est la méthode par défaut que depuis PostgreSQL 14, bien que disponible depuis PostgreSQL 10. Elle est plus complexe et plus sûre que l'ancienne méthode MD5. Notamment, le même mot de passe entré plusieurs fois pour un même utilisateur, même sur la même instance, donnera des versions chiffrées à chaque fois différentes, mais interchangeables.

Un chiffrement SCRAM-SHA-256 est de la forme :

```
SCRAM-SHA-256$<sel>:<nombre d'itérations>$<hash>
```

Pour quelques détails d'implémentation et une comparaison avec MD5, voir par exemple cette présentation de Jonathan Katz¹.

Générer soi-même des mots de passe chiffrés en SCRAM-SHA-256 en-dehors de `psql` est plus compliqué qu'avec MD5, et les outils comme `\password` s'appuient souvent sur les fonctions fournies par la bibliothèque `libpq`. Cette dernière sert aussi de base à la bibliothèque python `psycopg3`² et sa fonction `PGconn.encrypt_password()`³. Indépendamment de la `libpq`, il existe aussi un script python de Jonathan Katz⁴ (version 3.6 minimum).

Depuis la version 16 il est possible d'ajuster le nombre d'itérations avec le paramètre `scram_iterations`. Le défaut de 4096 est un compromis. Monter plus haut permet de lutter contre les attaques par force

¹<https://fr.slideshare.net/jkatz05/safely-protect-postgresql-passwords-tell-others-to-scam>

²<https://github.com/psycopg/psycopg>

³https://www.psycopg.org/psycopg3/docs/api/pq.html#psycopg.pq.PGconn.encrypt_password

⁴https://gist.github.com/jkatz/e0a1f52f66fa03b732945f6eb94d9c21#file-encrypt_password-py-L20

brute. Réduire ce paramètre permet de parer aux problèmes de performances depuis certains terminaux peu puissants, ou en cas de connexions très fréquentes (le début de la discussion entre les développeurs⁵ illustre bien ces deux contraintes).

Chiffrement MD5 :

Le chiffrement `md5` est celui par défaut jusque PostgreSQL 13 inclus. Il consiste à calculer la somme MD5 du mot de passe concaténé au nom du rôle (pour que deux utilisateurs de même mot de passe n'ait pas la même version chiffrée) ; puis « md5 » est ajouté devant. De manière plus générale, l'algorithme MD5 est considéré comme trop faible de nos jours.

Un autre problème de sécurité est que la version chiffrée d'un mot de passe est identique sur deux instances différentes pour un même nom d'utilisateur, ce qui ouvre la possibilité d'attaques par *rainbow tables*⁶.

Un inconvénient plus mineur du chiffrement MD5 est qu'il utilise le nom de l'utilisateur : en cas de changement de ce nom, il faudra de nouveau rentrer le mot de passe pour que son stockage chiffré soit correct.

Pour éviter de fournir un mot de passe en clair à PostgreSQL, il est facile de le chiffrer en MD5 avant de le fournir à PostgreSQL :

```
$ echo -n "supersecretu1" | md5sum
fb75f17111cea61e62b54ab950dd1268 -
```

```
ALTER ROLE u1 PASSWORD 'md5fb75f17111cea61e62b54ab950dd1268';
```

Dans les traces on ne trouvera que la version chiffrée :

```
psql - LOG: duration: 2.100 ms statement: ALTER ROLE u1
PASSWORD 'md5fb75f17111cea61e62b54ab950dd1268';
```

Ceci fait la même chose sous forme de script :

```
set +o history      # suspend l'historique du shell
MDP=supersecret
U=u1
psql -X --echo-all -c \
"$ (echo ALTER ROLE ${U} PASSWORD \ 'md5$(echo -n ${MDP}${U}|md5sum|cut -f1 -d' '))';"
```

```
...
ALTER ROLE u1 PASSWORD 'md5fb75f17111cea61e62b54ab950dd1268';
...
```

Cohabitation de mots de passe SCRAM-SHA-256 et MD5 :

Lors de la mise à jour d'une ancienne instance, il n'est pas forcément possible de ré-entrer immédiatement tous les mots de passe existants chiffrés en MD5. De plus, certains outils clients un peu anciens pourraient ne pas supporter SCRAM-SHA-256 (voir le wiki⁷). Changer la méthode d'authentification selon l'utilisateur est donc utile.

⁵<https://www.postgresql.org/message-id/F72E7BC7-189F-4B17-BF47-9735EB72C364@yesql.se>

⁶https://fr.wikipedia.org/wiki/Rainbow_table

⁷https://wiki.postgresql.org/wiki/List_of_drivers

La méthode d'authentification SCRAM-SHA-256 doit rester le défaut, et tout mot de passe réentré sera chiffré en SCRAM-SHA-256 :

```
password_encryption = "scram-sha-256"
```

Si le mot de passe est stocké au format SCRAM-SHA-256, une authentification paramétrée sur `md5` ou `password` dans le fichier `pg_hba.conf` fonctionnera aussi. Cela facilite la migration progressive des utilisateurs de `md5` à `scram-sha-256`, qui peuvent réentrer leur mot de passe quand ils veulent.

Par contre, indiquer la méthode `scram-sha-256` dans `pg_hba.conf` impose un chiffage en SCRAM-SHA-256 et interdit d'utiliser MD5.

On peut mixer les deux méthodes dans `pg_hba.conf` si le besoin se fait sentir, par exemple pour n'utiliser `md5` que pour une seule application (avec un compte dédié) avec un ancien client :

```
# vieille application (avant le cas général)
host  compta mathusalem 192.168.66.66/32 md5
# authentification habituelle
host  all      all      192.168.66.0/24  scram-sha-256
```

Dates de validité des mots de passe :

Les mots de passe ont une date de validité mais pas les rôles eux-mêmes. Par exemple, il reste possible de se connecter en local par la méthode `peer` même si le mot de passe a expiré.

Sécurité des mots de passe :

Enfin, il est à noter que PostgreSQL ne vérifie pas la faiblesse d'un mot de passe. Il est certes possible d'installer une extension appelée `passwordcheck` (voir sa documentation⁸).

```
postgres=# ALTER ROLE u1 PASSWORD 'supersecret';
ERROR: password must contain both letters and nonletters
```

Il est possible de modifier le code source de cette extension pour y ajouter les règles convenant à votre cas, ou d'utiliser la bibliothèque `Cracklib`. Des extensions de ce genre, extérieures au projet, existent. Cependant, ces outils exigent que le mot de passe soit fourni en clair, et donc sujet à une fuite (dans les traces par exemple), ce qui, répétons-le, est fortement déconseillé !

Un rôle peut tenter de se connecter autant de fois qu'il le souhaite, ce qui expose à des attaques de type force brute. Il est possible d'interdire toute connexion à partir d'un certain nombre de connexions échouées si vous utilisez une méthode d'authentification externe qui le gère (comme PAM, LDAP ou Active Directory). Vous pouvez aussi obtenir cette fonctionnalité en utilisant un outil comme `fail2ban`. Sa configuration est détaillée dans notre base de connaissances⁹.

⁸<https://docs.postgresql.fr/current/passwordcheck.html>

⁹<https://kb.dalibo.com/fail2ban>

1.4 DROITS SUR LES OBJETS



- Droits sur les objets
- Droits sur les méta-données
- Héritage des droits
- Changement de rôle

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

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

1.4.1 Droits sur les objets



- Donner un droit :


```
GRANT USAGE ON SCHEMA unschema TO utilisateur ;
GRANT SELECT,DELETE,INSERT ON TABLE matable TO utilisateur ;
```
- Retirer un droit :


```
REVOKE UPDATE ON TABLE matable FROM utilisateur ;
```
- Droits spécifiques pour chaque type d'objets :


```
ALTER DEFAULT PRIVILEGES IN SCHEMA ...
ALTER DEFAULT PRIVILEGES FOR ROLE ...
```
- Avoir le droit de donner le droit :


```
WITH GRANT OPTION
```

 - Groupe implicite : `public`
 - Schéma par défaut : `public` lisible par tous (≤ 14) !

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

Par défaut, seul le propriétaire a des droits sur son objet. Les superutilisateurs n'ont pas de droit spé-

cifique sur les objets mais étant donné leur statut de superutilisateur, ils peuvent tout faire sur tous les objets.

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

GRANT droits **ON** type_objet nom_objet **TO** rôle

Les droits disponibles dépendent du type d'objet visé. Par exemple, il est possible de donner le droit `SELECT` sur une table mais pas sur une fonction. Une fonction ne se lit pas, elle s'exécute. Il est donc possible de donner le droit `EXECUTE` sur une fonction.

La liste complète des droits figure dans la documentation officielle¹⁰.

Il faut donner les droits aux différents objets séparément. De plus, donner le droit `ALL` sur une base de données donne tous les droits sur la base de données, autrement dit l'objet base de donnée, pas sur les objets à l'intérieur de la base de données. `GRANT` n'est pas une commande récursive. Prenons un exemple :

```
b1=# CREATE ROLE u20 LOGIN;
```

```
CREATE ROLE
```

```
b1=# CREATE ROLE u21 LOGIN;
```

```
CREATE ROLE
```

```
b1=# \c b1 u20
```

```
You are now connected to database "b1" as user "u20".
```

```
b1=> CREATE SCHEMA s1;
```

```
ERROR: permission denied for database b1
```

```
b1=> \c b1 postgres
```

```
You are now connected to database "b1" as user "postgres".
```

```
b1=# GRANT CREATE ON DATABASE b1 TO u20;
```

```
GRANT
```

```
b1=# \c b1 u20
```

```
You are now connected to database "b1" as user "u20".
```

```
b1=> CREATE SCHEMA s1;
```

```
CREATE SCHEMA
```

```
b1=> CREATE TABLE s1.t1 (c1 integer);
```

```
CREATE TABLE
```

```
b1=> INSERT INTO s1.t1 VALUES (1), (2);
```

¹⁰<https://docs.postgresql.fr/current/sql-grant.html>

INSERT 0 2

b1=> **SELECT * FROM** s1.t1;

```
c1
----
 1
 2
```

b1=> \c b1 u21

You are now connected to database "b1" as user "u21".

b1=> **SELECT * FROM** s1.t1;

```
ERROR: permission denied for schema s1
LINE 1: SELECT * FROM s1.t1;
                        ^
```

b1=> \c b1 u20

You are now connected to database "b1" as user "u20".

b1=> **GRANT SELECT ON TABLE** s1.t1 **TO** u21;

GRANT

b1=> \c b1 u21

You are now connected to database "b1" as user "u21".

b1=> **SELECT * FROM** s1.t1;

```
ERROR: permission denied for schema s1
LINE 1: SELECT * FROM s1.t1;
                        ^
```

b1=> \c b1 u20

You are now connected to database "b1" as user "u20".

b1=> **GRANT USAGE ON SCHEMA** s1 **TO** u21;

GRANT

b1=> \c b1 u21

You are now connected to database "b1" as user "u21".

b1=> **SELECT * FROM** s1.t1;

```
c1
----
 1
 2
```

b1=> **INSERT INTO** s1.t1 **VALUES** (3);

```
ERROR: permission denied for relation t1
```

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

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

Notez aussi que, lors de la création d'une base, PostgreSQL ajoute automatiquement un schéma nommé `public`. Avant la version 15, tous les droits sont donnés sur ce schéma à un pseudo-rôle, lui aussi appelé `public`, et dont tous les rôles existants et à venir sont membres d'office. À partir de la version 15, le schéma `public` appartient au propriétaire de la base et aucun droit par défaut n'est donné aux autres utilisateurs.



Avec une version antérieure à la version 15, n'importe quel utilisateur peut donc, par défaut, créer des tables dans le schéma `public` de toute base où il peut se connecter (mais ne peut lire les tables créées là par d'autres, sans droit supplémentaire) !

Dans une logique de sécurisation, avant la version 15, il faut donc penser à enlever les droits à `public`. Une fausse bonne idée est de tout simplement supprimer le schéma `public`, ou de le récréer (par défaut, sans droits pour le groupe `public`). Cependant, une sauvegarde logique serait restaurée dans une base qui, par défaut, aurait à nouveau un schéma `public` ouvert à tous. Une révocation explicite des droits se retrouvera par contre dans une sauvegarde :

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

(Noter la subtilité de syntaxe : `GRANT... TO...` et `REVOKE... FROM...`)



Nombre de scripts et outils peuvent tomber en erreur sans ces droits. Il faudra remonter cela aux auteurs en tant que bugs.

Cette modification peut être faite aussi dans la base `template1` (qui sert de modèle à toute nouvelle base), sur toute nouvelle instance.

Enfin il est possible d'ajouter des droits pour des objets qui n'ont pas encore été créés. En fait, la commande `ALTER DEFAULT PRIVILEGES` permet de donner des droits par défaut à certains rôles. De cette façon, sur un schéma qui a tendance à changer fréquemment, il n'est plus nécessaire de se préoccuper des droits sur les objets.

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

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

```
b1=# CREATE TABLE t2 (id integer);
```

CREATE TABLE

```
b1=# INSERT INTO t2 VALUES (1);
```

```
INSERT 0 1
```

```
b1=# SELECT * FROM t2;
```

```
 id
----
  1
```

```
b1=# \c b1 u1
```

You are now connected to database "b1" as user "u1".

```
b1=> SELECT * FROM t2;
```

```
ERROR: permission denied for relation t2
```

```
b1=> \c b1 postgres
```

You are now connected to database "b1" as user "postgres".

```
b1=# GRANT SELECT ON TABLE t2 TO u1;
```

```
GRANT
```

```
b1=# \c b1 u1
```

You are now connected to database "b1" as user "u1".

```
b1=> SELECT * FROM t2;
```

```
 id
----
  1
```

```
b1=> \c b1 u2
```

You are now connected to database "b1" as user "u2".

```
b1=> SELECT * FROM t2;
```

```
ERROR: permission denied for relation t2
```

```
b1=> \c b1 u1
```

You are now connected to database "b1" as user "u1".

```
b1=> GRANT SELECT ON TABLE t2 TO u2;
```

```
WARNING: no privileges were granted for "t2"
```

```
GRANT
```

```
b1=> \c b1 postgres
```

You are now connected to database "b1" as user "postgres".

```
b1=# GRANT SELECT ON TABLE t2 TO u1 WITH GRANT OPTION;
```

```
GRANT
```

```
b1=# \c b1 u1
```

You are now connected to database "b1" as user "u1".

```
b1=> GRANT SELECT ON TABLE t2 TO u2;
```

```
GRANT
```

```
b1=> \c b1 u2
```

You are now connected to database "b1" as user "u2".

```
b1=> SELECT * FROM t2;
```

```
 id
----
  1
```

1.4.2 Afficher les droits



- Colonne `*acl` sur les tables systèmes
 - `dataacl` pour `pg_database`
 - `relacl` pour `pg_class`
- Codage `role1=xxxx/role2` (format `aclitem`)
 - `role1` : rôle concerné par les droits
 - `xxxx` : droits parmi `xxxx`
 - `role2` : rôle qui a donné les droits
- Sous `psql` :
 - `\dp` et `\z`
 - `df`, `\dconfig+` etc..

Les colonnes `*acl` des catalogues systèmes indiquent les droits sur un objet. Leur contenu est un codage au format `aclitem` indiquant le rôle concerné, ses droits, et qui lui a fourni ces droits (ou le propriétaire de l'objet, si celui qui a fourni les droits est un superutilisateur).

Les droits sont codés avec des lettres. Les voici avec leur signification :

- `r` pour la lecture (`SELECT`);
- `w` pour les modifications (`UPDATE`);
- `a` pour les insertions (`INSERT`);
- `d` pour les suppressions (`DELETE`);

- `D` pour la troncation (`TRUNCATE`) ;
- `x` pour l'ajout de clés étrangères ;
- `t` pour l'ajout de triggers ;
- `X` pour l'exécution de routines ;
- `U` pour l'utilisation (d'un schéma par exemple) ;
- `C` pour la création d'objets permanents (tables ou vues par exemple) ;
- `c` pour la connexion (spécifique aux bases de données) ;
- `T` pour la création d'objets temporaires (tables ou index temporaires) ;
- `s` pour la modification d'un paramètre superutilisateur avec `SET` ;
- `A` pour la modification d'un paramètre avec `ALTER SYSTEM`.

1.4.3 Droits sur les métadonnées



- Seul le propriétaire peut changer la structure d'un objet
 - le renommer
 - le changer de schéma ou de tablespace
 - lui ajouter/retirer des colonnes
- Un seul propriétaire
 - peut être un utilisateur ou un groupe
- `REASSIGN OWNER` / `DROP OWNED`

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

```
b1=# \c b1 u21
```

```
You are now connected to database "b1" as user "u21".
```

```
b1=> ALTER TABLE s1.t1 ADD COLUMN c2 text;
```

```
ERROR: must be owner of relation t1
```

```
b1=> \c b1 u20
```

```
You are now connected to database "b1" as user "u20".
```

```
b1=> GRANT u20 TO u21;
```

```
GRANT ROLE
```

```
b1=> \du
```

Role name	List of roles Attributes	Member of
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS	{}
u1		{}
u11	Create DB	{}
u12		{}
u2	Create DB	{}
u20		{}
u21		{u20}
u3	Cannot login	{}
u4	Create role, Cannot login	{}
u5		{u2,u6}
u6	Cannot login	{}
u7	Create role	{}
u8		{}
u9	Create DB	{}

b1=> \c b1 u21

You are now connected to database "b1" as user "u21".

b1=> ALTER TABLE s1.t1 ADD COLUMN c2 text;

ALTER TABLE

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

b1=# \d

List of relations			
Schema	Name	Type	Owner
public	t1	table	u2
public	t2	table	u21

b1=# REASSIGN OWNED BY u21 TO u1;

REASSIGN OWNED

b1=# \d

List of relations			
Schema	Name	Type	Owner
public	t1	table	u2
public	t2	table	u1

b1=# DROP OWNED BY u1;

DROP OWNED

b1=# \d

List of relations			
Schema	Name	Type	Owner
public	t1	table	u2

1.4.4 Droits plus globaux 1/2



- Rôles systèmes d'administration
 - `pg_signal_backend`
 - `pg_database_owner` (14+)
 - `pg_checkpoint` (15+)
 - `pg_reserved_connections` (16+)
- Rôles systèmes de supervision
 - `pg_read_all_stats`
 - `pg_read_all_settings`
 - `pg_stat_scan_tables`
 - `pg_monitor`

Certaines fonctionnalités nécessitent l'attribut `SUPERUSER` alors qu'il serait bon de pouvoir les effectuer sans avoir ce droit suprême. Cela concerne principalement l'administration, la sauvegarde et la supervision.

Après beaucoup de discussions, les développeurs de PostgreSQL ont décidé de créer des rôles systèmes permettant d'avoir plus de droits. Le premier rôle de ce type est `pg_signal_backend` qui donne le droit d'exécuter les fonctions `pg_cancel_backend()` et `pg_terminate_backend()`, même en simple utilisateur sur des requêtes autres que les siennes :

```
postgres=# \c - u1
```

```
You are now connected to database "postgres" as user "u1".
```

```
postgres=> SELECT username, pid FROM pg_stat_activity WHERE username IS NOT NULL;
```

```
username | pid
-----+-----
u2       | 23194
u1       | 23195
```

```
postgres=> SELECT pg_terminate_backend(23194);
```

```
ERROR:  must be a member of the role whose process is being terminated
        or member of pg_signal_backend
```

```
postgres=> \c - postgres
```

```
You are now connected to database "postgres" as user "postgres".
```

```
postgres=# GRANT pg_signal_backend TO u1;
```

```
GRANT ROLE
```



```
postgres=# \c - u1
```

You are now connected to database "postgres" as user "u1".

```
postgres=> SELECT pg_terminate_backend(23194);
```

```
pg_terminate_backend
-----
t
```

```
postgres=> SELECT username, pid FROM pg_stat_activity WHERE username IS NOT NULL;
```

```
username | pid
-----+-----
u1       | 23212
```

Par contre, les connexions des superutilisateurs ne sont pas concernées.

Parmi ces rôles, `pg_read_all_stats` permet de lire les tables de statistiques d'activité. `pg_read_all_settings` permet de lire la configuration de tous les paramètres. `pg_stat_scan_tables` permet d'exécuter les procédures stockées de lecture des statistiques. `pg_monitor` est le rôle typique pour de la supervision : il combine les trois rôles précédents. Leur utilisation est identique à `pg_signal_backend`.

En version 15 arrive le rôle `pg_checkpoint`. Ce dernier permet à un rôle non `SUPERUSER` de lancer un `CHECKPOINT`.

À partir de la version 16 apparaît un rôle `pg_use_reserved_connections`, qui donne accès à un pool de connexions réservées défini par le paramètre `reserved connections` (0 par défaut). Cela peut servir à garantir un accès à la supervision ou à divers utilitaires sans en faire des superutilisateurs.

