PostgreSQL par MICHAEL et JABEUR

1.Historique et présentation de PostgreSQL

PostgreSQL est un système de gestion de base de données relationnelle et objet (SGBDRO). C'est un outil libre disponible selon les termes d'une licence de type BSD. Ce système est concurrent d'autres systèmes de gestion de base de données, qu'ils soient libres (comme MariaDB, MySQL et Firebird), ou propriétaires (comme Oracle, Sybase, DB2, Informix et Microsoft SQL Server). Comme les projets libres Apache et Linux, PostgreSQL n'est pas contrôlé par une seule entreprise, mais est fondé sur une communauté mondiale de développeurs et d'entreprises. L'histoire de PostgreSQL remonte à la base de données Ingres, développée à Berkeley par Michael Stonebraker. Lorsque ce dernier décida en 1985 de recommencer le développement de zéro, il nomma le logiciel Postgres, comme raccourci de post-Ingres. Lors de l'ajout des fonctionnalités SQL en 1995, Postgres fut renommé Postgres95. Ce nom fut changé à la fin de 1996 en PostgreSQL.

Le projet s'organise de manière à maintenir simultanément plusieurs versions stables ainsi qu'un dépôt mis à jour en continue. Cette pratique est extrêmement appréciable pour les productions des systèmes informatiques car elle leur permet de lisser les besoins de migration obligée de leurs applications qui utilisent des bases de données sur les moments de faible charge et de disponibilités de leurs ressources. Elles réalisent ainsi de très sérieuses économies, puisqu'on estime en moyenne que cette charge représente au moins 40%, mais le plus souvent jusqu'à 60% des coûts de fonctionnement des organisations informatiques au sein des entreprises. Ce SGBDRO utilise des types de données modernes, dit composés ou enrichis suivant les terminologies utilisées dans le vocable informatique usuel. Ceci signifie que PostgreSQL peut stocker plus de types de données que les types simples traditionnels entiers, caractères, etc. L'utilisateur peut créer des types, des fonctions, utiliser l'héritage de type, etc.

PostgreSQL est largement reconnu pour son comportement stable, proche de Oracle. Mais aussi pour ses possibilités de programmation étendues, directement dans le moteur de la base de données, via PL/pgSQL. Le traitement interne des données peut aussi être couplé à d'autres modules externes compilés dans d'autres langages.

Dans le jargon des bases de données, PostgreSQL™ utilise un modèle client/serveur. Une session PostgreSQL™ est le résultat de la coopération des processus suivants :

1. Un processus serveur qui gère les fichiers de la base de données, accepte les connexions à la base de la part des applications clientes et effectue sur la base les actions des clients. Le programme serveur est appelé postgres.
2. L'application cliente qui veut effectuer des opérations sur la base de données. Les applications clientes peuvent être de nature très différentes : un client peut être un outil texte, une application graphique, un serveur web qui accède à la base de données ou un outil spécialisé dans la maintenance de bases de données. Certaines applications clientes sont fournies avec PostgreSQL™.

Comme souvent avec les applications client/serveur, le client et le serveur peuvent être sur des hôtes différents. Dans ce cas, ils communiquent à travers une connexion réseau TCP/IP. Vous devez garder cela à l'esprit car les fichiers qui sont accessibles sur la machine cliente peuvent ne pas l'être (ou l'être seulement en utilisant des noms de fichiers différents) sur la machine exécutant le serveur de bases de données. Le serveur PostgreSQL™ peut traiter de multiples connexions simultanées depuis les clients. Dans ce but, il démarre un nouveau processus pour chaque connexion. À ce moment, le client et le nouveau processus serveur communiquent sans intervention de la part du processus postgres original. Ainsi, le processus serveur maître s'exécute toujours, attendant de nouvelles connexions clientes, tandis que le client et les processus serveurs associés vont et viennent.

2.Avantages et Inconvénients par rapport à MySQL/MariaDB

Bien que PostgreSQL soit plus avancé technologiquement que MySQL, il a une lacune en vitesse sur les faibles volumes de données. Ainsi MySQL, par le fait qu'il ne gère pas l'intégrité référentielle par exemple, se révèle plus rapide que PostgreSQL puisqu'il ne doit pas faire les tests d'intégrité (qui permettent de vérifier qu'une base de données est cohérente pour rappel).

MySQL est jeune, bien plus que PostgreSQL, mais il a connu un développement plus qu'honorable et supporte depuis sa version 5 beaucoup d'aspects du langage SQL. Je pense en particulier aux vues, triggers et UDF qui ont été rajoutés dans cette version, ainsi qu'aux sous-requêtes implémentées depuis la version 4.1.

Malgré tout, PostgreSQL garde l'avantage avec une panoplie beaucoup plus étendue: il gère en plus les règles, les types utilisateur, les tableaux, des langages procéduraux tels que PHP, Python, Java et bien d'autres. PostgreSQL jouit également d'un système d'extensions vraiment utile et d'autres aspects moins connus mais bien pratique de la norme tels que les règles qui permettent d'insérer des données depuis une vue par exemple, les séquences (équivalent avancé de l'auto incrément de MySQL), les domaines (types sur lesquels on peut apposer une contrainte et réutilisables), ainsi que bien d'autres.