1.4.5 Droits plus globaux 2/2



- Rôles d'accès aux fichiers
 - `pg_read_server_files`
 - `pg_write_server_files`
 - `pg_execute_server_program`
- Rôles d'accès aux données (14+)
 - `pg_read_all_data`
 - `pg_write_all_data`
- Autres
 - `pg_create_subscription` (16+)

`pg_read_server_files` permet d'autoriser la lecture de fichiers auxquels le serveur peut accéder avec la commande SQL `COPY` et toutes autres fonctions d'accès de fichiers. `pg_write_server_files` permet la même chose en écriture. Cela sous-entend aussi les sauvegardes de fichiers côté serveur, par exemple avec `pg_basebackup` (à partir de la version 15) ou d'autres outils. `pg_execute_server_program` autorise les utilisateurs membres d'exécuter des programmes en tant que l'utilisateur qui exécute le serveur PostgreSQL au travers de la commande SQL `COPY` et de toute fonction permettant l'exécution d'un programme sur le serveur.

La version 14 ajoute trois nouveaux rôles. `pg_read_all_data` permet de lire toutes les données des tables, vues et séquences, alors que `pg_write_all_data` permet de les écrire. Quant à `pg_database_owner`, il permet de se comporter comme le propriétaire des bases de données.

En version 16 apparaît `pg_create_subscription` pour gérer des souscriptions en réplication logique.

1.4.6 Héritage des droits



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

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

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

Voici un exemple complet :

```
b1=# CREATE ROLE facturation;
CREATE ROLE
b1=# CREATE TABLE factures(id integer, dcreation date, libelle text,
montant numeric);
CREATE TABLE
b1=# GRANT ALL ON TABLE factures TO facturation;
```

GRANT

```
b1=# CREATE TABLE clients (id integer, nom text);
```

CREATE TABLE

```
b1=# GRANT ALL ON TABLE clients TO facturation;
```

GRANT

```
b1=# CREATE ROLE r1 LOGIN;
```

CREATE ROLE

```
b1=# GRANT facturation TO r1;
```

GRANT ROLE

```
b1=# \c b1 r1
```

You are now connected to database "b1" as user "r1".

```
b1=> SELECT * FROM factures;
```

```
 id | dcreation | libelle | montant  
----+-----+-----+-----  
(0 rows)
```

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

CREATE ROLE

```
b1=# \c b1 r2
```

You are now connected to database "b1" as user "r2".

```
b1=> SELECT * FROM factures;
```

```
ERROR: permission denied for relation factures
```

1.4.7 Changement de rôle



- Rôle par défaut
 - celui de la connexion
- Rôle emprunté :
 - après un `SET ROLE`
 - pour tout rôle dont il est membre

Par défaut, un utilisateur se connecte avec un rôle de connexion. Il obtient les droits et la configuration spécifique de ce rôle. Dans le cas où il hérite automatiquement des droits des rôles dont il est membre,

il obtient aussi ces droits qui s'ajoutent aux siens. Dans le cas où il n'hérite pas automatiquement de ces droits, il peut temporairement les obtenir en utilisant la commande `SET ROLE`. Il ne peut le faire qu'avec les rôles dont il est membre.

```
b1=# CREATE ROLE r31 LOGIN;
```

```
CREATE ROLE
```

```
b1=# CREATE ROLE r32 LOGIN NOINHERIT IN ROLE r31;
```

```
CREATE ROLE
```

```
b1=# \c b1 r31
```

```
You are now connected to database "b1" as user "r31".
```

```
b1=> CREATE TABLE t1(id integer);
```

```
CREATE TABLE
```

```
b1=> INSERT INTO t1 VALUES (1), (2);
```

```
INSERT 0 2
```

```
b1=> \c b1 r32
```

```
You are now connected to database "b1" as user "r32".
```

```
b1=> SELECT * FROM t1;
```

```
ERROR: permission denied for relation t1
```

```
b1=> SET ROLE TO r31;
```

```
SET
```

```
b1=> SELECT * FROM t1;
```

```
 id  
----  
  1  
  2
```

```
b1=> \c b1 postgres
```

```
You are now connected to database "b1" as user "postgres".
```

```
b1=# ALTER ROLE r32 INHERIT;
```

```
ALTER ROLE
```

```
b1=# \c b1 r32
```

```
You are now connected to database "b1" as user "r32".
```

```
b1=> SELECT * FROM t1;
```

```
 id  
----  
  1  
  2
```

```
b1=> \c b1 postgres
```

You are now connected to database "b1" as user "postgres".

```
b1=# REVOKE r31 FROM r32;
```

```
REVOKE ROLE
```

```
b1=# \c b1 r32
```

You are now connected to database "b1" as user "r32".

```
b1=> SELECT * FROM t1;
```

```
ERROR: permission denied for relation t1
```

```
b1=> SET ROLE TO r31;
```

```
ERROR: permission denied to set role "r31"
```

Le changement de rôle peut se faire uniquement au niveau de la transaction. Pour cela, il faut utiliser la clause `LOCAL`. Il peut se faire aussi sur la session, auquel cas il faut passer par la clause `SESSION`.

1.5 DROITS DE CONNEXION



- Lors d'une connexion, indication :
 - de l'hôte (socket Unix ou alias/adresse IP)
 - du nom de la base de données
 - du nom du rôle
 - du mot de passe (parfois optionnel)
- Selon les 3 premières informations
 - impose une méthode d'authentification

Lors d'une connexion, l'utilisateur fournit, explicitement ou non, plusieurs informations. PostgreSQL va choisir une méthode d'authentification en se basant sur les informations fournies et sur la configuration d'un fichier appelé `pg_hba.conf`.

1.5.1 Informations de connexion



Pour se connecter à une base, il faut :

- 4 informations
 - socket Unix ou adresse/alias IP
 - n° de port
 - nom de la base
 - nom du rôle
- Fournies explicitement
 - paramètres
 - environnement
- ou implicitement
 - environnement
 - défauts

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

- `-h` pour la socket Unix ou l'adresse/alias IP ;
- `-p` pour le numéro de port ;

- `-d` pour le nom de la base ;
- `-U` pour le nom du rôle.

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

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

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

- la socket Unix (`/var/run/postgresql` , parfois `/tmp`) en lieu d'un nom de machine ;
- le port 5432 ;
- la base `postgres` ou le nom de l'utilisateur PostgreSQL demandé, (suivant l'outil) ;
- le nom de l'utilisateur au niveau du système d'exploitation pour le nom du rôle.

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

1.5.2 Configuration de l'authentification : `pg_hba.conf`



PostgreSQL utilise les informations de connexion pour choisir la méthode de connexion

- Fichier de configuration : `pg_hba.conf`
- Exemple :

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

- Clauses `include` , `include_if_exists` , `include_dir` (v16)
- Contrôle avec la vue `pg_hba_file_rules`

Lorsque le serveur PostgreSQL récupère une demande de connexion, il connaît le type de connexion utilisé par le client (socket Unix, connexion TCP SSL, connexion TCP simple, etc.). Il connaît aussi l'adresse IP du client (dans le cas d'une connexion via une socket TCP), le nom de la base et celui de l'utilisateur.

PostgreSQL va donc chercher les lignes correspondantes dans le tableau enregistré dans le fichier `pg_hba.conf` (HBA est l'acronyme de *Host Based Authentication*). Ce fichier est présent à côté de `postgresql.conf` , donc par exemple dans `/var/lib/pgsql/16/data` (défaut du packaging

Red Hat) ou `/etc/postgresql/16/main` (défaut du packaging Debian). L'accès doit bien sûr en être le plus restreint possible.

Le contenu du fichier se présente en principe ainsi :

- 4 colonnes d'informations ;
- 1 colonne indiquant la méthode à appliquer ;
- 1 colonne optionnelle d'options.

Depuis PostgreSQL 16, il est possible d'indiquer ou plusieurs fichiers externes :

```
include pg_hba.ansible.conf
include "/chemin/fichier 2.conf"
```

(Existents aussi `include_if_exists`, qui ne tombe pas en erreur si le fichier n'existe pas, et `include_dir`, pour inclure tous les fichiers `.conf` d'un répertoire dans l'ordre de leurs noms.)

`pg_hba.conf` ne peut pas être modifié depuis PostgreSQL même, uniquement depuis le système d'exploitation. Après une modification, il faut dire explicitement à PostgreSQL de le recharger. Selon l'installation et l'OS, cet ordre peut être :

```
pg_ctl reload -D /mnt/base_pg/
systemctl reload postgresql-16
pg_ctlcluster 16 main reload
```

ou depuis PostgreSQL même :

```
SELECT pg_reload_conf() ;
```

Il y a une exception : sous Windows, le fichier est relu dès modification.



Si le fichier contient une erreur de syntaxe, il sera ignoré (et la ligne fautive apparaîtra dans les traces), sauf au (re)démarrage, où un fichier fautif bloque le démarrage !

Exemple de trace problématique :

```
...user=,db=,app=,client= LOG:  invalid connection type "hsot"
...user=,db=,app=,client= CONTEXT:  line 121 of configuration file
↪  "/etc/postgresql/16/main/pg_hba.conf"
...user=,db=,app=,client= FATAL:  could not load /etc/postgresql/16/main/pg_hba.conf
```

Depuis une session utilisateur, il est possible de consulter le fichier via la vue `pg_hba_file_rules`, tel qu'il est sur le disque, et tel qu'il serait compris par PostgreSQL après un rechargement. La colonne `error` est très pratique pour repérer immédiatement les erreurs de syntaxe ou de frappe. La colonne `file_name` indique le fichier source (à partir de PostgreSQL 16).

1.5.3 Exemple de pg_hba.conf



```
# TYPE DATABASE USER ADDRESS METHOD
# accès direct par la socket
local all all peer
# accès en local via réseau (localhost IPv6 et IPv4)
host all all ::1/128 scram-sha-256
host all all 127.0.0.0/24 scram-sha-256
# accès distants
hostssl erp all 192.168.0.0/16 scram-sha-256
hostssl logistique all 192.168.30.0/24 scram-sha-256
hostnssl test demo 192.168.74.150/32 scram-sha-256
# vieux client
hostssl compta durand 192.168.10.99/32 md5
# Connexions de réplication
local replication all peer
host replication all 192.168.74.5/32 scram-sha-256
host replication all 127.0.0.1/32 scram-sha-256
host replication all ::1/128 scram-sha-256
```

PostgreSQL lit le fichier dans l'ordre. La première ligne correspondant à la connexion demandée lui précise la méthode d'authentification à utiliser. Nous verrons que certaines méthodes comme `scram-sha-256` ou `md5` (les plus courantes) exigent un mot de passe, d'autres non, comme `ldap` ou `cert`. Il ne reste plus qu'à appliquer cette méthode. Si elle fonctionne, la connexion est autorisée et se poursuit.

Si elle ne fonctionne pas, quelle qu'en soit la raison, la connexion est refusée. Aucune autre ligne du fichier ne sera lue ! La raison du refus est tracée sur le serveur de manière beaucoup plus complète que pour le client :

```
2023-11-03 10:41:24 CET [1805700]: [2-1] user=attaquant,db=erp,
app=[unknown],client=::1 DETAIL: Role "mechant_hacker" does not exist.
Connection matched file "/etc/postgresql/16/defo/pg_hba.conf"
line 110: "host all all ::1/128 scram-sha-256"
```

Il est donc essentiel de bien configurer ce fichier pour avoir une protection maximale, et de faire très attention à l'ordre des entrées.

1.5.4 pg_hba.conf : colonne type



- 4 valeurs possibles

- local
- host
- hostssl (ssl=on prérequis)
- hostnossl

La colonne type peut contenir quatre valeurs différentes. La valeur `local` concerne les connexions via la socket Unix, donc depuis un compte sur le serveur même. Elle est généralement couplée à la méthode `peer`. En pratique, on s'en sert surtout pour les connexions administratives (de l'utilisateur système **postgres** au rôle **postgres**), pour les sauvegardes depuis le serveur même, ou pour une connexion depuis une application qui tourne sur la même machine.

Toutes les autres valeurs concernent les connexions via le réseau (socket TCP). La différence réside dans l'utilisation forcée ou non du SSL :

- `host` : SSL au choix du client ;
- `hostssl` : impérativement avec chiffage SSL ;
- `hostnossl` : impérativement sans SSL.

Il est à noter que l'option `hostssl` n'est utilisable que si le paramètre `ssl` du fichier `postgresql.conf` est à `on` (c'est le cas par défaut sous Debian, pas forcément ailleurs).

1.5.5 pg_hba.conf : colonne database



À quelle(s) base(s) autoriser la connexion ?

- mabase
- base1,base2,base3
- sameuser / samerole
- all
- @fichier.txt (fichier avec plusieurs bases)
- /db_[1-6] , /erp.* (regex, v16+)
- replication (pseudo-base pour réplication physique)

La seconde colonne de `pg_hba.conf` peut recueillir le nom d'une base, le nom de plusieurs bases en

les séparant par des virgules, le nom d'un fichier contenant la liste des bases ou quelques valeurs en dur. La valeur `all` indique toutes les bases.

Enfin, la valeur `sameuser` spécifie une base de données de même nom que le rôle demandé (l'utilisateur **durand** peut se connecter à la base **durand**), alors que la valeur `samerole` spécifie que le rôle demandé doit être membre du rôle portant le même nom que la base de données demandée (par exemple, si l'utilisateur **durand** est membre du groupe **compta**, il peut se connecter à la base **compta**).

Il est possible d'indiquer un fichier (préfixé de `@`) qui contiendra une liste de rôles. Il peut être à côté de `pg_hba.conf` mais on peut fournir un chemin complet.

Depuis PostgreSQL 16, il est possible d'utiliser des expressions régulières, qui doivent être précédées de `/` (voir exemple plus bas).

La valeur `replication` est utilisée pour définir les connexions de réplication physique (mais pas logique). Il n'est pas nécessaire d'avoir une base nommée **replication**, toutes les bases sont concernées par une réplication physique.

1.5.6 pg_hba.conf : colonne user



À quel(s) utilisateur(s) permettre la connexion ?

- `unrole`
- `role1,role2,role3`
- `all`
- `+groupe` (membres d'un rôle-groupe)
- `@fichier.txt` (fichier avec plusieurs utilisateurs)
- `/user_.*` (regex, v16+)

La troisième colonne peut recueillir :

- le nom d'un rôle, ou plusieurs rôles séparés par des virgules ;
- le nom d'un groupe (en le précédant d'un signe `+`) : tous les membres de ce groupe pourront se connecter avec leur rôle propre ;
- le nom d'un fichier (précédé de `@`) contenant une liste de rôles ;
- la valeur `all` qui indique tous les rôles ;
- une regex (précédée de `/`) pour indiquer plusieurs rôles (nouveau de la version 16).

Exemple :

#	TYPE	DATABASE	USER	ADDRESS	METHOD
	hostssl	erp_hr	u_virginie,u_sharon	10.1.17.67/32	scram-sha-256
	hostssl	all	+grp_admin	10.0.0.0/8	scram-sha-256

```
hostssl /erp_dev[1-5] @prestataires.txt 10.1.0.0/16 scram-sha-256
hostssl /erp.* /u.* 10.0.0.0/8 scram-sha-256
```

Ici, ne peuvent se connecter à la base **erp_hr** que deux utilisatrices nommées, et uniquement depuis une IP précise. Les membres du groupe **grp_admin** peuvent se connecter à toutes les bases depuis tout le réseau privé. Aux bases **erp_dev1** à **erp_dev5** peuvent se connecter les prestataires dont les rôles sont dans le fichier `prestataires.txt`, depuis un certain sous-réseau. Et tous les utilisateurs dont le rôle commence par **u_** peuvent se connecter aux bases dont le nom commence par **erp_**.

1.5.7 pg_hba.conf : colonne adresse IP



- Uniquement pour `host` / `hostssl` / `hostnossl`
- Adresse IPv4 ou IPv6 : format CIDR ou avec masque
 - `192.168.1.0/24`
 - `192.168.1.1/32`
 - `192.168.1.0 255.255.255.0`
- Nom d'hôte possible
 - mais coût recherche DNS

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

```
192.168.168.1 255.255.255.255
192.168.168.0 255.255.255.0
```

Et voici quelques exemples d'adresses IP (ou plages d'adresses) au format CIDR :

```
192.168.0.0/8      # réseau 192.168.x.y
192.168.66.0/24   # réseau 192.168.66.x
192.168.168.1/32  # 1 IP
0.0.0.0/0         # toutes les IPv4
::0/0            # toutes les IPv6
::1/128          # localhost en IPv6
fe80::7a31:c1ff:0000:0000/96 # un réseau en IPv6
```

Dans le doute, la vue `pg_hba_file_rules` indique le masque calculé.

Les alias `samehost` et `samenet` désignent le serveur (avec toutes ses adresses) et tous les réseaux auxquels il appartient.

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



Rappelons que le paramètre `listen_addresses` dans `postgresql.conf` restreint les IP sur lesquelles que PostgreSQL écoute. En général on le modifie dès l'installation.

1.5.8 pg_hba.conf : colonne méthode



Quelle méthode d'authentification utiliser ?

- Interne / externe
 - et options dans la dernière colonne
- Le client peut en exiger une (v16)

La dernière colonne est généralement la méthode d'authentification, mais certaines acceptent des options dans une colonne supplémentaire. Nous verrons plus bas les différentes méthodes d'authentification, internes ou externes.

Le serveur a toujours le dernier mot pour imposer une méthode d'authentification, mais pour des raisons de sécurité le client peut aussi exiger celle qu'il attend (depuis PostgreSQL 16). Dans l'exemple suivant, le client exige la méthode `scram-sha-256`, mais le serveur est encore configuré en `md5` et la connexion échoue.

```
psql 'host=s1 dbname=postgres user=postgres require_auth=scram-sha-256'
psql: error: connection to server at "s1" (192.168.74.65), port 54325 failed:
authentication method requirement "scram-sha-256" failed:
server requested a hashed password
```

D'autres variantes pour `require_auth` sont :

- `require_auth=scram-sha-256,md5`
- `require_auth=!password`
- `require_auth=cert`

1.5.9 pg_hba.conf : colonne options



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

Les options disponibles dépendent de la méthode d'authentification sélectionnée. Cependant, toutes les méthodes externes permettent l'utilisation de l'option `map`. Cette option a notamment pour but d'indiquer la carte de correspondance à sélectionner dans le fichier `pg_ident.conf` (voir plus bas).

1.5.10 pg_hba.conf : méthodes internes



- `trust` : dangereux !
- `reject` : pour interdire
- `password` : en clair, à éviter !
- `md5` : déconseillée
- `scram-sha-256`

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

La méthode `password` force la saisie d'un mot de passe. Cependant, ce dernier est envoyé en clair sur le réseau. Il n'est donc pas conseillé d'utiliser cette méthode, surtout sur un réseau non sécurisé. (Noter que PostgreSQL chiffrera toujours le mot de passe chiffré avec une des méthodes ci-dessous.)

La méthode `md5` est encore très utilisée mais obsolète. La saisie du mot de passe est forcée. Le mot de passe transite chiffré en `md5`. Cette méthode souffre néanmoins de certaines faiblesses décrites dans la section **Mot de passe**, on préférera la suivante.

La méthode `scram-sha-256` est la plus sécurisée, elle offre moins d'angles d'attaque que `md5`. Elle est à privilégier.

La méthode `reject` sert pour interdire définitivement un accès à un utilisateur, un réseau ou une base quelles que soient les lignes suivantes de `pg_hba.conf`. Par exemple, on veut que le rôle **u1** puisse se connecter à la base de données `b1` mais pas aux autres. Voici un moyen de le faire (pour une connexion via les sockets Unix) :

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

1.5.11 pg_hba.conf : méthodes externes



- Système externe : `ldap`, `radius`
- Kerberos : `gss`, `sspi`
- SSL : `cert`
- OS : `peer`, `pam`, `ident`, `bsd`

Certaines méthodes permettent d'utiliser des annuaires d'entreprise comme RADIUS, LDAP ou ActiveDirectory. Certaines méthodes sont spécifiques à un système d'exploitation.

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

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

La méthode `gss` (GSSAPI) correspond au protocole du standard de l'industrie pour l'authentification sécurisée définie dans RFC 2743. PostgreSQL supporte GSSAPI avec l'authentification Kerberos suivant la RFC 1964 ce qui permet de faire du *Single Sign-On*. C'est la méthode à utiliser avec Active Directory. `sspi` (uniquement dans le monde Windows) permet d'utiliser NTLM faute d'Active Directory.

La méthode `cert` utilise un certificat SSL sur le client¹¹, sans mot de passe.

La méthode `ident` permet d'associer les noms des utilisateurs du système d'exploitation aux noms des utilisateurs du système de gestion de bases de données. Un démon fournissant le service `ident` est nécessaire. (Les paquets RPM la mettaient en place jusque PostgreSQL 12 inclus pour une connexion depuis `localhost`, mais `peer` est maintenant recommandé.)

La méthode `peer` permet d'associer les noms des utilisateurs du système d'exploitation aux noms des utilisateurs du système de gestion de bases de données. Ceci n'est possible qu'avec une connexion locale. Par défaut, les paquets d'installation l'utilisent pour autoriser l'utilisateur système **postgres** à se connecter à l'instance locale, sans passer par le réseau, en tant que **postgres** sans mot de passe.

La méthode `pam` authentifie l'utilisateur en passant par les *Pluggable Authentication Modules* (PAM) fournis par Linux et d'autres Unix. De manière similaire, la méthode `bsd` utilise le système d'authentification BSD sur OpenBSD.

1.5.12 Un (mauvais) exemple de `pg_hba.conf`



Un exemple:

TYPE	DATABASE	USER	ADDRESS	METHOD
local	all	postgres		peer
local	web	web		md5
local	sameuser	all		peer
host	all	postgres	127.0.0.1/32	peer
host	all	all	127.0.0.1/32	md5
host	all	all	89.192.0.3/8	md5
hostssl	recherche	recherche	89.192.0.4/32	scram-sha-256

à ne pas suivre...

Ce fichier comporte plusieurs erreurs. Tout d'abord,

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

¹¹<https://docs.postgresql.fr/current/auth-cert.html>

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

```
local sameuser all peer
```

qui, sera appliqué d'abord et restreint à la base de même nom.

Dans cette règle :

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

le masque CIDR `/8` est incorrect, et donc, au lieu d'autoriser seulement 89.192.0.3 à se connecter, on autorise tout le réseau 89.*.

L'entrée :

```
hostssl recherche recherche 89.192.0.4/32 scram-sha-256
```

est bonne, mais inutile, car masquée par la ligne précédente : toute ligne correspondant à cette entrée correspondra aussi à la ligne précédente. Le fichier étant lu séquentiellement, cette dernière entrée ne sert à rien.

Enfin, il aurait mieux valu que tous les mots de passe utilisent le chiffrement `scram-sha-256` et non `md5`.

1.5.13 Mapping : pg_ident.conf



- Correspondance entre :
 - utilisateurs authentifiés (serveur, LDAP)
 - rôle de la base

- Fichier `pg_ident.conf` :

#	MAPNAME	SYSTEM-USERNAME	PG-USERNAME
	pgadmins	joe	postgres
	pgadmins	zoe	postgres

- Prise en compte des modifications : rechargement
- Includes possibles
- Contrôle : `pg_ident_file_mapping` (v15+)

En utilisant une méthode d'authentification externe, locale ou à distance, le nom de l'utilisateur authentifié ne correspond pas forcément au nom du rôle avec lequel on souhaite se connecter. On peut le préciser avec `-U` mais il y a plus simple et sécurisé.

Par exemple, deux DBA ont les comptes **joe** et **zoe** sur le système d'exploitation. Ils souhaitent pouvoir se connecter en tant que rôle **postgres** sans saisie de mot de passe depuis leur compte, sans devoir faire un `sudo -iu postgres` pour lesquels on ne va pas forcément leur donner les droits.

Ceci n'est pas possible avec la configuration par défaut :

```
joe$ psql -U postgres
psql: error: FATAL: Peer authentication failed for user "postgres"
```

Mais cela devient possible en configurant correctement les fichiers `pg_hba.conf` et `pg_ident.conf`. Il est tout d'abord nécessaire d'indiquer trois informations dans le fichier `pg_ident.conf`, une « map », les utilisateurs authentifiés (ici **joe** et **zoe**) et le rôle de connexion (ici **postgres**) :

```
# MAPNAME          SYSTEM-USERNAME    PG-USERNAME
pgadmins           joe                postgres
pgadmins           zoe                postgres
```

À partir de la version 15, il est possible de vérifier que la configuration de ce fichier est bonne. Pour cela, il faut lire la vue `pg_ident_file_mappings` (et vérifier que le champ `error` est vide).

```
SELECT * FROM pg_ident_file_mappings;
```

line_number	map_name	sys_name	pg_username	error
43	pgadmins	joe	postgres	
44	pgadmins	zoe	postgres	

Il faut ensuite ajouter cette *map* à la ligne de configuration d'authentification en haut du fichier `pg_hba.conf` :

```
local all all peer map=pgadmins
```

Après rechargement de la conf, la connexion de **joe** ou **zoe** à **postgres** en local se fera. Noter que la sécurité de l'accès à PostgreSQL repose à présent sur la sécurisation des utilisateurs système **joe** et **zoe** !

Tout ceci n'est valable que pour les connexions depuis le système d'exploitation même. En pratique, `pg_ident.conf` est utilisé pour se connecter directement en tant que **postgres** depuis un compte utilisateur sur le serveur, ou avec une authentification LDAP.

Depuis PostgreSQL 16, `pg_ident.conf` supporte les mêmes directives d'inclusion que `pg_hba.conf` (`include`, `include_if_exists`, `include_dir`), et le fichier d'origine d'une ligne est visible dans `pg_ident_file_mappings`.

1.6 SYNCHRONISATION DES RÔLES



- PostgreSQL externalise l'authentification uniquement
 - Le rôle, ses paramètres, ses héritages sont toujours dans PostgreSQL
- ldap2pg synchronise les rôles depuis un annuaire LDAP
 - un rôle PostgreSQL absent de l'annuaire est supprimé
 - sait synchroniser les privilèges
 - configuration avancée en YAML
 - <https://labs.dalibo.com/ldap2pg>

PostgreSQL sait externaliser la vérification du mot de passe ou authentifier depuis une autre méthode qu'un mot de passe. Néanmoins, PostgreSQL n'externalise jamais la configuration d'un rôle.

Au delà de son nom et de son mot de passe, un rôle possède des options (`SUPERUSER` , `LOGIN` , etc.), des héritages, des privilèges, des privilèges par défaut pour chaque base voire pour chaque schéma, des paramètres globaux ou par base. Tout cela reste dans PostgreSQL, même si l'authentification est externe.

L'outil ldap2pg permet de centraliser les rôles dans l'annuaire d'entreprise. ldap2pg est un binaire (programmé en Go) à installer à côté de PostgreSQL ou ailleurs.

ldap2pg traite une configuration avancée au format YAML décrivant toutes les recherches LDAP énumérant les rôles à provisionner dans l'annuaire. Par exemple : la liste des DBA, la liste des développeurs d'une application, etc.

À chaque recherche LDAP est associé un modèle de rôle où sont substitués des attributs LDAP fournis par l'annuaire. ldap2pg sait également configurer les paramètres propres à chaque rôle, les privilèges par défaut, et les privilèges sur les objets.

La recherche LDAP des rôles d'une instance est indépendante de l'authentification. On peut très bien rechercher dans l'annuaire la liste des DBA avec le protocole LDAP et authentifier les connexions avec Kerberos.

Page du projet : <https://labs.dalibo.com/ldap2pg>

1.7 TÂCHES DE MAINTENANCE



- Trois opérations essentielles
 - `VACUUM`
 - `ANALYZE`
 - `REINDEX`
- En arrière-plan : démon autovacuum (pour les deux premiers)
- Optionnellement : automatisable par cron
- Manuellement : `VACUUM ANALYZE table` (batchs, gros imports...)

PostgreSQL demande peu de maintenance par rapport à d'autres SGBD. Néanmoins, un suivi vigilant de ces tâches participera beaucoup à conserver un système performant et agréable à utiliser.

La maintenance d'un serveur PostgreSQL revient à s'occuper de trois opérations :

- le `VACUUM`, pour éviter une fragmentation trop importante des tables ;
- l' `ANALYZE`, pour mettre à jour les statistiques sur les données contenues dans les tables ;
- le `REINDEX`, pour reconstruire les index.

Il s'agit donc de maintenir, voire d'améliorer, les performances du système. Il ne s'agit en aucun cas de s'assurer de la stabilité du système.

Généralement on se repose sur le processus d'arrière-plan **autovacuum**, qui s'occupe des `VACUUM` et `ANALYZE` (mais pas `REINDEX`) en fonction de l'activité, et prend soin de ne pas la gêner. Il est possible de planifier des exécutions régulières avec cron (ou tout autre ordonnanceur), notamment pour des `REINDEX`.

Un appel explicite est parfois nécessaire, notamment au sein de batchs ou de gros imports... L'autovacuum n'a pas forcément eu le temps de passer entre deux étapes, et les statistiques ne sont alors pas à jour : le planificateur pense que les tables sont encore vides et peut choisir un plan désastreux. On lancera donc systématiquement au moins un `ANALYZE` sur les tables modifiées après les modifications lourdes. Un `VACUUM ANALYZE` est parfois encore plus intéressant, notamment si des données ont été modifiées ou effacées, ou si les requêtes suivantes peuvent profiter d'un parcours d'index seul (`Index Only Scan`).

1.7.1 Maintenance : VACUUM



- `VACUUM nomtable ;`
 - cartographie les espaces libres pour une réutilisation (& autre maintenance)
 - utilisable en parallèle avec les autres opérations
 - et même automatisé
 - vue `pg_stat_progress_vacuum`
- `vacuumdb --echo`

PostgreSQL ne supprime pas des tables les versions périmées des lignes après un `UPDATE` ou un `DELETE`, elles deviennent juste invisibles. La commande `VACUUM` permet de récupérer l'espace utilisé par ces lignes afin d'éviter un accroissement continu du volume occupé sur le disque.

Une table qui subit beaucoup de mises à jour et suppressions nécessitera des nettoyages plus fréquents que les tables rarement modifiées. Le `VACUUM` « simple » (`VACUUM nomdematable ;`) marque les données expirées dans les tables et les index pour une utilisation future. Il ne tente pas de rendre au système de fichiers l'espace utilisé par les données obsolètes, sauf si l'espace est à la fin de la table et qu'un verrou exclusif de table peut être facilement obtenu. L'espace inutilisé au début ou au milieu du fichier ne provoque pas un raccourcissement du fichier et ne redonne pas d'espace mémoire au système d'exploitation. De même, l'espace d'une colonne supprimée n'est pas rendu.

Cet espace libéré n'est pas perdu : il sera disponible pour les prochaines lignes insérées et mises à jour, et la table n'aura pas besoin de grandir.

Un `VACUUM` peut être lancé sans aucune gêne pour les utilisateurs. Il va juste générer des écritures supplémentaires. On verra plus loin que l'autovacuum s'occupe de tout cela en tâche de fond et de manière non intrusive, mais il arrive encore que l'on lance un `VACUUM` manuellement. Noter qu'un `VACUUM` s'occupe également de quelques autres opérations de maintenance qui ne seront pas détaillées ici.

L'option `VERBOSE` vous permet de suivre ce qui a été fait. Dans l'exemple suivant, 100 000 lignes sont nettoyées dans 541 blocs, mais 300 316 lignes ne peuvent être supprimées car une autre transaction reste susceptible de les voir.

```
VACUUM VERBOSE livraisons ;
```

```
INFO: vacuuming "public.livraisons"
INFO: "livraisons": removed 100000 row versions in 541 pages
INFO: "livraisons": found 100000 removable, 300316 nonremovable
                                row versions in 2165 out of 5406 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 6249883
There were 174 unused item pointers.
Skipped 0 pages due to buffer pins, 540 frozen pages.
0 pages are entirely empty.
CPU: user: 0.04 s, system: 0.00 s, elapsed: 0.08 s.
```

VACUUM

Temps : 88,990 ms

L'ordre `VACUUM` possède de nombreux paramètres que nous ne décrivons pas ici car ils servent peu au quotidien (au besoin, voir le module M5¹²).

Supervision :

La vue `pg_stat_progress_vacuum` permet de suivre un `VACUUM` simple pendant son déroulement.

La vue `pg_stat_user_tables` contient pour chaque table la date du dernier passage d'un `VACUUM` simple (champ `last_vacuum`) celle du dernier passage automatique (`last_autovacuum`). Pour les passages précédents, il faudra se rabattre sur les traces (on conseille de positionner `log_autovacuum_min_duration` suffisamment bas, ou à 0). Il est important de vérifier que les tables actives sont régulièrement nettoyées.

Outil en ligne de commande :

L'outil `vacuumdb` permet d'exécuter depuis le shell un `VACUUM` sur une ou toutes les bases. C'est l'outil idéal lorsque l'on planifie un nettoyage quotidien ou hebdomadaire.

Il sait exécuter des `VACUUM` sur plusieurs tables en parallèle, ou profiter des optimisations liées aux nettoyages en masse apparues en v16.

```
$ vacuumdb --echo --all
```

```
SELECT pg_catalog.set_config('search_path', '', false);
vacuumdb : exécution de VACUUM sur la base de données « pgbench »
RESET search_path;
SELECT c.relname, ns.nspname FROM pg_catalog.pg_class c
  JOIN pg_catalog.pg_namespace ns ON c.relnamespace OPERATOR(pg_catalog.=) ns.oid
  LEFT JOIN pg_catalog.pg_class t ON c.reltoastrelid OPERATOR(pg_catalog.=) t.oid
 WHERE c.relkind OPERATOR(pg_catalog.=) ANY (array['r', 'm'])
  ORDER BY c.relpages DESC;
SELECT pg_catalog.set_config('search_path', '', false);
VACUUM (SKIP_DATABASE_STATS) public.pgbench_accounts;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_proc;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_attribute;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_description;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_statistic;
...
...
VACUUM (ONLY_DATABASE_STATS);
SELECT pg_catalog.set_config('search_path', '', false);
vacuumdb : exécution de VACUUM sur la base de données « postgres »
...
...
VACUUM (ONLY_DATABASE_STATS);
```



L'ordre `VACUUM` connaît beaucoup de variantes. Surtout, ne confondez pas la version simple ci-dessus avec sa variante `VACUUM FULL` ci-dessous.

¹²https://dali.bo/m5_html

1.7.2 Maintenance : VACUUM FULL



- `VACUUM FULL nomtable ;`
 - défragmente la table
 - réécriture (place nécessaire !)
 - verrou exclusif (ni lecture ni écriture !)
 - réindexation
 - utilisation exceptionnelle
 - vue `pg_stat_progress_cluster` (v12)

Un `VACUUM` simple fait rarement gagner de l'espace disque. Il faut utiliser l'option `FULL` pour ça : la commande `VACUUM FULL nomtable ;` réécrit la table en ne gardant que les données actives, et au final libère donc l'espace consommé par les lignes périmées ou les colonnes supprimées, et le rend au système d'exploitation. Les index sont réécrits au passage.

Inconvénient principal : `VACUUM FULL` acquiert un verrou exclusif sur chaque table concernée : personne ne peut plus y écrire ni même lire avant la fin de l'opération, et les sessions accédant aux tables sont mises en attente. Cela peut être long pour de grosses tables. D'autre part, le `VACUUM FULL` peut lui-même attendre la libération d'un verrou, tout en bloquant les transactions suivantes (phénomène d'empilement des verrous). Il est conseillé d'opérer par exemple ainsi :

```
SET lock_timeout TO '3s';  
VACUUM (FULL, VERBOSE) nomtable ;
```

Ainsi l'ordre `VACUUM FULL` sera annulé s'il n'obtient pas son verrou assez vite.

Autre inconvénient : `VACUUM FULL` écrit la nouvelle version de la table à côté de l'ancienne version avant d'effacer cette dernière : l'espace occupé peut donc temporairement doubler. Si vos disques sont presque pleins, vous ne pourrez donc pas faire un `VACUUM FULL` d'une grosse table pour récupérer de l'espace !

L'autovacuum ne procédera jamais à un `VACUUM FULL`, vous devrez toujours le demander explicitement. On le réservera aux périodes de maintenance, dans les cas où il est vraiment nécessaire.

En effet, il ne sert à rien de chercher à réduire au strict minimum la taille des tables par des `VACUUM FULL` répétés. Dans une base active, les espaces libres sont vite réutilisés par de nouvelles données. Le *bloat* (l'espace inutilisé d'une table) se stabilise généralement dans une proportion dépendant des débits d'insertions, suppressions et modifications dans la table.

Supervision :

La vue `pg_stat_progress_cluster` permet de suivre un `VACUUM FULL` (à partir de PostgreSQL 12). Son nom provient de la proximité avec la commande `CLUSTER` (voir plus bas).

Les `VACUUM FULL` ne sont pas tracés dans la vue `pg_stat_user_tables`.

Outil en ligne de commande :

L'outil `vacuumdb` possède une option `--full`.

1.7.3 VACUUM vs VACUUM FULL



- `VACUUM`
 - maintenance quotidienne
 - entre étapes d'un batch
 - l'autovacuum suffit généralement
- `VACUUM FULL`
 - après de grosses modifications
 - exceptionnel

Quand faut-il utiliser `VACUUM` sur une table ?

- pour des nettoyages réguliers ;
- entre les étapes d'un batch ;
- si vous constatez que l'autovacuum ne passe pas assez souvent et qu'un changement de paramétrage ne suffit pas ;
- et ce, pendant que votre base tourne.

Quand faut-il utiliser `VACUUM FULL` sur une table ?

- après des suppressions massives de données ;
- si le verrou exclusif ne gêne pas la production ;
- dans le cadre d'une maintenance exceptionnelle.

Des `VACUUM` standards et une fréquence modérée sont une meilleure approche que des `VACUUM FULL`, même non fréquents, pour maintenir des tables mises à jour fréquemment : faites confiance à l'autovacuum jusque preuve du contraire.

`VACUUM FULL` est recommandé dans les cas où vous savez que vous avez supprimé ou modifié une grande partie des lignes d'une table, et que les espaces libres ne seront pas à nouveau remplis assez vite, de façon à ce que la taille de la table soit réduite de façon conséquente.

Les deux outils peuvent se lancer à la suite. Après un `VACUUM FULL` (bloquant) sur une table, on lance souvent immédiatement un `VACUUM ANALYZE`. Cela semble inutile du point de vue des données, mais les autres opérations de maintenance impliquées peuvent améliorer les performances.

1.7.4 Maintenance : ANALYZE



- Met à jour les statistiques sur les données pour l'optimiseur de requêtes
- Géré par l'autovacuum
 - Parfois manuel : batch, `ALTER TABLE`, tables temporaires...
- Échantillonnage :
 - `default_statistics_target` (défaut 100)
 - `ALTER TABLE ma_table ALTER ma_colonne SET STATISTICS 500;`
 - Attention au temps de planification !
- Progression avec `pg_stat_progress_analyze` (v13)

L'optimiseur de requêtes de PostgreSQL s'appuie sur des informations statistiques calculées à partir des données des tables. Ces statistiques sont récupérées par la commande `ANALYZE`, qui peut être invoquée seule ou comme une option de `VACUUM`. Il est important d'avoir des statistiques relativement à jour sans quoi des mauvais choix dans les plans d'exécution pourraient pénaliser les performances de la base.

L'autovacuum de PostgreSQL appelle au besoin `ANALYZE` si l'activité de la table le nécessite. C'est généralement suffisant, même s'il est fréquent de modifier le paramétrage sur de grosses tables.

Il est possible de programmer `ANALYZE` périodiquement (le dimanche, la nuit par exemple, à l'aide d'une commande `cron` par exemple), éventuellement couplé à un `VACUUM` :

```
VACUUM ANALYZE nomdematable ;
```

Il existe des cas où lancer un `ANALYZE` manuellement est nécessaire :

- en mode « batch » : l'autovacuum n'a pas forcément le temps de passer entre deux étapes, on peut être amené à intercaler un `VACUUM ANALYZE` sur des tables modifiées ;
- quand certains plans de requêtes affichent des statistiques aberrantes : la mise à jour des statistiques peut suffire (et l'on regardera ensuite dans `pg_stat_user_tables.last_autoanalyze` si l'autovacuum a tardé et s'il y a un ajustement à faire ce côté) ;
- après un `ALTER TABLE [...] ALTER COLUMN`, car les statistiques de la colonne peuvent disparaître, ou bien sûr lors de l'ajout d'une colonne pré-remplie ;
- lors de l'ajout d'un index fonctionnel : l'`ANALYZE` mène à la création d'une nouvelle entrée dans `pg_statistics` ;

- lors de l'utilisation des tables temporaires : l'autovacuum ne les voit pas.

Le paramètre `default_statistics_target` définit l'échantillonnage par défaut des statistiques pour les colonnes de chacune des tables. La valeur par défaut est de 100. Ainsi, pour chaque colonne, 30 000 lignes sont choisies au hasard, et les 100 valeurs les plus fréquentes et un histogramme à 100 bornes sont stockés dans `pg_statistics` en guise d'échantillon représentatif des données.

Des valeurs supérieures provoquent un ralentissement important d'`ANALYZE`, un accroissement de la table `pg_statistics`, et un temps de calcul des plans d'exécution plus long. On conserve généralement la valeur `100` par défaut (sauf peut-être sur certaines grosses bases aux requêtes complexes et longues, comme des entrepôts de données).

Voici la commande à utiliser si l'on veut modifier cette valeur pour une colonne précise, la valeur ainsi spécifiée prévalant sur la valeur de `default_statistics_target` :

```
ALTER TABLE ma_table ALTER ma_colonne SET STATISTICS 200 ;
```

```
ANALYZE ma_table ;
```

Sans l'`ANALYZE` explicite, la mise à jour attendrait le prochain passage de l'autovacuum.

La vue `pg_stat_user_tables` contient aussi les dates du dernier passage d'un `ANALYZE` manuel (champ `last_analyze`) ou automatique (`last_autoanalyze`). Là encore, vérifier que les tables actives sont régulièrement analysées.

La version 13 apporte une vue appelée `pg_stat_progress_analyze` qui permet de suivre l'exécution des `ANALYZE` en cours.

1.7.5 Maintenance : REINDEX



- Lancer `REINDEX` régulièrement permet
 - de gagner de l'espace disque
 - d'améliorer les performances
 - de réparer un index corrompu/invalidé
- `VACUUM` ne provoque pas de réindexation
- `VACUUM FULL` réindexe
- Clause `CONCURRENTLY` (v12+)
- Clause `TABLESPACE` (v14+)

`REINDEX` reconstruit un index en utilisant les données stockées dans la table, remplaçant l'ancienne copie de l'index. La même commande peut réindexer tous les index d'une table :

```
REINDEX INDEX nomindex ;  
REINDEX (VERBOSE) TABLE nomtable ;
```

Les pages d'index qui sont devenues complètement vides sont récupérées pour être réutilisées. Il existe toujours la possibilité d'une utilisation inefficace de l'espace : même s'il ne reste qu'une clé d'index dans une page, la page reste allouée. La possibilité d'inflation n'est pas indéfinie, mais il est souvent utile de planifier une réindexation périodique pour les index fréquemment modifiés.

De plus, pour les index B-tree, un index tout juste construit est plus rapide qu'un index qui a été mis à jour plusieurs fois. En effet, dans un index nouvellement créé, les pages logiquement adjacentes sont aussi physiquement adjacentes.

La réindexation est aussi utile dans le cas d'un index corrompu. Ce cas est heureusement très rare, et souvent lié à des problèmes matériels.

Les index « invalides » sont inutilisables et ignorés, et doivent également être reconstruits. Ce statut apparaît en bas de la description de la table associée :

```
# \d+ pgbench_accounts  
...  
Index :  
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid) INVALID
```

Un index peut devenir invalide pour deux raisons. La première ne concerne plus les versions supportées : avant PostgreSQL 10, des index de type hash (uniquement) pouvaient devenir invalides après un redémarrage brutal, car ils n'étaient alors pas journalisés. La seconde raison est une conséquence de la clause `CONCURRENTLY` des ordres `CREATE INDEX` et `REINDEX`. Cette clause permet de créer/réindexer un index sans bloquer les écritures dans la table. Cependant, si, au bout de deux passes, l'index n'est toujours pas complet, il est considéré comme invalide, et doit être soit détruit, soit reconstruit avec la commande `REINDEX`.

Noter que, sans `CONCURRENTLY`, un `REINDEX` bloque non seulement les écritures, mais aussi souvent les lectures. On préférera donc le `CONCURRENTLY` si la table est utilisée :

```
REINDEX (VERBOSE) INDEX nomindex CONCURRENTLY ;
```

Enfin, depuis la version 14, il est possible de réindexer un index tout en le changeant de tablespace. Pour cela, il faut utiliser la clause `TABLESPACE` avec en argument le nom du tablespace de destination.

Il est à savoir que l'opération `VACUUM` (sans `FULL`) ne provoque pas de réindexation. Une réindexation est effectuée lors d'un `VACUUM FULL`.

La commande système `reindexdb` peut être utilisée pour réindexer une table, une base ou une instance entière.

1.7.6 Maintenance : CLUSTER



- `CLUSTER`
 - alternative à `VACUUM FULL`
 - tri des données de la table suivant un index
- Attention, `CLUSTER` nécessite près du double de l'espace disque utilisé pour stocker la table et ses index
- Progression avec `pg_stat_progress_cluster`

La commande `CLUSTER` provoque une réorganisation des données de la table en triant les lignes suivant l'ordre indiqué par l'index. Du fait de la réorganisation, le résultat obtenu est équivalent à un `VACUUM FULL` dans le contexte de la fragmentation. Elle verrouille tout aussi complètement la table et nécessite autant de place.

Attention, cette réorganisation est ponctuelle, et les données modifiées ou insérées par la suite n'en tiennent généralement pas compte. L'opération peut donc être à refaire après un certain temps.

Comme après un `VACUUM FULL`, lancer un `VACUUM ANALYZE` manuellement peut être bénéfique pour les performances.

En ligne de commande, l'outil associé `clusterdb` permet de lancer la réorganisation de tables ayant déjà fait l'objet d'une « clusterisation ».

La vue `pg_stat_progress_cluster` permet de suivre le déroulement du `CLUSTER`.

1.7.7 Maintenance : automatisation



- Automatisation des tâches de maintenance
- Cron sous Unix
- Tâches planifiées sous Windows

L'exécution des commandes `VACUUM`, `ANALYZE` et `REINDEX` peut se faire manuellement dans certains cas. Il est cependant préférable de mettre en place une exécution automatique de ces commandes. La plupart des administrateurs utilise cron sous Unix et les tâches planifiées sous Windows. pgAgent peut aussi être d'une aide précieuse pour la mise en place de ces opérations automatiques.

Peu importe l'outil. L'essentiel est que ces opérations soient réalisées et que le statut de leur exécution soit vérifié périodiquement.

La fréquence d'exécution dépend principalement de la fréquence des modifications et suppressions pour le `VACUUM` et de la fréquence des insertions, modifications et suppressions pour l'`ANALYZE`.

1.7.8 Maintenance : autovacuum



- Automatisation par cron
 - simple, voire simpliste
- Processus autovacuum
 - `VACUUM` / `ANALYZE` si nécessaire
 - Nombreux paramètres
 - Nécessite la récupération des statistiques d'activité

L'automatisation du vacuum par cron est simple à mettre en place. Cependant, elle s'exécute pour toutes les tables, sans distinction. Que la table ait été modifiée plusieurs millions de fois ou pas du tout, elle sera traitée par le script. À l'inverse, l'autovacuum est un outil qui vérifie l'état des tables et, suivant le dépassement d'une limite, déclenche ou non l'exécution d'un `VACUUM` ou d'un `ANALYZE`, voire des deux.

L'autovacuum est activé par défaut, et il est conseillé de le laisser ainsi. Son paramétrage permet d'aller assez loin si nécessaire selon la taille et l'activité des tables.

1.7.9 Maintenance : Script de REINDEX



- Automatisation par cron
- Recherche des index fragmentés
- Si clé primaire ou contrainte unique :
 - `REINDEX`
- Sinon :
 - `CREATE INDEX CONCURRENTLY`
- Exemple

Voici un script créé pour un client dans le but d'automatiser la réindexation uniquement pour les index le méritant. Pour cela, il vérifie les index fragmentés avec la fonction `pgstatindex()` de

l'extension `pgstattuple` (installable avec un simple `CREATE EXTENSION pgstattuple ;` dans chaque base).

Au-delà de 30 % de fragmentation (par défaut), l'index est réindexé. Pour minimiser le risque de blocage, le script utilise `CREATE INDEX CONCURRENTLY` en priorité, et `REINDEX` dans les autres cas (clés primaires et contraintes uniques).

La version 12 permet d'utiliser l'option `CONCURRENTLY` avec `REINDEX`. Ce script pourrait l'utiliser après avoir détecté qu'il se trouve sur une version compatible.

```
#!/bin/bash
# Script de réindexation d'une base
# ce script va récupérer la liste des index disponibles sur la base
# et réindexer l'index s'il est trop fragmenté ou invalide

# Mode debug
#set -x

# Récupération de la base maintenance
if test -z "$PGDATABASE"; then
    export PGDATABASE=postgres
fi

# quelques constantes personnalisables
TAUX_FRAGMENTATION_MAX=30
NOM_INDEX_TEMPORAIRE=index_traitement_en_cours
NB_TESTS=3
BASES=""

# Quelques requêtes
REQ_LISTEBASES="SELECT array_to_string(array(
    SELECT datname
    FROM pg_database
    WHERE datallowconn AND datname NOT IN ('postgres', 'template1')), ' ')"
REQ_LISTEINDEX="
SELECT n.nspname as \"Schéma\", tc.relname as \"Table\", ic.relname as \"Index\",
    i.indexrelid as \"IndexOid\",
    i.indisprimary OR i.indisunique as \"Contrainte\", i.indisvalid as \"Valide?\",
    round(100-(pgstatindex(n.nspname||'.'||ic.relname)).avg_leaf_density)
    as \"Fragmentation\",
    pg_get_indexdef(i.indexrelid) as \"IndexDef\"
FROM pg_index i
JOIN pg_class ic ON i.indexrelid=ic.oid
JOIN pg_class tc ON i.indexrelid=tc.oid
JOIN pg_namespace n ON tc.relnamespace=n.oid
WHERE n.nspname <> 'pg_catalog'
    AND n.nspname !~ '^pg_toast'
ORDER BY ic.relname;"

# vérification de la liste des bases
if test $# -gt 1; then
    echo "Usage: $0 [nom_base]"
    exit 1
elif test $# -eq 1; then
    BASE_PRESENTE=$(psql -XAtqc \
        "SELECT count(*) FROM pg_database WHERE datname='$1'" 2>/dev/null)
```

```
if test $BASE_PRESENTES -ne 1; then
    echo "La base de données $BASE n'existe pas."
    exit 2
fi

BASES=$1
else
BASES=$(psql -XAtqc "$REQ_LISTEBASES" 2>/dev/null)
fi

# Pour chaque base
for BASE in $BASES
do
    # Afficher la base de données
    echo "##### $BASE #####"

    # Vérification de la présence de la fonction pgstatindex
    FONCTION_PRESENTES=$(psql -XAtqc \
"SELECT count(*) FROM pg_proc WHERE proname='pgstatindex'" $BASE 2>/dev/null)
    if test $FONCTION_PRESENTES -eq 0; then
        echo "La fonction pgstatindex n'existe pas."
        echo "Veuillez installer le module pgstattuple."
        exit 3
    fi

    # pour chaque index
    echo "Récupération de la liste des index (ratio cible $TAUX_FRAGMENTATION_MAX)..."
    psql -XAtF " " -c "$REQ_LISTEINDEX" $BASE | \
while read schema table index indexoid contrainte valideite fragmentation definition
do
    # NaN (not a number) est possible si la table est vide
    # dans ce cas, une réindexation est rapide
    if test "$fragmentation" = "NaN"; then
        fragmentation=0
    fi

    # afficher index, validité et fragmentation
    if test "$valideite" = "t"; then
        chaine_valideite="valide"
    else
        chaine_valideite="invalide"
    fi
    echo "Index $index, $chaine_valideite, ratio libre ${fragmentation}%"

    # si index fragmenté ou non valide
    if test "$valideite" = "f" -o $fragmentation -gt $TAUX_FRAGMENTATION_MAX; then
    # vérifier les verrous sur l'index, attendre un peu si nécessaire
        verrous=1
        tests=0
        while test $verrous -gt 0 -a $tests -le $NB_TESTS
        do
            if test $tests -gt 0; then
                echo \
" objet verrouillé, attente de $tests secondes avant nouvelle tentative..."
                sleep $tests
            fi
        fi
    fi
fi
```

```

verrous=$(psql -XAtqc \
"SELECT count(*) FROM pg_locks WHERE relation=$indexoid" 2>/dev/null)
tests=$(( $tests + 1))
done
if test $verrous -gt 0; then
    echo " objet toujours verrouillé, pas de reindexation pour $schema.$index"
    continue
fi

# si contrainte, reindexation simple
if test "$contrainte" = "t"; then
    echo -n " reindexation de la contrainte... "
    psql -Xqc "REINDEX INDEX $schema.$index;" $BASE
    if test $? -eq 0; then
        echo "OK"
    else
        echo "PROBLEME!!"
        continue
    fi
fi
# sinon
else
# renommer <ancien nom> en <index_traitement_en_cours>
    echo -n " renommage... "
    psql -Xqc \
"ALTER INDEX $schema.$index RENAME TO $NOM_INDEX_TEMPORAIRE;" $BASE
    if test $? -eq 0; then
        echo "OK"
    else
        echo "PROBLEME!!"
        continue
    fi
fi
# create index <ancien nom>
    echo -n " création nouvel index..."
    psql -Xqc "$definition;" $BASE
# si create OK, drop index <index_traitement_en_cours>
    if test $? -eq 0; then
        echo "OK"
        echo -n " suppression ancien index..."
        psql -Xqc "DROP INDEX $schema.$NOM_INDEX_TEMPORAIRE;" $BASE
        if test $? -eq 0; then
            echo "OK"
        else
            echo "PROBLEME!!"
            continue
        fi
    fi
# sinon, renommer <index_traitement_en_cours> en <ancien nom>
    else
        echo "PROBLEME!!"
        echo -n " renommage inverse..."
        psql -Xqc \
"ALTER INDEX $schema.$NOM_INDEX_TEMPORAIRE RENAME TO $index;" $BASE
        if test $? -eq 0; then
            echo "OK"
        else
            echo "PROBLEME!!"
            continue
        fi
    fi

```

fi
fi
fi
done
done

1.8 SÉCURITÉ



- Ce qu'un utilisateur standard peut faire
 - et ne peut pas faire
- Restreindre les droits
- Chiffrement
- Corruption de données

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

Ce chapitre va faire le point sur ce qu'un utilisateur peut faire par défaut et sur ce qu'il ne peut pas faire. Nous verrons ensuite comment restreindre les droits. Enfin, nous verrons les possibilités de chiffrement et de non-corruption de PostgreSQL.

1.8.1 Droits par défaut



Un utilisateur standard peut :

- `CONNECT` : accéder à toutes les bases de données
- `CREATE` :
 - créer des objets dans le schéma `public` de **toute** base de données
 - révocation fréquente
 - ... mais plus en v15+ !
- `SELECT` : voir les données de ses tables
- `INSERT` , `UPDATE` , `DELETE` , `TRUNCATE` : les modifier
- `TEMP` : créer des objets temporaires
- `CREATE` , `USAGE ON LANGUAGE` : créer des fonctions
- `EXECUTE` : exécuter des fonctions définies par d'autres dans le schéma `public`

Par défaut, un utilisateur a beaucoup de droits.

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

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

Avant la version 15, l'utilisateur peut créer des objets dans le schéma disponible par défaut (nommé `public`) sur chacune des bases de données où il peut se connecter, même si la base ne lui appartient pas. Il est assez courant de supprimer ce droit :

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

ou de manière globale :

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

Certains DBA suppriment même le schéma de la base. Dans tous les cas, il faut faire attention à ce que le schéma ne réapparaisse pas lors d'une restauration de sauvegarde logique, car sa création est implicite.

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

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

L'utilisateur peut créer des fonctions, uniquement avec les langages de confiance, uniquement dans les schémas où il a le droit de créer des objets (donc dans le schéma `public` de toute base avant la V15). Il existe deux solutions :

- supprimer le droit d'utiliser un langage :

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

- supprimer le droit de créer des objets dans un schéma :

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

L'utilisateur peut exécuter n'importe quelle fonction, y compris définie par quelqu'un d'autre, à condition que la fonction soit dans un schéma où il a accès (dont le schéma `public`, par défaut) (et donc `public`, par défaut) Il est possible d'empêcher cela en supprimant le droit d'exécution d'une fonction :

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

1.8.2 Droits par défaut (suite)



Un utilisateur standard peut aussi :

- récupérer des informations sur l'instance
- visualiser les sources des vues et des fonctions
- Modifier des paramètres de la session :

```
SET parametre TO valeur ;
SET LOCAL parametre TO valeur ;
SHOW parametre ;
```

```
PGOPTIONS='-c param=valeur' psql
```

- Vue : `pg_settings`
- Dans `psql` : `\dconfig`

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

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

Un utilisateur peut agir sur de nombreux paramètres au sein de sa session pour modifier les valeurs par défaut du `postgresql.conf` ou ceux imposés à son rôle ou à sa base.

Un cas courant consiste à modifier la liste des schémas par défaut où chercher les tables :

```
SET search_path TO rh,admin,ventes,public ;
```

L'utilisateur peut aussi décider de s'octroyer plus de mémoire de tri :

```
SET work_mem TO '500MB' ;
```

Il est impossible d'interdire cela. Toutefois, cela permet de conserver un paramétrage par défaut prudent, tout en autorisant l'utilisation de plus de ressources quand cela s'avère nécessaire.

Les exemples suivants modifient le fuseau horaire du client, désactivent la parallélisation le temps de la session, et changent le nom de l'applicatif visible dans les outils de supervision :

```
SET timezone TO GMT ;
SET max_parallel_workers_per_gather TO 0 ;
SET application_name TO 'batch_comptabilite' ;
```

Pour une session lancée en ligne de commande, pour les outils qui utilisent la libpq, on peut fixer les paramètres à l'appel grâce à la variable d'environnement `PGOPTIONS` :

```
PGOPTIONS="-c max_parallel_workers_per_gather=0 -c work_mem=4MB" psql < requete.sql
```

La valeur en cours est visible avec :

```
SHOW parametre ;
```

ou :

```
SELECT current_setting('parametre') ;
```

ou encore :

```
SELECT * FROM pg_settings WHERE name = 'parametre' ;
```

Sous `psql` à partir de la version 15 du client, la métacommande `\dconfig` affiche les paramètres qui n'ont pas leur valeur par défaut.

Ce paramétrage est limité à la session en cours, et disparaît avec elle à la déconnexion, ou si l'on demande un retour à la valeur par défaut :

```
RESET parametre ;
```

Enfin, on peut n'appliquer des paramètres que le temps d'une transaction, c'est-à-dire jusqu'au prochain `COMMIT` ou `ROLLBACK` :

```
SET LOCAL work_mem TO '100MB' ;
```

De nombreux paramètres sont cependant non modifiables, ou réservés aux superutilisateurs.

1.8.3 Droits par défaut (suite)



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

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

Il ne peut pas accéder au contenu (aux données) d'objets créés par d'autres utilisateurs. Ces derniers doivent lui donner ce droit explicitement : `GRANT USAGE ON SCHEMA secret TO utilisateur ;` pour lire un schéma, ou `GRANT SELECT ON TABLE matable TO utilisateur ;` pour lire une table.

De même, il ne peut pas modifier le contenu et la définition d'objets créés par d'autres utilisateurs. Là-aussi, ce droit doit être donné explicitement : `GRANT INSERT,DELETE,UPDATE,TRUNCATE ON TABLE matable TO utilisateur ;`

Il existe d'autres droits plus rares, dont :

- `GRANT TRIGGER ON TABLE ...` autorise la création de trigger ;
- `GRANT REFERENCES ON TABLE ...` autorise la création d'une clé étrangère pointant vers cette table (ce qui est interdit par défaut car cela interdit au propriétaire de supprimer ou modifier des lignes, entre autres) ;
- `GRANT USAGE ON SEQUENCE...` autorise l'utilisation d'une séquence.

Par facilité, on peut octroyer des droits en masse :

- `GRANT ALL PRIVILEGES ON TABLE matable TO utilisateur ;`
- `GRANT SELECT ON TABLE matable TO public ;`
- `GRANT SELECT ON ALL TABLES IN SCHEMA monschema ;`

1.8.4 Restreindre les droits



- Sur les connexions
 - `pg_hba.conf`
- Sur les objets
 - `GRANT / REVOKE`
 - `SECURITY LABEL`
- Sur les fonctions
 - `SECURITY DEFINER`
 - `LEAKPROOF`
- Sur les vues
 - `security_barrier`
 - `WITH CHECK OPTION`

Pour sécuriser plus fortement une instance, il est nécessaire de restreindre les droits des utilisateurs.

Connexions :

Cela commence par la gestion des connexions. Les droits de connexion sont généralement gérés via le fichier de configuration `pg_hba.conf`. Cette configuration a déjà été abordée dans le chapitre *Droits de connexion* de ce module de formation.

GRANT & REVOKE :

Cela passe ensuite par les droits sur les objets. On dispose pour cela des instructions `GRANT` et `REVOKE`, qui ont été expliquées dans le chapitre *Droits sur les objets* de ce module de formation.

SECURITY LABEL :

Il est possible d'aller plus loin avec l'instruction `SECURITY LABEL`. Un label de sécurité est un commentaire supplémentaire pris en compte par un module de sécurité qui disposera de la politique de sécurité. Le seul module de sécurité actuellement disponible est un module contrib pour l'intégration à SELinux, appelé `sepgsql`¹³.

SECURITY DEFINER & LEAKPROOF :

Certains objets disposent de droits particuliers. Par exemple, les fonctions disposent des clauses `SECURITY DEFINER` et `LEAKPROOF`.

`SECURITY DEFINER` indique au système que la fonction doit s'exécuter avec les droits de son propriétaire (et non pas avec ceux de l'utilisateur l'exécutant). Cela permet d'éviter de donner des droits d'accès à certains objets.

`LEAKPROOF` permet de dire à PostgreSQL que cette fonction ne peut pas occasionner de fuites d'informations. Ainsi, le planificateur de PostgreSQL sait qu'il peut optimiser l'exécution des fonctions.

Vues avec `security_barrier` :

Quant aux vues, elles disposent d'une option appelée `security_barrier`. Certains utilisateurs créent des vues pour filtrer des lignes, afin de restreindre la visibilité sur certaines données. Or, cela peut se révéler dangereux si un utilisateur malintentionné a la possibilité de créer une fonction car il peut facilement contourner cette sécurité si cette option n'est pas utilisée. Voici un exemple complet :

```
CREATE TABLE elements (id int, contenu text, prive boolean);

CREATE TABLE

INSERT INTO elements (id, contenu, prive)
VALUES (1, 'a', false), (2, 'b', false), (3, 'c super prive', true),
       (4, 'd', false), (5, 'e prive aussi', true) ;

INSERT 0 5

SELECT * FROM elements ;
```

id	contenu	prive
1	a	f
2	b	f
3	c super prive	t
4	d	f
5	e prive aussi	t

La table `elements` contient cinq lignes, dont deux considérées comme privées. Nous allons donc créer une vue ne permettant de voir que les lignes publiques, sauf pour un utilisateur particulier :

```
CREATE OR REPLACE VIEW elements_public AS
SELECT * FROM elements
```

¹³<https://docs.postgresql.fr/current/sepgsql.html>

```
WHERE CASE WHEN current_user = 'guillaume' THEN TRUE
        ELSE NOT prive
END ;
```

```
CREATE VIEW
```

Ce nouvel utilisateur ne verra donc pas les lignes privées masquées par la vue :

```
CREATE USER u1;
```

```
CREATE ROLE
```

```
GRANT SELECT ON elements_public TO u1;
```

```
GRANT
```

```
\c - u1
```

You are now connected to database "demo" as user "u1".

```
SELECT * FROM elements ;
```

```
ERROR: permission denied for relation elements
```

```
SELECT * FROM elements_public ;
```

id	contenu	prive
1	a	f
2	b	f
4	d	f

L'utilisateur **u1** n'a pas le droit de lire directement la table `elements` mais a le droit d'y accéder via la vue `elements_public`, uniquement pour les lignes dont le champ `prive` est à `false`.

Avec une simple fonction, **u1** peut contourner le problème. (Il devra toutefois avoir le droit d'écrire dans un schéma de la base, ce qui est par défaut possible dans `public` avant la version 15.)

```
CREATE SCHEMA schemau1;
```

```
CREATE OR REPLACE FUNCTION schemau1.abracadabra (integer, text, boolean)
```

```
RETURNS bool AS $$
```

```
BEGIN
```

```
    RAISE NOTICE '% - % - %', $1, $2, $3 ;
```

```
    RETURN true;
```

```
END$$
```

```
LANGUAGE plpgsql
```

```
COST 0.000000000000000000000001 ;
```

```
CREATE SCHEMA
```

```
CREATE FUNCTION
```

```
SELECT * FROM elements_public
```

```
WHERE schemau1.abracadabra(id, contenu, prive) ;
```

```
NOTICE: 1 - a - f
```

```
NOTICE: 2 - b - f
```

```
NOTICE: 3 - c super prive - t
```

```
NOTICE: 4 - d - f
```

NOTICE: 5 - e prive aussi - t

id	contenu	prive
1	a	f
2	b	f
4	d	f

Les lignes secrètes ne font toujours pas partie du résultat, mais la fonction est capable d'y accéder, et de les afficher. Que s'est-il passé ? Pour comprendre, il suffit de regarder le plan d'exécution de cette requête :

```
EXPLAIN SELECT * FROM elements_public
WHERE schemau1.abracadabra(id, contenu, prive) ;
```

QUERY PLAN

```
Seq Scan on elements (cost=0.00..28.75 rows=208 width=37)
  Filter: (schemau1.abracadabra(id, contenu, prive)
    AND CASE WHEN (CURRENT_USER = 'guillaume'::name) THEN true ELSE (NOT
  ↪  prive) END)
```

La fonction `abracadabra` a un coût délibérément si faible que PostgreSQL l'exécute avant le filtre de la vue : la fonction voit toutes les lignes de la table.

L'option `security_barrier` permet d'interdire cette optimisation au planificateur :

```
\c - postgres
```

You are now connected to database "demo" as user "postgres".

```
ALTER VIEW elements_public SET (security_barrier);
```

```
ALTER VIEW
```

```
\c - u1
```

You are now connected to database "demo" as user "u1".

```
SELECT * FROM elements_public WHERE schemau1.abracadabra(id, contenu, prive);
```

NOTICE: 1 - a - f

NOTICE: 2 - b - f

NOTICE: 4 - d - f

id	contenu	prive
1	a	f
2	b	f
4	d	f

La fonction n'affiche plus les données privées. Son plan montre qu'elle filtre par la vue, puis appelle la fonction :

```
EXPLAIN SELECT * FROM elements_public
WHERE schemau1.abracadabra(id, contenu, prive);
```


QUERY PLAN

```

Subquery Scan on elements_public (cost=0.00..35.00 rows=208 width=37)
  Filter: schemau1.abracadabra(elements_public.id, elements_public.contenu,
↪ elements_public.prive)
  -> Seq Scan on elements (cost=0.00..28.75 rows=625 width=37)
      Filter: CASE WHEN (CURRENT_USER = 'guillaume'::name) THEN true ELSE (NOT
↪ prive) END

```

Noter que `security_barrier` peut avoir un effet négatif sur les performances puisqu'il interdit de « pousser » des critères de recherche dans la vue. Voir aussi cet article, par Robert Haas¹⁴.

Vue WITH CHECK OPTION :

Écrire à travers une vue permet de modifier les lignes même si la nouvelle valeur les lui rend inaccessibles :

```

UPDATE elements_public
SET prive = true
WHERE id = 1 ;

```

```
UPDATE 1
```

```
SELECT * FROM elements_public ;
```

```

id | contenu | prive
---+-----+-----
 2 | b       | f
 4 | d       | f

```

Pour bloquer ce comportement, la vue peut porter l'option `WITH CHECK OPTION` :

```

CREATE OR REPLACE VIEW elements_public
WITH (security_barrier)
AS
  SELECT * FROM elements
  WHERE CASE WHEN current_user = 'guillaume'
  THEN true ELSE NOT prive END
WITH CHECK OPTION ;

```

```
\c - u1
```

You are now connected to database "demo" as user "u1".

```

UPDATE elements_public
SET prive = true
WHERE id = 2 ;

```

```

ERROR: new row violates check option for view "elements_public"
DETAIL: Failing row contains (prive) = (t).

```

¹⁴<https://rhaas.blogspot.com/2012/03/security-barrier-views.html>

1.8.5 Arrêter une requête ou une session



- Annuler une requête
 - `pg_cancel_backend (pid int)`
- Fermer une connexion
 - `pg_terminate_backend(pid int, timeout bigint)`
 - `kill -SIGTERM pid`, `kill -15 pid` (éviter)
- Jamais `kill -9 !!`

Les fonctions `pg_cancel_backend` et `pg_terminate_backend` sont le plus souvent utilisées. Le paramètre est le numéro du processus auprès de l'OS. À partir de la version 14, `pg_terminate_backend` comprend un deuxième argument, dont la valeur par défaut est 0. Si cet argument n'est pas indiqué ou vaut 0, la fonction renvoie le booléen `true` si elle a réussi à envoyer le signal. Ce résultat n'indique donc pas la bonne terminaison du processus serveur visé. À une valeur supérieure à 0, la fonction attend que le processus visé soit arrêté. S'il ne s'est pas arrêté dans le temps indiqué par cette valeur (en millisecondes), la valeur `false` est renvoyée avec un message de niveau `WARNING`.

La première permet d'annuler une requête en cours d'exécution. Elle requiert un argument, à savoir le numéro du PID du processus `postgres` exécutant cette requête. Généralement, l'annulation est immédiate. Voici un exemple de son utilisation.

L'utilisateur, connecté au processus de PID 10901 comme l'indique la fonction `pg_backend_pid`, exécute une très grosse insertion :

```
b1=# SELECT pg_backend_pid();
 pg_backend_pid
-----
          10901

b1=# INSERT INTO t4 SELECT i, 'Ligne '||i
FROM generate_series(2000001, 3000000) AS i;
```

Supposons qu'on veuille annuler l'exécution de cette requête. Voici comment faire à partir d'une autre connexion :

```
b1=# SELECT pg_cancel_backend(10901);
 pg_cancel_backend
-----
t
```

L'utilisateur qui a lancé la requête d'insertion verra ce message apparaître :

```
ERROR: canceling statement due to user request
```

Si la requête du `INSERT` faisait partie d'une transaction, la transaction elle-même devra se conclure par un `ROLLBACK` à cause de l'erreur. À noter cependant qu'il n'est pas possible d'annuler une transaction qui n'exécute rien à ce moment. En conséquence, `pg_cancel_backend` ne suffit pas pour parer à une session en statut `idle in transaction`.

Il est possible d'aller plus loin en supprimant la connexion d'un utilisateur. Cela se fait avec la fonction `pg_terminate_backend` qui se manie de la même manière :

```
b1=# SELECT pid, datname, username, application_name, state
      FROM pg_stat_activity WHERE backend_type = 'client backend';
```

procpid	datname	username	application_name	state
13267	b1	u1	psql	idle
10901	b1	guillaume	psql	active

```
b1=# SELECT pg_terminate_backend(13267);
```

```
pg_terminate_backend
-----
t
```

```
b1=# SELECT pid, datname, username, application_name, state
      FROM pg_stat_activity WHERE backend_type = 'client backend';
```

procpid	datname	username	application_name	state
10901	b1	guillaume	psql	active

L'utilisateur de la session supprimée verra un message d'erreur au prochain ordre qu'il enverra. `psql` se reconnecte automatiquement mais cela n'est pas forcément le cas d'autres outils client.

```
b1=# select 1 ;
```

```
FATAL: terminating connection due to administrator command
la connexion au serveur a été coupée de façon inattendue
      Le serveur s'est peut-être arrêté anormalement avant ou durant le
      traitement de la requête.
La connexion au serveur a été perdue. Tentative de réinitialisation : Succès.
Temps : 7,309 ms
```

Depuis la ligne de commande du serveur, un `kill <pid>` (c'est-à-dire `kill -SIGTERM` ou `kill -15`) a le même effet qu'un `SELECT pg_terminate_backend (<pid>)`. Cette méthode n'est pas recommandée car il n'y a pas de vérification que vous tuez bien un processus postgres.

N'utilisez jamais `kill -9 <pid>` (ou `kill -SIGKILL`), ou (sous Windows) `taskkill /f /pid <pid>` pour tuer une connexion : l'arrêt est alors brutal, et le processus principal n'a aucun moyen de savoir pourquoi. Pour éviter une corruption de la mémoire partagée, il va arrêter et redémarrer immédiatement tous les processus, déconnectant tous les utilisateurs au passage !

L'utilisation de `pg_terminate_backend` et `pg_cancel_backend` n'est disponible que pour les utilisateurs appartenant au même rôle que l'utilisateur à déconnecter, les utilisateurs membres du rôle `pg_signal_backend` (à partir de la 9.6) et bien sûr les superutilisateurs.

1.8.6 Chiffrements



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

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

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

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

1.8.7 En cas de crash



- Ça arrive
- Redémarrage sans souci, en général
- Quelques statistiques d'activité sont perdues
 - `ANALYZE` (voire `VACUUM`)
- Et analyser la cause

Même si PostgreSQL est conçu pour être le plus robuste possible, un arrêt brutal est toujours possible, pour des causes externes (perte de courant, plantage du système d'exploitation, corruption de fichier sur le disque...), liées à l'utilisation de PostgreSQL (requêtes menant à une saturation mémoire sur un système mal paramétré...) ou exceptionnellement internes (extension buggée, bug de PostgreSQL).

Dans l'écrasante majorité des cas, le système de journalisation de PostgreSQL vous permettra de redémarrer sans perte de données (sinon ce qui n'a pas été committé). Le rejeu des journaux est souvent si rapide qu'un redémarrage peut passer inaperçu.

Malheureusement, les statistiques d'activités sont perdues lors d'un arrêt brutal (ou d'un arrêt en mode *immediate*, ou d'une restauration physique). Cela peut provoquer par la suite un retard dans le déclenchement de l'autovacuum et la mise à jour des statistiques. Pour éviter cela, le DBA doit penser à relancer un ordre `ANALYZE`, ainsi que, dans l'idéal, un ordre `VACUUM`.

Surtout, il faut comprendre quelle est la cause pour que le problème ne se reproduise pas.

1.8.8 Corruption de données



- `initdb --data-checksums`
- Détecte les corruptions silencieuses
- Impact faible sur les performances
- Fortement conseillé !

PostgreSQL ne verrouille pas tous les fichiers dès son ouverture. Sans mécanisme de sécurité, il est donc possible de modifier un fichier sans que PostgreSQL s'en rende compte, ce qui aboutit à une corruption silencieuse.

L'apparition des sommes de contrôles (*checksums*) permet de se prémunir contre des corruptions silencieuses de données. Leur mise en place est fortement recommandée sur une nouvelle instance.

À titre d'exemple, créons une instance sans utiliser les *checksums*, et une autre qui les utilisera :

```
$ initdb --pgdata /tmp/sans_checksums/
$ initdb --pgdata /tmp/avec_checksums/ --data-checksums
```

Insérons une valeur de test, sur chacune des deux instances :

```
postgres=# CREATE TABLE test (name text);
CREATE TABLE
postgres=# INSERT INTO test (name) VALUES ('toto');
INSERT 0 1
```

On récupère le chemin du fichier de la table pour aller le corrompre à la main (seul celui sans *checksums* est montré en exemple).

```
postgres=# SELECT pg_relation_filepath('test');
pg_relation_filepath
-----
base/14415/16384
```

Instance arrêtée (pour ne pas être gêné par le cache), on va s'attacher à corrompre ce fichier, en remplaçant la valeur « toto » par « goto » avec un éditeur hexadécimal :

```
$ hexedit /tmp/sans_checksums/base/base/14415/16384
```

Enfin, on peut ensuite exécuter des requêtes sur ces deux instances.

Sans *checksums* :

```
postgres=# TABLE test;
```

```
 name
-----
 goto
```

Avec *checksums* :

```
postgres=# TABLE test;
```

```
WARNING: page verification failed, calculated checksum 29393
         but expected 24228
ERROR:  invalid page in block 0 of relation base/14415/16384
```

L'outil `pg_verify_checksums`, disponible depuis la version 11 et renommé `pg_checksums` en 12, permet de vérifier une instance complète :

```
$ pg_checksums -D /tmp/avec_checksums
```

```
pg_checksums: error: checksum verification failed
  in file "/tmp/avec_checksums/base/14415/16384",
  block 0: calculated checksum 72D1 but block contains 5EA4
Checksum operation completed
Files scanned:  919
Blocks scanned: 3089
Bad checksums:  1
Data checksum version: 1
```

En pratique, si vous utilisez PostgreSQL 9.5 au moins et si votre processeur supporte les instructions SSE 4.2 (voir dans `/proc/cpuinfo`), il n'y aura pas d'impact notable en performances. Par contre vous générerez un peu plus de journaux.

L'outil `pg_checksums` permet aussi d'activer et de désactiver la gestion des sommes de contrôle (instance arrêtée). Ceci n'était pas possible avant la version 12. Il fallait donc le faire dès la création de l'instance.

1.9 CONCLUSION



- PostgreSQL demande peu de travail au quotidien
- À l'installation :
 - veiller aux accès et aux droits
 - mettre la maintenance en place
- Pour le reste, surveiller :
 - les scripts automatisés
 - le contenu des journaux applicatifs
- Supervisez le serveur !

1.9.1 Pour aller plus loin



- Documentation officielle, chapitre Planifier les tâches de maintenance¹⁵
- Documentation officielle, chapitre Catalogues système¹⁶
- Opérations de maintenance sous PostgreSQL¹⁷
- Total security in a PostgreSQL database (copie)¹⁸
- Managing rights in PostgreSQL, from the basics to SE PostgreSQL¹⁹, Nicolas Thauvin, 2011

1.9.2 Questions



N'hésitez pas, c'est le moment !

1.10 QUIZ



https://dali.bo/f_quiz

1.11 TRAVAUX PRATIQUES

1.11.1 Traces maximales



But : suivre toutes les requêtes dans les traces

À titre pédagogique et pour alimenter un rapport pgBadger plus tard, toutes les requêtes vont être tracées.

Dans `postgresql.conf`, positionner ceci :

```
log_min_duration_statement = 0
log_temp_files = 0
log_autovacuum_min_duration = 0
lc_messages = 'C'
log_line_prefix = '%t [%p]: db=%d,user=%u,app=%a,client=%h '
```

Puis passer à `on` les paramètres suivants s'ils ne le sont pas déjà :

```
log_checkpoints
log_connections
log_disconnections
log_lock_waits
```

Recharger la configuration.

Laisser une fenêtre ouverte pour voir défiler le contenu des traces.

1.11.2 Méthode d'authentification



But : Gérer les rôles et les permissions

Activer la méthode d'authentification `scram-sha-256` dans `postgresql.conf` si elle n'est pas déjà en place.

Consulter les droits définis dans `pg_hba.conf` au travers de la vue système `pg_hba_file_rules`.

Dans `pg_hba.conf`, supprimer les accès avec la méthode `trust` pouvant rester après les précédents exercices. Vérifier dans la vue avant de recharger la configuration.

1.11.3 Création des bases



But : Créer des bases appartenant à un utilisateur non privilégié

Créer un utilisateur nommé **testperf** avec attribut `LOGIN`.

Créer une base **pgbench** lui appartenant.

Vérifier que la connexion ne marche pas encore depuis votre compte habituel, y compris avec `-h localhost`.

Créer un rôle **patron** avec attribut `LOGIN`, et une base **entreprise** lui appartenant.

1.11.4 Mots de passe



But : Mise en place de l'authentification par mot de passe

Créer des mots de passe pour les rôles **patron** et **testperf**.

Consulter la table `pg_authid` pour voir la version chiffrée.

Pour ouvrir les accès :

- ajuster `pg_hba.conf` pour permettre l'accès à la base **pgbench** uniquement à l'utilisateur **testperf**, permettre l'accès à la base **entreprise** à tout utilisateur, en local avec son mot de passe, en authentification `scram-sha-256` ;
- vérifier avec la vue `pg_hba_file_rules` ;
- recharger la configuration ;
- tester la connexion.

Créer un fichier `.pgpass` dans votre répertoire utilisateur (`/home/dalibo`) ou dans celui de l'utilisateur `postgres` (`/var/lib/postgresql`) pour qu'il puisse se connecter aux bases **entreprise** et **pgbench** sans entrer le mot de passe.

Le superutilisateur PostgreSQL **postgres** peut-il se connecter aux bases **entreprise** et **pgbench** ? Si non, comment lui permettre ?

Remplir la base **pgbench** et générer un peu d'activité :

```
/usr/postgresql-15/bin/pgbench -i -s1 --foreign-keys pgbench -U testperf  
/usr/postgresql-15/bin/pgbench -P1 -T3 pgbench -U testperf
```

Aucun mot de passe ne doit être demandé, avec ou sans `-h localhost`.

Compléter `pg_ident.conf` pour pouvoir se connecter au rôle **postgres** depuis le compte système habituel (**dalibo**), en `local`.

Tester la connexion depuis **postgres** au rôle **postgres**, de **dalibo** à **postgres**.

1.11.5 Rôles et permissions



But : Gérer les rôles et les permissions sur les tables

Sous les utilisateurs **dalibo** comme **postgres**, créer un nouveau fichier `~/.psqlrc` contenant `\set PROMPT1 '%n@%/%R%# '` pour que l'invite indique quels sont les utilisateur et base en cours.

Ajouter à la base de données **entreprise** la table `facture (id int, objet text, creations timestamp)`. Elle appartiendra à **patron**, administrateur de la base.

Créer un rôle **secretariat** sans droit de connexion, mais qui peut visualiser, ajouter, mettre à jour et supprimer des éléments de la table `facture`.

Créer un rôle **boulier** qui peut se connecter et appartient au rôle **secretariat**, avec un mot de passe (à ajouter au `.pgpass`).

Vérifier la création des deux rôles.

En tant que **boulier**, créer une table `brouillon` identique à `facture`. Dans quel schéma cela se passe-t-il ? Avec PostgreSQL 15 ou supérieur, il y a une étape supplémentaire.

Vérifier les tables présentes et les droits `\dp`. Comment les lire ?

À l'aide du rôle **boulier** : insérer 2 factures ayant pour objet « Vin de Bordeaux » et « Vin de Bourgogne » avec la date et l'heure courante.

Afficher le contenu de la table `facture`.

Mettre à jour la deuxième facture avec la date et l'heure courante.

Supprimer la première facture.

Retirer les droits `DELETE` sur la table `facture` au rôle **secretariat**.

Vérifier qu'il n'est plus possible de supprimer la deuxième facture avec le rôle **boulier**.

En tant que **patron**, créer une table **produit** contenant une colonne texte nommée `appellation` et la remplir avec des noms de boissons.

Afficher les droits sur cette table avec `\dt` et `\dp`. Vérifier que le rôle **boulier** appartenant au rôle **secretariat** ne peut pas sélectionner les produits contenus dans la table `produit`.

Retirer tous les droits pour le groupe **secretariat** sur la table `produit`.

Que deviennent les droits affichés ? **boulier** peut-il lire la table ?

Autoriser l'utilisateur **boulier** à accéder à la table `produit` en lecture.

Vérifier que **boulrier** peut désormais accéder à la table `produit`.

Créer un rôle **tina** appartenant au rôle **secretariat**, avec l'attribut `LOGIN`, mais n'héritant pas des droits à la connexion. Vérifier les droits avec `\du`. Lui donner un mot de passe.

Vérifier que **tina** ne peut pas accéder à la table `facture`.

En tant que **tina**, activer le rôle **secretariat** (`SET ROLE`).

Vérifier que **tina** possède maintenant les droits du rôle **secretariat**. Sélectionner les données de la table `facture`.

1.11.6 Autorisation d'accès distant



But : Mettre en place les accès dans `pg_hba.conf`.

Autoriser tous les membres du réseau local à se connecter avec un mot de passe (autorisation en IP sans SSL) avec les utilisateurs **boulrier** ou **tina**. Tester avec l'IP du serveur avant de demander aux voisins de tester.

1.11.7 VACUUM, VACUUM FULL, DELETE, TRUNCATE



But : Effacer des données, distinguer `VACUUM` et `VACUUM FULL`.

Désactiver le démon autovacuum de l'instance.

Se connecter à la base **pgbench** en tant que **testperf**.

Grâce aux fonctions `pg_relation_size` et `pg_size_pretty`, afficher la taille de la table `pgbench_accounts`.

Copier le contenu de la table dans une nouvelle table (`pba_copie`).

Supprimer le contenu de la table `pba_copie`, à l'exception de la dernière ligne (`aid=100000`), avec un ordre `DELETE`. Quel est alors l'espace disque utilisé par cette table ?

Insérer le contenu de la table `pgbench_accounts` dans la table `pba_copie`. Quel est alors l'espace disque utilisé par la table ?

Effectuer un `VACUUM` simple sur `pba_copie`. Vérifier la taille de la base.

Vider à nouveau la table `pba_copie` des lignes d'`aid` inférieur à 100 000. Insérer à nouveau le contenu de la table `pgbench_accounts`. L'espace mis à disposition a-t-il été utilisé ?

Voir la taille de la base. Supprimer la table `pba_copie`. Voir l'impact sur la taille de la base.

Tout d'abord, repérer les tailles des différentes tables et le nombre de lignes de chacune.

Pour amplifier le phénomène à observer, on peut créer une session de très longue durée, laissée ouverte sans `COMMIT` ni `ROLLBACK`. Il faut qu'elle ait consulté une des tables pour que l'effet soit visible :

```
testperf@pgbench=> BEGIN ;
BEGIN
Temps : 0,608 ms

testperf@pgbench=> SELECT count(*) FROM pgbench_accounts ;

 count
-----
 100000
(1 ligne)

Temps : 26,059 ms
testperf@pgbench=> SELECT pg_sleep (10000) ;
```

Depuis un autre terminal, générer de l'activité sur la table, ici avec 10 000 transactions sur 20 clients :

```
PGOPTIONS='-c synchronous_commit=off' \
/usr/pgsql-15/bin/pgbench -U testperf -d pgbench \
--client=20 --jobs=2 -t 10000 --no-vacuum
```

(NB : La variable d'environnement `PGOPTIONS` restreint l'utilisation des journaux de transaction pour accélérer les écritures (données NON critiques ici). Le `--no-vacuum` est destiné à éviter que l'outil demande lui-même un `VACUUM`. Le test dure quelques minutes. Le relancer au besoin.)

(Optionnel) C'est l'occasion d'installer l'outil `pg_activity` depuis les dépôts du PGDG (il peut y avoir besoin du dépôt EPEL) et de le lancer en tant que **postgres** pour voir ce qui se passe dans la base.

Comparer les nouvelles tailles des tables (taille sur le disque et nombre de lignes). La table `pg_stat_user_tables` contient l'activité sur chaque table. Comment s'expliquent les évolutions ?

Exécuter un `VACUUM FULL VERBOSE` sur `pgbench_tellers`.

Exécuter un `VACUUM FULL VERBOSE` sur `pgbench_accounts`.

Effectuer un `VACUUM FULL VERBOSE`. Quel est l'impact sur la taille de la base ?

Créer `copie1` et `copie2` comme des copies de `pgbench_accounts`, données incluses.

Effacer le contenu de `copie1` avec `DELETE`.

Effacer le contenu de `copie2` avec `TRUNCATE`.

Quelles sont les tailles de ces deux tables après ces opérations ?

Réactiver l'autovacuum de l'instance.

Attendre quelques secondes et voir si `copie1` change de taille.

1.11.8 Statistiques



But : Savoir trouver les statistiques des données et les mettre à jour

Créer une table `copie3`, copie de `pgbench_accounts`.

Dans la vue système `pg_stats`, afficher les statistiques collectées pour la table `copie3`.

Lancer la collecte des statistiques pour cette table uniquement.

Afficher de nouveau les statistiques.

1.11.9 Réindexation



But : Réindexer

Recréer les index de la table `pgbench_accounts`.

Comment recréer tous les index de la base **pgbench** ?

Comment recréer uniquement les index des tables systèmes ?

Quelle est la différence entre la commande `REINDEX` et la séquence `DROP INDEX` + `CREATE INDEX` ?

1.11.10 Traces



But : Gérer les fichiers de traces

Quelle est la méthode de gestion des traces utilisée par défaut ?

Paramétrer le programme interne de rotation des journaux :

- modifier le fichier `postgresql.conf` pour utiliser le *logging collector* ;
- les traces doivent désormais être sauvegardés dans le répertoire `/var/lib/pgsql/traces` ;
- la rotation des journaux doit être automatisée pour générer un nouveau fichier de logs toutes les 30 minutes, quelle que soit la quantité de logs archivés ; le nom du fichier de-

vra donc comporter les minutes.

- Tester en forçant des rotations avec la fonction `pg_rotate_logfile`.
- Augmenter la trace (niveau `info`).

Comment éviter l'accumulation des fichiers ?

1.12 TRAVAUX PRATIQUES (SOLUTIONS)



Tout ce qui suit suppose :

- une installation sous Rocky Linux avec les paquets RPM de yum.postgresql.org²⁰ ;
- une instance installée avec les options par défaut ;
- un administrateur dont le compte habituel sur la machine, non privilégié, est nommé **dalibo**.

1.12.1 Traces maximales

But : suivre toutes les requêtes dans les traces

À titre pédagogique et pour alimenter un rapport pgBadger plus tard, toutes les requêtes vont être tracées.

Dans `postgresql.conf`, positionner ceci :

```
log_min_duration_statement = 0
log_temp_files = 0
log_autovacuum_min_duration = 0
lc_messages = 'C'
log_line_prefix = '%t [%p]: db=%d,user=%u,app=%a,client=%h '
```



Éviter de faire cela en production, surtout `log_min_duration_statement = 0` ! Sur une base très active, les traces peuvent rapidement monter à plusieurs dizaines de gigaoctets !

Dans le présent TP, il faut surveiller l'espace disque pour cette raison.

Puis passer à `on` les paramètres suivants s'ils ne le sont pas déjà :

```
log_checkpoints
log_connections
log_disconnections
log_lock_waits
```

Recharger la configuration.

Laisser une fenêtre ouverte pour voir défiler le contenu des traces.

Dans `postgresql.conf` :

```
log_min_duration_statement = 0
log_temp_files = 0
```

```
log_autovacuum_min_duration = 0
lc_messages='C'
log_line_prefix = '%t [%p]: db=%d,user=%u,app=%a,client=%h '
log_checkpoints = on
log_connections = on
log_disconnections = on
log_lock_waits = on
```

```
SELECT pg_reload_conf() ;
```

```
pg_reload_conf
-----
t
```

```
SHOW log_min_duration_statement ;
```

```
log_min_duration_statement
-----
0
```

Laisser une fenêtre ouverte avec le fichier, par exemple avec :

```
tail -f /var/lib/pgsql/15/data/log/postgresql-Wed.log
```

La moindre requête doit y apparaître, y compris quand l'utilisateur effectue un simple `\d`.

1.12.2 Méthode d'authentification



But : Gérer les rôles et les permissions

Activer la méthode d'authentification `scram-sha-256` dans `postgresql.conf` si elle n'est pas déjà en place.

Cette méthode existe depuis PostgreSQL 10 mais n'est le défaut que depuis PostgreSQL 14.

Dans `postgresql.conf` :

```
password_encryption = scram-sha-256
```

Ne pas oublier de recharger la configuration. Depuis psql en tant que **postgres** :

```
SELECT pg_reload_conf() ;
```

```
pg_reload_conf
-----
t
```

Alternative depuis le shell :

```
# systemctl reload postgresql-15
```

```
SHOW password_encryption ;
```

```
password_encryption
-----
scram-sha-256
```

Consulter les droits définis dans `pg_hba.conf` au travers de la vue système `pg_hba_file_rules`.

La vue permet de voir le contenu de `pg_hba.conf` avant de le recharger.

```
SELECT * FROM pg_hba_file_rules ;
```

ln	type	database	user_name	address	netmask	auth_method	...
80	local	{pgbench}	{all}			trust	
81	local	{all}	{all}			peer	
84	host	{all}	{all}	127.0.0.1	255.255.255.255	scram-sha-256	
86	host	{all}	{all}	:::1	ffff:ffff:ffff:ffff:...	scram-sha-256	
89	local	{replica...	{all}			peer	
90	host	{replica...	{all}	127.0.0.1	255.255.255.255	scram-sha-256	
91	host	{replica...	{all}	:::1	ffff:ffff:ffff:ffff:...	scram-sha-256	

Dans `pg_hba.conf`, supprimer les accès avec la méthode `trust` pouvant rester après les précédents exercices. Vérifier dans la vue avant de recharger la configuration.

Dans le fichier `pg_hba.conf`, supprimer les lignes avec la méthode `trust`. La vue se met à jour immédiatement. En cas d'erreur de syntaxe dans le fichier, elle doit aussi indiquer une erreur.

Puis on recharge :

```
SELECT pg_reload_conf() ;
```

1.12.3 Création des bases



But : Créer des bases appartenant à un utilisateur non privilégié

Créer un utilisateur nommé **testperf** avec attribut `LOGIN`.

Au choix, on peut utiliser les commandes shell ou les commandes SQL qu'elles affichent :

```
$ sudo -iu postgres
```

```
$ createuser --login --echo testperf
SELECT pg_catalog.set_config('search_path', '', false);
CREATE ROLE testperf NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

Créer une base **pgbench** lui appartenant.

```
$ createdb --echo pgbench --owner testperf
SELECT pg_catalog.set_config('search_path', '', false);
CREATE DATABASE pgbench OWNER testperf;
```

Vérifier que la connexion ne marche pas encore depuis votre compte habituel, y compris avec `-h localhost`.

La connexion ne peut se faire car dans `pg_hba.conf` ne figure, en `local`, que l'accès `peer` pour un utilisateur nommé **postgres** (nom système comme nom du rôle.) Elle ne fonctionnera pas pour des utilisateurs nommés différemment.

Corriger cela impliquerait de modifier `pg_ident.conf`.

```
$ psql -U testerf pgbench
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432"
↪ failed: FATAL: Peer authentication failed for user "testerf"
```

Si l'on ajoute `-h localhost`, l'accès est contrôlé via une ligne `host`, et exige un mot de passe qui n'existe pas encore :

```
$ psql -U testerf pgbench -h localhost
Password for user testerf:
```

Nous créerons ces mots de passe plus bas.

Créer un rôle **patron** avec attribut `LOGIN`, et une base **entreprise** lui appartenant.

```
$ createuser --login patron
$ createdb --owner patron entreprise
```

Ce qui est équivalent à :

```
CREATE ROLE patron LOGIN;
CREATE DATABASE entreprise OWNER patron;
```

Noter que, là encore, c'est le superutilisateur **postgres** qui crée la base et affecte les droits à **patron**. Celui-ci n'a pas le droit de créer des bases.

1.12.4 Mots de passe



But : Mise en place de l'authentification par mot de passe

Créer des mots de passe pour les rôles **patron** et **testperf**.

Déclarer le mot de passe se fait facilement depuis `psql` (en tant que superutilisateur **postgres**) :

```
postgres=# \password testperf
```

```
Saisissez le nouveau mot de passe :
Saisissez-le à nouveau :
```

```
postgres=# \password patron
```

```
Saisissez le nouveau mot de passe :
Saisissez-le à nouveau :
```

Un outil courant pour générer des mots de passe aléatoires longs est `pwgen` :

```
$ pwgen 20 1
jahT0eeRov2aiQuae1iM
```

Si `pwgen` n'est pas présent sur le système, un bon mot de passe peut être généré ainsi :

```
$ echo "faitespasserunchatsurleclavier" | md5sum
b0cdc36ff6c986b3930bfc269f37f3f2
```

Pour l'exercice, il est possible de donner le même mot de passe à tous les utilisateurs (ce que personne ne fait en production, bien sûr).

Consulter la table `pg_authid` pour voir la version chiffrée.

Noter que, même identiques, les mots de passe n'ont pas la même signature.

```
SELECT * FROM pg_authid WHERE rolname IN ('testperf', 'patron') \gx
```

```
-[ RECORD 1 ]--
oid          | 25097
rolname      | patron
rolsuper     | f
rolinherit   | t
rolcreaterole | f
rolcreatedb  | f
rolcanlogin  | t
rolreplication | f
rolbypassrpls | f
rolconnlimit | -1
rolpassword  | SCRAM-SHA-256$4096:a0IE9MKLZRTYd9FLXDX0g==$wT0rQtaoLI2gpP...
rolvaliduntil |
-[ RECORD 2 ]--
oid          | 25096
rolname      | testperf
rolsuper     | f
rolinherit   | t
rolcreaterole | f
rolcreatedb  | f
rolcanlogin  | t
```

```

rolreplication | f
rolbypassrls  | f
rolconlimit   | -1
rolpassword   | SCRAM-SHA-256$4096:Xd9Ndrb6LjGAVyTek3sig==$ofeTaBumh2p6WA...
rolvaliduntil |

```

Pour ouvrir les accès :

- ajuster `pg_hba.conf` pour permettre l'accès à la base **pgbench** uniquement à l'utilisateur **testperf**, permettre l'accès à la base **entreprise** à tout utilisateur, en local avec son mot de passe, en authentification `scram-sha-256` ;
- vérifier avec la vue `pg_hba_file_rules` ;
- recharger la configuration ;
- tester la connexion.

Ajouter ceci dans `pg_hba.conf`, **en tête** (nous verrons que c'est une erreur) :

#	TYPE	DATABASE	USER	ADDRESS	METHOD
	local	pgbench	testperf		scram-sha-256
	local	entreprise	all		scram-sha-256

Recharger la configuration et tenter une connexion depuis un compte utilisateur normal : cela doit fonctionner en entrant le mot de passe.

```

$ sudo -iu postgres psql -c 'SELECT pg_reload_conf()';
$ psql -U testperf -d pgbench
Mot de passe pour l'utilisateur testperf :
psql (15.1)
Type "help" for help.

```

pgbench=>

patron doit aussi pouvoir se connecter :

```
$ psql -U patron -d entreprise
```

Créer un fichier `.pgpass` dans votre répertoire utilisateur (`/home/dalibo`) ou dans celui de l'utilisateur *postgres* (`/var/lib/pgsql`) pour qu'il puisse se connecter aux bases **entreprise** et **pgbench** sans entrer le mot de passe.

Le fichier doit contenir le mot de passe en clair sous cette forme :

```
localhost:5432:pgbench:testperf:b0cdc36ff6c986b3930bfc269f37f3f2
localhost:5432:entreprise:patron:b0cdc36ff6c986b3930bfc269f37f3f2
```

NB : la mention `localhost` dans ce fichier couvre aussi bien les accès via `:::1` ou `127.0.0.1` (lignes `host` de `pg_hba.conf`) que les accès via la socket unix (lignes `local`).

Si le mot de passe est le même pour tous les utilisateurs créés par la suite, le nom d'utilisateur **patron** peut même être remplacé par `*`.

Si la connexion échoue, vérifier que le fichier est en mode 600 :

WARNING: password file "/home/dalibo/.pgpass" has group or world access; permissions
↪ should be u=rw (0600) or less

```
$ chmod u=rw,go= ~/.pgpass
```

La connexion doit à présent se faire sans mot de passe.

Le superutilisateur PostgreSQL **postgres** peut-il se connecter aux bases **entreprise** et **pgbench** ?
Si non, comment lui permettre ?

Même depuis l'utilisateur système **postgres**, la connexion aux deux bases que nous venons de créer échoue :

```
$ sudo -iu postgres psql -d entreprise -U postgres
Mot de passe pour l'utilisateur postgres :
psql: fe_sendauth: no password supplied
```

La cause est dans les traces :

```
FATAL: password authentication failed for user "postgres"
DETAIL: User "postgres" has no password assigned.
         Connection matched pg_hba.conf line 81:
         "local  entreprise  all          scram-sha-256"
```

L'échec est donc normal : la ligne de `pg_hba.conf` qui permet un accès incondtionnel à toute base depuis le compte système **postgres** est à présent la troisième. Pour corriger cela sans créer de mot de passe, la remplacer par cette **toute première ligne** de `pg_hba.conf` :

#	TYPE	DATABASE	USER	ADDRESS	METHOD
local	all		postgres		peer

Et recharger la configuration.

Depuis l'utilisateur système `postgres`, il ne doit plus y avoir de mot de passe demandé à la connexion :

```
$ psql -d entreprise
```

Remplir la base **pgbench** et générer un peu d'activité :

```
/usr/pgsql-15/bin/pgbench -i -s1 --foreign-keys pgbench -U testperf
/usr/pgsql-15/bin/pgbench -P1 -T3 pgbench -U testperf
```

Aucun mot de passe ne doit être demandé, avec ou sans `-h localhost`.

```
$ sudo -iu postgres
```

```
$ createuser --login --echo testperf
SELECT pg_catalog.set_config('search_path', '', false);
CREATE ROLE testperf NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

```
$ createdb --echo pgbench
SELECT pg_catalog.set_config('search_path', '', false);
CREATE DATABASE pgbench OWNER testperf;
```


Initialisation de la base (23 Mo) :

```
$ sudo -iu postgres
$ /usr/pgsql-15/bin/pgbench -i --foreign-keys -d pgbench -U testperf -h localhost
```

Génération de données :

```
/usr/pgsql-15/bin/pgbench -P1 -T3 pgbench -U testperf -h localhost
```

Le `pg_hba.conf` doit ressembler à celui-ci :

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

L'accès sans `-h localhost` fonctionne grâce aux lignes `local`. L'accès via `-h localhost` fonctionne grâce à l'une des lignes `host`. On avait vu qu'il ne manquait que le mot de passe. Évidemment cela ne marche pas depuis une adresse extérieure.

Les 3 dernières lignes sont réservées à des connexions de réplication.

Compléter `pg_ident.conf` pour pouvoir se connecter au rôle **postgres** depuis le compte système habituel (**dalibo**), en `local`.

Remplacer la première ligne de `pg_hba.conf` par :

```
local all postgres peer map=admins
```

Dans `pg_ident.conf`, ajouter :

#	MAPNAME	SYSTEM-USERNAME	PG-USERNAME
	admins	postgres	postgres
	admins	dalibo	postgres

On recharge :

```
$ sudo systemctl reload postgresql-15
```



Toute erreur peut empêcher la connexion au rôle **postgres** !

Tester la connexion depuis **postgres** au rôle **postgres**, de **dalibo** à **postgres**.

Ces connexions doivent fonctionner sans mot de passe :

```
postgres$ psql
postgres$ psql -d entreprise

dalibo$ psql -U postgres
dalibo$ psql -U postgres -d entreprise
```

1.12.5 Rôles et permissions



But : Gérer les rôles et les permissions sur les tables

Sous les utilisateurs **dalibo** comme **postgres**, créer un nouveau fichier `~/.psqlrc` contenant `\set PROMPT1 '%n@%/%R%# '` pour que l'invite indique quels sont les utilisateur et base en cours.

```
$ psql -U testperf -d pgbench
psql (15.1)
Type "help" for help.

testperf@pgbench=>
```

Noter que l'affichage de l'invite est légèrement différente selon que le type d'utilisateur : superutilisateur ou un utilisateur « normal ».

Ajouter à la base de données **entreprise** la table `facture (id int, objet text, creations timestamp)`. Elle appartiendra à **patron**, administrateur de la base.

Se connecter avec l'utilisateur **patron** (administrateur de la base **entreprise**) :

```
$ psql -U patron entreprise
```

Créer la table `facture` :

```
patron@entreprise=> CREATE TABLE facture (id int, objet text, creations timestamp);
```

Noter que la table appartient à celui qui la crée :

```
patron@entreprise=> \d

          Liste des relations
-----+-----+-----+-----
Schéma |  Nom  | Type | Propriétaire
-----+-----+-----+-----
public | facture | table | patron
```

Création d'un utilisateur et d'un groupe

Créer un rôle **secretariat** sans droit de connexion, mais qui peut visualiser, ajouter, mettre à jour et supprimer des éléments de la table `facture`.

Il faut le faire avec le rôle **postgres**, car **patron** n'a pas les droits :

```
patron@entreprise=# \c - postgres
You are now connected to database "entreprise" as user "postgres".

postgres@entreprise=# CREATE ROLE secretariat;
postgres@entreprise=# GRANT SELECT, INSERT, UPDATE, DELETE ON facture TO secretariat;
```

Créer un rôle **boulier** qui peut se connecter et appartient au rôle **secretariat**, avec un mot de passe (à ajouter au `.pgpass`).

```
postgres@entreprise=# CREATE ROLE boulier LOGIN IN ROLE SECRETARIAT;
postgres@entreprise=# \password boulier
```

Saisissez le nouveau mot de passe :
Saisissez-le à nouveau :

Vérifier la création des deux rôles.

```
postgres@entreprise=# \du
                Liste des rôles
  Nom du rôle |          Attributs          |  Membre de
-----+-----+-----
 boulier     |                               | {secretariat}
 dupont      |                               | {}
 patron      |                               | {}
 postgres    | Superutilisateur, Créer un rôle, ... | {}
 secretariat | Ne peut pas se connecter      | {}
 testperf    |                               | {}
```

En tant que **boulier**, créer une table `brouillon` identique à `facture`. Dans quel schéma cela se passe-t-il ? Avec PostgreSQL 15 ou supérieur, il y a une étape supplémentaire.

La connexion doit se faire sans problème avec le mot de passe.

```
$ psql -U boulier -d entreprise
```

Une astuce à connaître pour créer une table vide de même structure qu'une autre est :

```
boulier@entreprise=> CREATE TABLE brouillon (LIKE facture INCLUDING ALL) ;
```

Cet ordre fonctionnera directement jusque PostgreSQL 14 compris, car la table est créée implicitement dans le schéma **public**. À partir de PostgreSQL 15, par défaut, seul le propriétaire de la base (**patron**) peut écrire dans le schéma **public** :

```
ERROR: permission denied for schema public
LINE 1: CREATE table brouillon (like facture);
```

Il doit donc donner ce droit à **boulier** :

```
patron@entreprise=> GRANT CREATE ON schema public TO boulier ;
patron@entreprise=> \dn+ public
```

List of schemas			
Name	Owner	Access privileges	Descript...
public	pg_database_owner	pg_database_owner=UC/pg_database_owner+ =U/pg_database_owner	standard public
		boulier=C/pg_database_owner	schema

```
boulier@entreprise=> CREATE TABLE brouillon (LIKE facture INCLUDING ALL) ;
```

Vérifier les tables présentes et les droits `\dp`. Comment les lire ?

La nouvelle table appartient bien à **boulier** :

```
boulier@entreprise=> \dt
List of relations
Schema | Name | Type | Owner
-----+-----+-----+-----
public | brouillon | table | boulier
public | facture | table | patron
(2 rows)
```

```
boulier@entreprise=> \dp
```

Access privileges			
Schema	Name	Type	Access privileges
public	brouillon	table	...
public	facture	table	patron=arwdDxt/patron + secretariat=arwd/patron

Sans affectation explicite de droits, les droits par défauts ne figurent pas : par exemple, `brouillon` pourra être lu et modifié par son propriétaire, **boulier**.

patron a tous les droits sur la table `facture` (il possède la table).

On retrouve les droits donnés plus haut au rôle **secretariat** : insertion (`a` pour *append*), lecture (`r` pour *read*), modification (`w` pour *write*) et suppression (`d` pour *delete*).

On ne voit pas explicitement les droits de **boulier** (membre de **secretariat**) sur `facture`.

À l'aide du rôle **boulier** : insérer 2 factures ayant pour objet « Vin de Bordeaux » et « Vin de Bourgogne » avec la date et l'heure courante.

```
boulier@entreprise=> INSERT INTO facture VALUES
(1, 'Vin de Bordeaux', now()),
(2, 'Vin de Bourgogne', now());
```

Afficher le contenu de la table `facture`.

```
boulier@entreprise=> SELECT * FROM facture;
```

id	objet	creationts
1	Vin de Bordeaux	2019-07-16 17:50:28.942862
2	Vin de Bourgogne	2019-07-16 17:50:28.942862

Mettre à jour la deuxième facture avec la date et l'heure courante.

```
boulier@entreprise=> UPDATE facture SET creationts = now() WHERE id = 2 ;
UPDATE 1
```

Supprimer la première facture.

```
boulier@entreprise=> DELETE FROM facture WHERE id = 1 ;
DELETE 1
```

Modification des permissions

Retirer les droits `DELETE` sur la table `facture` au rôle `secretariat`.

```
boulier@entreprise=> \c - patron
```

Vous êtes maintenant connecté à la base de données « entreprise » en tant qu'utilisateur « patron ».

```
patron@entreprise=> REVOKE DELETE ON facture FROM secretariat;
REVOKE
```

Vérifier qu'il n'est plus possible de supprimer la deuxième facture avec le rôle `boulier`.

```
patron@entreprise=> \c - boulier
```

Vous êtes maintenant connecté à la base de données « entreprise » en tant qu'utilisateur « boulier ».

```
boulier@entreprise=> DELETE FROM facture WHERE id = 2;
ERROR: permission denied for table facture
```

En tant que `patron`, créer une table `produit` contenant une colonne texte nommée `appellation` et la remplir avec des noms de boissons.

```
boulier@entreprise=> \c - patron
```

Vous êtes maintenant connecté à la base de données « entreprise » en tant qu'utilisateur « patron ».

```
patron@entreprise=> CREATE TABLE produit (appellation text) ;
CREATE TABLE
```

```
patron@entreprise=> INSERT INTO produit VALUES
('Gewurtzraminer vendanges tardives'), ('Cognac'), ('Eau plate'),
('Eau gazeuse'), ('Jus de groseille') ;
```

```
INSERT 0 5
```

Afficher les droits sur cette table avec `\dt` et `\dp`. Vérifier que le rôle **boulier** appartenant au rôle **secretariat** ne peut pas sélectionner les produits contenus dans la table `produit`.

On voit bien que `produit` appartient à **patron** et que **secretariat** n'a à priori aucun droit dessus.

```
patron@entreprise=> \dt
```

```

      List of relations
 Schema | Name      | Type | Owner
-----+-----+-----+-----
 public | brouillon | table | boulier
 public | facture   | table | patron
 public | produit   | table | patron

```

```
patron@entreprise=> \dp
```

```

 Schema | Name      | Type | Access privileges
-----+-----+-----+-----
 public | brouillon | table |
 public | facture   | table | patron=arwdDxt/patron +
 public | produit   | table | secretariat=arw/patron

```

En conséquence, **boulier** ne peut lire la table :

```
patron@entreprise=> \c - boulier
```

Vous êtes maintenant connecté à la base de données « entreprise » en tant qu'utilisateur « boulier ».

```
boulier@entreprise=> SELECT * FROM produit;
ERROR: permission denied for table produit
```

Retirer tous les droits pour le groupe **secretariat** sur la table `produit`.

```
patron@entreprise=> REVOKE ALL ON produit from secretariat;
```

Que deviennent les droits affichés ? **boulier** peut-il lire la table ?

secretariat n'avait pourtant aucun droit, mais l'affichage a changé et énumère à présent explicitement les droits présents :

```
patron@entreprise=> \dp
```

```

 Schema | Name      | Type | Access privileges
-----+-----+-----+-----
 public | brouillon | table |
 public | facture   | table | patron=arwdDxt/patron +
 public | produit   | table | secretariat=arw/patron

```

Autoriser l'utilisateur **boulier** à accéder à la table `produit` en lecture.

```
patron@entreprise=> GRANT SELECT ON produit TO boulier ;
```

```
GRANT
```

Vérifier que **boulier** peut désormais accéder à la table `produit`.

```
boulier@entreprise=> SELECT * FROM produit ;
```

```

          appellation
-----
Gewurtzraminer vendanges tardives
Cognac
Eau plate
Eau gazeuse
Jus de groseille
(5 rows)

```

Héritage des droits au login

Créer un rôle **tina** appartenant au rôle **secretariat**, avec l'attribut `LOGIN`, mais n'héritant pas des droits à la connexion. Vérifier les droits avec `\du`. Lui donner un mot de passe.

La clause `NOINHERIT` évite qu'un rôle hérite immédiatement des droits des autres rôles :

```
postgres@entreprise=> CREATE ROLE tina LOGIN NOINHERIT ;
```

```
CREATE ROLE
```

```
postgres@entreprise=> GRANT secretariat TO tina;
```

```
postgres@entreprise=# \du
```

Role name	Attributes	List of roles	Member of
...			
tina	No inheritance		{secretariat}
...			

```
postgres@entreprise=# \password tina
```

Tester la connexion en tant que **tina**.

Vérifier que **tina** ne peut pas accéder à la table `facture`.

```
tina@entreprise=> SELECT * FROM facture;
```

```
ERROR: permission denied for table facture
```

En tant que **tina**, activer le rôle **secretariat** (`SET ROLE`).

```
tina@entreprise=> SET ROLE secretariat;
```

Vérifier que **tina** possède maintenant les droits du rôle **secretariat**. Sélectionner les données de la table `facture`.

L'utilisateur **tina** possède maintenant les droits du rôle **secretariat** :

```
tina@entreprise=> SELECT * FROM facture;
```

id	objet	creations
2	Vin de Bourgogne	2019-07-16 17:50:53.725971

1.12.6 Autorisation d'accès distant



But : Mettre en place les accès dans `pg_hba.conf`.

Autoriser tous les membres du réseau local à se connecter avec un mot de passe (autorisation en IP sans SSL) avec les utilisateurs **boulier** ou **tina**. Tester avec l'IP du serveur avant de demander aux voisins de tester.

Pour tester, repérer l'adresse IP du serveur avec `ip a`, par exemple `192.168.28.1`, avec un réseau local en `192.168.28.*`.

Ensuite, lors des appels à `psql`, utiliser `-h 192.168.28.1` au lieu de la connexion locale ou de **localhost** :

```
$ psql -h 192.168.123.180 -d entreprise -U tina
```

Ajouter les lignes suivantes dans le fichier `pg_hba.conf` :

```
host      entreprise    tina,boulier    192.168.28.0/24    scram-sha-256
```

Il ne faut pas oublier d'ouvrir PostgreSQL aux connexions extérieures dans `postgresql.conf` :

```
listen_addresses = '*'
```

Cette modification nécessite un redémarrage

Plus prudemment, on peut juste ajouter l'adresse publique de l'instance PostgreSQL :

```
listen_addresses = 'localhost,192.168.28.1'
```

Il y a peut-être un *firewall* à désactiver :

```
$ sudo systemctl status firewalld
$ sudo systemctl stop firewalld
```


1.12.7 VACUUM, VACUUM FULL, DELETE, TRUNCATE



But : Effacer des données, distinguer `VACUUM` et `VACUUM FULL` .

Pré-requis

Désactiver le démon autovacuum de l'instance.

Dans le fichier `postgresql.conf` , désactiver le démon autovacuum en modifiant le paramètre suivant :

```
autovacuum = off
```



Ne jamais faire cela en production !

On recharge la configuration :

```
$ psql -c 'SELECT pg_reload_conf()'
```

On vérifie que le paramètre a bien été modifié :

```
postgres@postgres=# show autovacuum ;
```

```
autovacuum
-----
off
```

Nettoyage avec VACUUM

Se connecter à la base `pgbench` en tant que `testperf`.

```
$ psql -U testperf -d pgbench
```

Grâce aux fonctions `pg_relation_size` et `pg_size_pretty` , afficher la taille de la table `pgbench_accounts` .

`\d+` affiche les tailles des tables, mais il existe des fonctions plus ciblées.

Pour visualiser la taille de la table, il suffit d'utiliser la fonction `pg_relation_size` . Comme l'affichage a parfois trop de chiffres pour être facilement lisible, on utilise `pg_size_pretty` .



Il est facile de retrouver facilement une fonction en effectuant une recherche par mot clé dans `psql`, notamment pour retrouver ses paramètres. Exemple :

```
postgres=# \df *pretty*
```

```
Liste des fonctions
```

```
-[ RECORD 1 ]-----+-----
Schéma                | pg_catalog
Nom                   | pg_size_pretty
Type de données du résultat | text
Type de données des paramètres | bigint
Type                  | normal
```

Cela donne au final :

```
testperf@pgbench=> SELECT pg_relation_size('pgbench_accounts');
```

```
pg_relation_size
-----
13434880
```

```
testperf@pgbench=> SELECT pg_size_pretty(pg_relation_size('pgbench_accounts'));
```

```
pg_size_pretty
-----
13 MB
```

Copier le contenu de la table dans une nouvelle table (`pba_copie`).

```
testperf@pgbench=> CREATE table pba_copie AS SELECT * FROM pgbench_accounts;
SELECT 100000
```

Supprimer le contenu de la table `pba_copie`, à l'exception de la dernière ligne (`aid=100000`), avec un ordre `DELETE`. Quel est alors l'espace disque utilisé par cette table ?

```
testperf@pgbench=> DELETE FROM pba_copie WHERE aid <100000;
DELETE 99999
```

Il ne reste qu'une ligne, mais l'espace disque est toujours utilisé :

```
testperf@pgbench=> SELECT pg_size_pretty(pg_relation_size('pba_copie'));
```

```
pg_size_pretty
-----
13 MB
```

Noter que même si l'autovacuum n'était pas désactivé, il n'aurait pas réduit l'espace occupé par la table car il reste la ligne à la fin de celle-ci. De plus, il n'aurait pas eu forcément le temps de passer sur la table entre les deux ordres précédents.

Insérer le contenu de la table `pgbench_accounts` dans la table `pba_copie`. Quel est alors l'espace disque utilisé par la table ?

```
testperf@pgbench=> INSERT into pba_copie SELECT * FROM pgbench_accounts;
INSERT 0 100000
```

L'espace disque utilisé a doublé :

```
testperf@pgbench=> SELECT pg_size_pretty(pg_relation_size('pba_copie'));
 pg_size_pretty
-----
26 MB
```

Les nouvelles données se sont ajoutées à la fin de la table. Elles n'ont pas pris la place des données effacées précédemment.

Effectuer un `VACUUM` simple sur `pba_copie`. Vérifier la taille de la base.

La commande vacuum « nettoie » mais ne libère pas d'espace disque :

```
testperf@pgbench=> VACUUM pba_copie;
VACUUM
testperf@pgbench=> SELECT pg_size_pretty(pg_relation_size('pba_copie'));
 pg_size_pretty
-----
26 MB
```

Vider à nouveau la table `pba_copie` des lignes d'`aid` inférieur à 100 000. Insérer à nouveau le contenu de la table `pgbench_accounts`. L'espace mis à disposition a-t-il été utilisé ?

```
testperf@pgbench=> DELETE FROM pba_copie WHERE aid <100000;
DELETE 99999
testperf@pgbench=> INSERT into pba_copie SELECT * FROM pgbench_accounts;
INSERT 0 100000
testperf@pgbench=> SELECT pg_size_pretty(pg_relation_size('pba_copie'));
 pg_size_pretty
-----
26 MB
```

Cette fois, la table n'a pas augmenté de taille. PostgreSQL a pu réutiliser la place des lignes effacées que `VACUUM` a marqué comme disponibles.

Voir la taille de la base. Supprimer la table `pba_copie`. Voir l'impact sur la taille de la base.

Nous verrons plus tard comment récupérer de l'espace. Pour le moment, on se contente de supprimer la table.

```
postgres@pgbench=# SELECT pg_size_pretty(pg_database_size ('pgbench')) ;
```

```
pg_size_pretty
-----
49 MB
```

```
postgres@pgbench=# DROP TABLE pba_copie ;
```

```
DROP TABLE
```

```
postgres@pgbench=# SELECT pg_size_pretty(pg_database_size ('pgbench')) ;
```

```
pg_size_pretty
-----
23 MB
```

Supprimer une table rend immédiatement l'espace disque au système.

VACUUM avec les requêtes de pgbench

Tout d'abord, repérer les tailles des différentes tables et le nombre de lignes de chacune.

```
postgres@pgbench=#d+
```

Schéma	Nom	Liste des relations			Taille	Description
		Type	Propriétaire			
public	pgbench_accounts	table	testperf		13 MB	
public	pgbench_branches	table	testperf		40 kB	
public	pgbench_history	table	testperf		0 bytes	
public	pgbench_tellers	table	testperf		40 kB	

```
testperf@pgbench=> SELECT count(*) FROM pgbench_accounts;
```

```
count
-----
100000
```

```
testperf@pgbench=> SELECT count(*) FROM pgbench_tellers;
```

```
count
-----
10
```

```
testperf@pgbench=> SELECT count(*) FROM pgbench_branches;
```

```
count
-----
1
```

```
testperf@pgbench=> SELECT count(*) FROM pgbench_history;
```

```
count
```

```
-----
0
```

(Le contenu de cette dernière table dépend de l'historique de la base.)

Pour amplifier le phénomène à observer, on peut créer une session de très longue durée, laissée ouverte sans `COMMIT` ni `ROLLBACK`. Il faut qu'elle ait consulté une des tables pour que l'effet soit visible :

```
testperf@pgbench=> BEGIN ;

BEGIN
Temps : 0,608 ms

testperf@pgbench=> SELECT count(*) FROM pgbench_accounts ;

count
-----
100000
(1 ligne)

Temps : 26,059 ms
testperf@pgbench=> SELECT pg_sleep (10000) ;
```

Depuis un autre terminal, générer de l'activité sur la table, ici avec 10 000 transactions sur 20 clients :

```
PGOPTIONS='-c synchronous_commit=off' \
/usr/pgsql-15/bin/pgbench -U testperf -d pgbench \
--client=20 --jobs=2 -t 10000 --no-vacuum
```

(NB : La variable d'environnement `PGOPTIONS` restreint l'utilisation des journaux de transaction pour accélérer les écritures (données NON critiques ici). Le `--no-vacuum` est destiné à éviter que l'outil demande lui-même un `VACUUM`. Le test dure quelques minutes. Le relancer au besoin.)

Après quelques minutes, `pgbench` affichera le nombre de transactions par seconde, bien sûr très dépendant de la machine :

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 20
number of threads: 2
number of transactions per client: 10000
number of transactions actually processed: 200000/200000
latency average = 58.882 ms
tps = 339.663189 (including connections establishing)
tps = 339.664978 (excluding connections establishing)
```

(Optionnel) C'est l'occasion d'installer l'outil `pg_activity` depuis les dépôts du PGDG (il peut y avoir besoin du dépôt EPEL) et de le lancer en tant que **postgres** pour voir ce qui se passe dans la base.

Pour `pg_activity` :

```
$ sudo yum install epel-release
$ sudo yum install pg_activity
```

Il se lance ainsi :

```
$ sudo -iu postgres pg_activity
```

Le premier écran affiche les sessions en cours, le deuxième celles en attente de libération d'un verrou, le troisième celles qui en bloquent d'autres.

Noter que la session restée délibérément ouverte n'est pas bloquante.

Comparer les nouvelles tailles des tables (taille sur le disque et nombre de lignes). La table `pg_stat_user_tables` contient l'activité sur chaque table. Comment s'expliquent les évolutions ?

La volumétrie des tables a explosé :

```
testperf@pgbench=> \d+
```

Schéma	Nom	Type	Propriétaire	Taille	Description
public	pgbench_accounts	table	testperf	39 MB	
public	pgbench_branches	table	testperf	7112 kB	
public	pgbench_history	table	testperf	10 MB	
public	pgbench_tellers	table	testperf	8728 kB	

On constate que le nombre de lignes reste le même malgré l'activité, sauf pour la table d'historique :

```
testperf@pgbench=> SELECT count(*) FROM pgbench_accounts;
```

```
count
-----
100000
```

```
testperf@pgbench=> SELECT count(*) FROM pgbench_tellers;
```

```
count
-----
10
```

```
testperf@pgbench=> SELECT count(*) FROM pgbench_branches;
```

```
count
-----
1
```

```
testperf@pgbench=> SELECT count(*) FROM pgbench_history;
```

```
count
-----
200000
```

Ce dernier chiffre dépend de l'activité réelle et du nombre de requêtes.

Les statistiques d'activité de la table sont dans `pg_stat_user_tables`. Pour `pgbench_accounts`, la plus grosse table, on y trouve ceci :

```
testperf@pgbench=> SELECT * FROM pg_stat_user_tables ORDER BY relname \gx
```

```
-[ RECORD 1 ]-----+-----
reloid          | 17487
schemaname      | public
relname         | pgbench_accounts
seq_scan        | 6
seq_tup_read    | 300000
idx_scan        | 600000
idx_tup_fetch   | 600000
n_tup_ins       | 100000
n_tup_upd       | 200000
n_tup_del       | 0
n_tup_hot_upd   | 1120
n_live_tup      | 100000
n_dead_tup      | 200000
n_mod_since_analyze | 300000
last_vacuum     | 2021-09-04 18:51:31.889825+02
last_autovacuum | x
last_analyze    | 2021-09-04 18:51:31.927611+02
last_autoanalyze | x
vacuum_count    | 1
autovacuum_count | 0
analyze_count   | 1
autoanalyze_count | 0
```

Le champ `n_tup_upd` montre qu'il y a eu 200 000 mises à jour après l'insertion initiale de 100 000 lignes (champ `n_tup_ins`). Il y a toujours 100 000 lignes visibles (`n_live_tup`).

Le `VACUUM` a été demandé explicitement à la création (`last_vacuum`) mais n'est pas passé depuis.

Le `VACUUM` étant inhibé, il est normal que les lignes mortes se soient accumulées (`n_dead_tup`) : il y en a 200 000, ce sont les anciennes versions des lignes modifiées.

Pour la table `pgbench_history` :

```
-[ RECORD 3 ]-----+-----
reloid          | 17481
schemaname      | public
relname         | pgbench_history
seq_scan        | 4
seq_tup_read    | 200000
idx_scan        | x
idx_tup_fetch   | x
n_tup_ins       | 200000
n_tup_upd       | 0
n_tup_del       | 0
```

```

n_tup_hot_upd      | 0
n_live_tup         | 200000
n_dead_tup         | 0
...

```

La table `pgbench_history` a subi 200 000 insertions et contient à présent 200 000 lignes : il est normal qu'elle ait grossi de 0 à 10 Mo.

Pour la table `pgbench_tellers` :

```

-[ RECORD 4 ]-----+-----
...
relname          | pgbench_tellers
seq_scan         | 20383
seq_tup_read     | 117437
idx_scan         | 379620
idx_tup_fetch    | 379620
n_tup_ins        | 10
n_tup_upd        | 200000
...
n_live_tup       | 10
n_dead_tup       | 199979
n_mod_since_analyze | 200010
...

```

Elle ne contient toujours que 10 lignes visibles (`n_live_tup`), mais 199 979 lignes « mortes » (`n_dead_tup`).

Même s'il n'a généré aucune opération du point de vue de l'utilisateur, le verrou posé par la session en attente est visible dans la table des verrous `pg_locks` :

```

postgres@pgbench=# SELECT * FROM pg_locks
  WHERE relation = (SELECT relid FROM pg_stat_user_tables
  WHERE relname = 'pgbench_accounts' ) ;

```

```

-[ RECORD 1 ]-----+-----
locktype          | relation
database          | 16729
relation          | 17487
page              | x
tuple             | x
virtualxid        | x
transactionid     | x
classid           | x
objid             | x
objsubid          | x
virtualtransaction | 1/37748
pid               | 22581
mode              | AccessShareLock
granted           | t
fastpath          | t
waitstart         | 2021-09-04 19:01:27.824567+02

```

Nettoyage avec VACUUM FULL

Exécuter un `VACUUM FULL VERBOSE` sur `pgbench_tellers`.

```
postgres@pgbench=# VACUUM FULL VERBOSE pgbench_tellers ;  
  
INFO: vacuuming "public.pgbench_tellers"  
INFO: "pgbench_tellers": found 200000 removable, 10 nonremovable row versions in  
  ↪ 1082 pages  
DÉTAIL : 0 dead row versions cannot be removed yet.  
CPU: user: 0.03 s, system: 0.00 s, elapsed: 0.03 s.  
VACUUM
```

Un `\d+` indique que la taille de la table est bien retombée à 8 ko (1 bloc), ce qui suffit pour 10 lignes.

Exécuter un `VACUUM FULL VERBOSE` sur `pgbench_accounts`.

Si celui-ci reste bloqué, il faudra sans doute arrêter la transaction restée ouverte plus haut.

```
postgres@pgbench=# VACUUM FULL VERBOSE pgbench_accounts ;  
  
INFO: vacuuming "public.pgbench_accounts"  
INFO: "pgbench_accounts": found 200000 removable,100000 nonremovable row versions  
  in 4925 pages  
DÉTAIL : 0 dead row versions cannot be removed yet.  
CPU: user: 0.09 s, system: 0.06 s, elapsed: 0.17 s.  
VACUUM  
Durée : 16411,719 ms (00:16,412)
```

Soit : 100 000 lignes conservées, 200 000 mortes supprimées dans 4925 blocs (39 Mo).

Effectuer un `VACUUM FULL VERBOSE`. Quel est l'impact sur la taille de la base ?

Même les tables système seront nettoyées :

```
postgres@pgbench=> VACUUM FULL VERBOSE ;  
  
INFO: vacuuming "pg_catalog.pg_statistic"  
INFO: "pg_statistic": found 11 removable, 433 nonremovable row versions in 20 pages  
DÉTAIL : 0 dead row versions cannot be removed yet.  
...  
INFO: vacuuming "public.pgbench_branches"  
INFO: "pgbench_branches": found 200000 removable, 1 nonremovable row versions in  
  ↪ 885 pages  
DÉTAIL : 0 dead row versions cannot be removed yet.  
CPU: user: 0.03 s, system: 0.00 s, elapsed: 0.03 s.  
INFO: vacuuming "public.pgbench_history"  
INFO: "pgbench_history": found 0 removable, 200000 nonremovable row versions in  
  ↪ 1281 pages  
DÉTAIL : 0 dead row versions cannot be removed yet.  
CPU: user: 0.11 s, system: 0.02 s, elapsed: 0.13 s.  
INFO: vacuuming "public.pgbench_tellers"  
INFO: "pgbench_tellers": found 0 removable, 10 nonremovable row versions in 1 pages  
DÉTAIL : 0 dead row versions cannot be removed yet.  
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
```

```
INFO: vacuuming "public.pgbench_accounts"
INFO: "pgbench_accounts": found 0 removable, 100000 nonremovable row versions in
↳ 1640 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 0.03 s, system: 0.01 s, elapsed: 0.05 s.
VACUUM
```

Seule `pgbench_branches` était encore à nettoyer (1 ligne conservée).

La taille de la base retombe à 32 Mo selon `\l+`. Elle faisait au départ 22 Mo, et 10 Mo ont été ajoutés dans `pgbench_history`.

Truncate ou Delete ?

Créer `copie1` et `copie2` comme des copies de `pgbench_accounts`, données incluses.

```
postgres@pgbench=# CREATE TABLE copie1 AS SELECT * FROM pgbench_accounts ;
SELECT 100000
```

```
postgres@pgbench=# CREATE TABLE copie2 AS SELECT * FROM pgbench_accounts ;
SELECT 100000
```

Effacer le contenu de `copie1` avec `DELETE`.

```
postgres@pgbench=# DELETE FROM copie1 ;
DELETE 100000
```

Effacer le contenu de `copie2` avec `TRUNCATE`.

```
postgres@pgbench=# TRUNCATE copie2 ;
TRUNCATE TABLE
```

Quelles sont les tailles de ces deux tables après ces opérations ?

```
postgres@pgbench=# \d+
```

Schéma	Nom	Liste des relations			
		Type	Propriétaire	Taille	Description
public	copie1	table	postgres	13 MB	
public	copie2	table	postgres	0 bytes	
...					

Pour une purge complète, `TRUNCATE` est à préférer : il vide la table et rend l'espace au système. `DELETE` efface les lignes mais l'espace n'est pas encore rendu.

1.12.8 Réactivation de l'autovacuum

Réactiver l'autovacuum de l'instance.

Dans `postgresql.conf` :

```
autovacuum = on
```

```
postgres@pgbench=# SELECT pg_reload_conf() ;
```

```
pg_reload_conf
-----
t
```

```
postgres@pgbench=# SHOW autovacuum;
```

```
autovacuum
-----
on
```

Attendre quelques secondes et voir si `copie1` change de taille.

Après quelques instants, la taille de `copie1` (qui avait été vidée plus tôt avec `DELETE`) va redescendre à quelques kilooctets.

Le passage de l'autovacuum en arrière-plan est tracé dans `last_autovacuum` :

```
postgres@pgbench=# SELECT * FROM pg_stat_user_tables WHERE relname = 'copie1' \gx
```

```
-[ RECORD 1 ]-----+-----
reloid          | 18920
schemaname      | public
relname         | copie1
seq_scan        | 1
seq_tup_read    | 100000
idx_scan        |
idx_tup_fetch   |
n_tup_ins       | 100000
n_tup_upd       | 0
n_tup_del       | 100000
n_tup_hot_upd   | 0
n_live_tup      | 0
n_dead_tup      | 0
n_mod_since_analyze | 0
last_vacuum     |
last_autovacuum | 2019-07-17 14:04:21.238296+01
last_analyze    |
last_autoanalyze | 2019-07-17 14:04:21.240525+01
vacuum_count    | 0
autovacuum_count | 1
analyze_count   | 0
autoanalyze_count | 1
```

1.12.9 Statistiques des données



But : Savoir trouver les statistiques des données et les mettre à jour

Créer une table `copie3`, copie de `pgbench_accounts`.

```
CREATE TABLE copie3 AS SELECT * FROM pgbench_accounts ;
SELECT 100000
```

Dans la vue système `pg_stats`, afficher les statistiques collectées pour la table `copie3`.

```
postgres@pgbench=# SELECT * FROM pg_stats WHERE tablename = 'copie3' ;
(0 ligne)
```

L'autovacuum n'est pas passé, les statistiques ne sont pas encore présentes. Noter que, même activé, il n'aurait pas forcément eu le temps de passer entre les deux ordres précédents.

Lancer la collecte des statistiques pour cette table uniquement.

La collecte se déclenche avec la commande `ANALYZE` :

```
postgres@pgbench=# ANALYZE VERBOSE copie3 ;
INFO:  analyzing "public.copie3"
INFO:  "copie3": scanned 1640 of 1640 pages,
        containing 100000 live rows and 0 dead rows;
        30000 rows in sample, 100000 estimated total rows
ANALYZE
```

30 000 lignes font partie de l'échantillonnage.

Afficher de nouveau les statistiques.

```
SELECT * FROM pg_stats WHERE tablename = 'copie3' ;
```

Cette fois, la vue `pg_stats` renvoie des informations, colonne par colonne.

Le champ `aid` est la clé primaire, ses valeurs sont toutes différentes. L'histogramme des valeurs compte 100 valeurs qui délimite autant de *buckets*. Ils sont là régulièrement répartis, ce qui indique une répartition homogène des valeurs.

```
SELECT * FROM pg_stats WHERE tablename = 'copie3' ;
```

```
-[ RECORD 1 ]-----
schemaname      | public
tablename       | copie3
```

```

attname          | aid
inherited        | f
null_frac        | 0
avg_width        | 4
n_distinct       | -1
most_common_vals |
most_common_freqs |
histogram_bounds | {2,1021,2095,3098,4058,5047,6120,
7113,8058,9075,10092,11090,12061,13064,14053,15091,16106,
17195,18234,19203,20204,21165,22183,23156,24162,25156,26192,
27113,28159,29193,30258,31260,32274,33316,34346,35350,36281,
37183,38158,39077,40007,41070,42084,43063,44064,45101,46089,
47131,48189,49082,50100,51157,52113,53009,54033,55120,56114,
57066,58121,59111,60122,61088,62151,63217,64195,65168,66103,
67088,68126,69100,70057,71104,72105,73092,73994,75007,76067,
77092,78141,79180,80165,81100,82085,83094,84107,85200,86242,
87246,88293,89288,90286,91210,92197,93172,94084,95070,96086,
97067,98031,99032,99998}
correlation      | 1
most_common_elems |
most_common_elem_freqs |
elem_count_histogram |

```

Autre exemple, le champ `bid` : on voit qu'il ne possède qu'une seule valeur.

```

-[ RECORD 2 ]-----
schemaname      | public
tablename       | copie3
attname         | bid
inherited       | f
null_frac       | 0
avg_width       | 4
n_distinct      | 1
most_common_vals | {1}
most_common_freqs | {1}
histogram_bounds |
correlation     | 1
most_common_elems |
most_common_elem_freqs |
elem_count_histogram |

```

De même, on pourra vérifier que le champ `filler` a une taille moyenne de 85 octets, ou voir la répartition des valeurs du champ `abalance`.

1.12.10 Réindexation



But : Réindexer

Recréer les index de la table `pgbench_accounts`.

La réindexation d'une table se fait de la manière suivante :

```
postgres@pgbench=# REINDEX (VERBOSE) TABLE pgbench_accounts ;
INFO:  index "pgbench_accounts_pkey" was reindexed
DÉTAIL : CPU: user: 0.05 s, system: 0.00 s, elapsed: 0.08 s
REINDEX
```

Comment recréer tous les index de la base `pgbench` ?

```
postgres@pgbench=# REINDEX (VERBOSE) DATABASE pgbench ;
...
INFO:  index "pg_shseclabel_object_index" was reindexed
DÉTAIL : CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.01 s
INFO:  index "pg_toast_3592_index" was reindexed
DÉTAIL : CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.01 s
INFO:  table "pg_catalog.pg_shseclabel" was reindexed
INFO:  index "pgbench_branches_pkey" was reindexed
DÉTAIL : CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.01 s
INFO:  table "public.pgbench_branches" was reindexed
INFO:  index "pgbench_tellers_pkey" was reindexed
DÉTAIL : CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.01 s
INFO:  table "public.pgbench_tellers" was reindexed
INFO:  index "pgbench_accounts_pkey" was reindexed
DÉTAIL : CPU: user: 0.07 s, system: 0.01 s, elapsed: 0.12 s
INFO:  table "public.pgbench_accounts" was reindexed
REINDEX
```

Comment recréer uniquement les index des tables systèmes ?

Pour réindexer uniquement les tables systèmes :

```
postgres@pgbench=# REINDEX SYSTEM pgbench ;
```

Quelle est la différence entre la commande `REINDEX` et la séquence `DROP INDEX` + `CREATE INDEX` ?

`REINDEX` est similaire à une suppression et à une nouvelle création de l'index. Cependant, les conditions de verrouillage sont différentes :

- `REINDEX` verrouille les écritures mais pas les lectures de la table mère de l'index. Il prend aussi un verrou exclusif sur l'index en cours de traitement, ce qui bloque les lectures qui tentent d'utiliser l'index.
- Au contraire, `DROP INDEX` crée temporairement un verrou exclusif sur la table parent, bloquant ainsi écritures et lectures. Le `CREATE INDEX` qui suit verrouille les écritures mais pas les lectures ; comme l'index n'existe pas, aucune lecture ne peut être tentée, signifiant qu'il n'y a aucun blocage et que les lectures sont probablement forcées de réaliser des parcours séquentiels complets.

1.12.11 Traces



But : Gérer les fichiers de traces

Quelle est la méthode de gestion des traces utilisée par défaut ?

Par défaut, le mode de journalisation est **stderr** :

```
postgres@pgbench=# SHOW log_destination ;
log_destination
-----
stderr
```

Paramétrer le programme interne de rotation des journaux :

- modifier le fichier `postgresql.conf` pour utiliser le *logging collector* ;
- les traces doivent désormais être sauvegardés dans le répertoire `/var/lib/pgsql/traces` ;
- la rotation des journaux doit être automatisée pour générer un nouveau fichier de logs toutes les 30 minutes, quelle que soit la quantité de logs archivés ; le nom du fichier devra donc comporter les minutes.
- Tester en forçant des rotations avec la fonction `pg_rotate_logfile`.
- Augmenter la trace (niveau `info`).

Sur Red Hat/CentOS/Rocky Linux, le collecteur des traces (*logging collector*) est activé par défaut dans `postgresql.conf` (mais ce ne sera pas le cas sur un environnement Debian ou avec une installation compilée, et il faudra redémarrer pour l'activer) :

```
logging_collector = on
```

On crée le répertoire, où **postgres** doit pouvoir écrire :

```
$ sudo mkdir -m700 /var/lib/pgsql/traces
$ sudo chown postgres: /var/lib/pgsql/traces
```

Puis paramétrer le comportement du récupérateur :

```
log_directory = '/var/lib/pgsql/traces'
log_filename = 'postgresql-%Y-%m-%d_%H-%M.log'
log_rotation_age = 30min
log_rotation_size = 0
log_min_messages = info
```

Recharger la configuration et voir ce qui se passe dans `/var/lib/pgsql/traces` :

```
$ sudo systemctl reload postgresql-12
$ sudo watch -n 5 ls -lh /var/lib/pgsql/traces
```

Dans une autre fenêtre, générer un peu d'activité, avec `pgbench` ou tout simplement avec :

```
postgres@pgbench=# SELECT 1 ;  
postgres@pgbench=# \watch 1
```

Les fichiers générés doivent porter un nom ressemblant à `postgresql-2019-08-02_16-55.log`.

Pour forcer le passage à un nouveau fichier de traces :

```
postgres@pgbench=# SELECT pg_rotate_logfile() ;
```

Comment éviter l'accumulation des fichiers ?

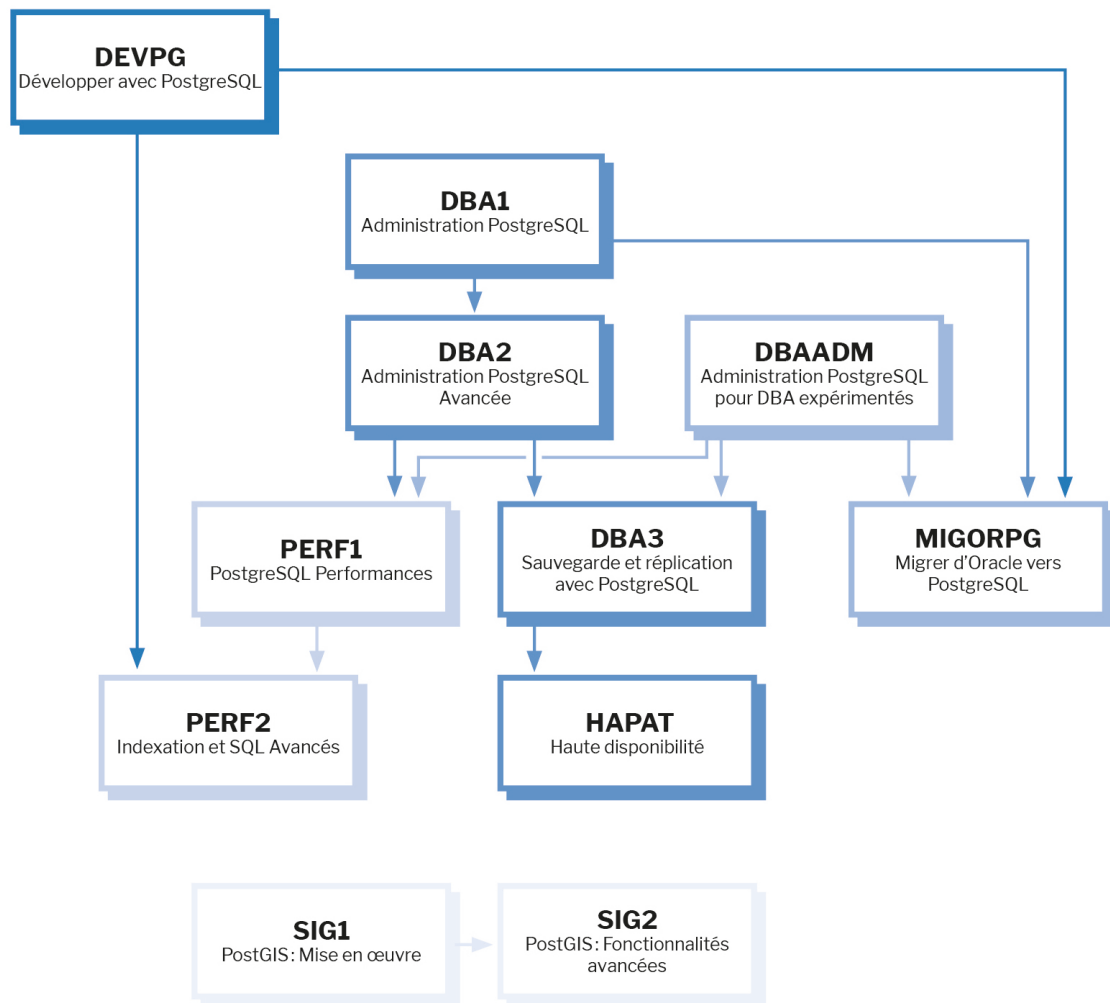
- La première méthode consiste à avoir un `log_filename` cyclique. C'est le cas par défaut sur Red Hat/CentOS/Rocky Linux avec `postgresql-%a`, qui reprend les jours de la semaine. Noter qu'il n'est pas forcément garanti qu'un `postgresql-%H-%M.log` planifié toutes les 5 minutes écrase les fichiers de la journée précédente. En pratique, on descend rarement en-dessous de l'heure.
- Utiliser l'utilitaire `logrotate`, fourni avec toute distribution Linux, dont le travail est de gérer rotation, compression et purge. Il est activé par défaut sur Debian.
- Enfin, on peut rediriger les traces vers un système externe.

Les formations Dalibo

Retrouvez nos formations et le calendrier sur <https://dali.bo/formation>

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

Cursus des formations



Retrouvez nos formations dans leur dernière version :

- 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>

Les livres blancs

- Migrer d'Oracle à PostgreSQL
<https://dali.bo/dlb01>
- Industrialiser PostgreSQL
<https://dali.bo/dlb02>
- Bonnes pratiques de modélisation avec PostgreSQL
<https://dali.bo/dlb04>
- Bonnes pratiques de développement avec PostgreSQL
<https://dali.bo/dlb05>

Téléchargement gratuit

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.