MySQL s'occupe avec excellence des petits volumes de données, avec un faible nombre d'utilisateurs. En effet, dans ces cas là l'intégrité référentielle n'est souvent pas nécessaire puisqu'il est rare que plusieurs utilisateurs agissent simultanément sur la même table et encore plus sur le même enregistrement. La vitesse d'exécution des requêtes s'en retrouve accélérée, ce qui explique les résultats incomparables de MySQL sur ce genre de base de données. PostgreSQL par contre peut gérer les très gros volumes de données. Des bases de 13 teras existent et tournent parfaitement sous PostgreSQL. Son optimiseur fait des merveilles à partir du moment où il y a suffisamment de données pour qu'il soit efficace.

Gérer une base de données n'est pas une chose simple, surtout si on veut bien le faire. MySQL a permis à tout un chacun de s'essayer dans ce domaine, mais pour moi ce n'est pas réellement un SGBDR. Pour ceux qui ont connu ce moment, vous avez connu également la différence considérable de performances : MySQL n'était tout simplement plus capable de gérer une base de données devenue trop grosse et trop complexe.

<note important>En conclusion, il faut utiliser l'outil adéquat pour chaque projet :

1. MySQL pour les petites bases de données avec un nombre faible d'accès.
2. PostGreSQL pour les bases de données plus grosses.

</note>

3.Installation et configuration de PostgreSQL 9 sur CentOS

Accéder au référentiel PostgreSQL page de téléchargement, et ajouter le dépôt PostgreSQL 9.4 en fonction de l' architecture de votre serveur.

Pour CentOS 6.x 64bit:

```
#rpm -Uvh http://yum.postgresql.org/9.4/redhat/rhel-6-x86_64/pgdg-centos94-9.4-2.noarch.rpm
```

Pour CentOS 7 64bit:

```
#rpm -Uvh http://yum.postgresql.org/9.4/redhat/rhel-7-x86_64/pgdg-centos94-9.4-2.noarch.rpm
```

Mettre à jour la liste des dépots en utilisant la commande:

```
#yum update
```

<note warning>Les dépôts par défaut de CentOS contiennent des emballages de Postgres, afin que nous puissions les installer sans tracas à l' aide du yum système de package.</note> Installez le paquet postgresql-server et le paquet “contrib”, qui ajoute quelques utilitaires et des fonctionnalités supplémentaires:

```
#yum install postgresql-server postgresql-contrib
```

Accepter l'invite, en répondant avec un y .

Initialisation base de données PostgreSQL en utilisant la commande suivante:

Sur CentOS 6:

```
# service postgresql-setup initdb
```

Sur CentOS 7:

```
#/usr/pgsql-9.4/bin/postgresql94-setup initdb
```

Ensuite, démarrer le service PostgreSQL et le faire démarrer automatiquement à chaque redémarrage.

Sur CentOS 6:

```
#Service postgresql-9.4 start
#chkconfig postgresql-9.4 on
```

Sur CentOS 7:

```
#systemctl enable postgresql-9.4
#systemctl start postgresql-9.4
```

Ensuite, ajuster les iptables et firewall pour accéder aux systèmes postgresql distants.

Sur CentOS 6:

```
#vi /etc/sysconfig/iptables-config
```

Ajouter la ligne suivante:

```
#-A INPUT -m state --state NEW -m tcp -p tcp --dport 5432 -j ACCEPT
#-A INPUT -m state --state NEW -m tcp -p tcp --dport 80 -j ACCEPT
```

Sauvegader et quitter le fichier.Redémarrer le service iptables:

```
#service iptables restart
```

Sur CentOS 7:

```
firewall-cmd --permanent --add-port = 5432 / tcp
firewall-cmd --permanent --add-port = 80 / tcp
firewall-cmd --reload
```

Sauvegader et quitter le fichier.Redémarrer le service iptables:

```
#systemctl restart firewalld.service
```

Exécuter la commande suivante pour que PostgreSQL puisse fonctionné si SELinux est activé sur votre système.

```
#setsebool -P httpd_can_network_connect_db 1
```

<note warning>Vous ne pouvez pas vous connecter à PostegreSQL si vous n'avez pas exécuté la commande ci-dessus.</note>

Authentification du client par mot de passe

L'authentification du client est contrôlée par un fichier, traditionnellement nommé pg_hba.conf et situé dans le répertoire data du groupe de bases de données, par exemple /usr/local/pgsql/data/pg_hba.conf (HBA signifie “host-based authentication”). Un fichier pg_hba.conf par défaut est installé lorsque le répertoire data est initialisé par initdb. Néanmoins, il est possible de placer le fichier de configuration de l'authentification ailleurs.

Chaque enregistrement précise un type de connexion, une plage d'adresses IP (si approprié au type de connexion), un nom de base de données, un nom d'utilisateur et la méthode d'authentification à utiliser pour les connexions correspondant à ces paramètres. Le premier enregistrement qui correspond au type de connexion, à l'adresse client, à la base de données demandée et au nom d'utilisateur est utilisé pour effectuer l'authentification.

Un enregistrement peut avoir l'un des sept formats suivants.

```
local      database  user  auth-method  [auth-options]
host       database  user  address  auth-method  [auth-options]
hostssl    database  user  address  auth-method  [auth-options]
hostnossl  database  user  address  auth-method  [auth-options]
host       database  user  IP-address  IP-mask  auth-method  [auth-options]
hostssl    database  user  IP-address  IP-mask  auth-method  [auth-options]
```

```
hostnossl  database  user  IP-address  IP-mask  auth-method  [auth-options]
```

Les méthodes fondées sur une authentification par mot de passe sont md5 et password. Ces méthodes fonctionnent de façon analogue à l'exception du mode d'envoi du mot de passe à travers la connexion : respectivement, hachage MD5 et texte en clair. <note important>L'authentification MD5 exige que le client fournisse un mot de passe chiffré. Pour ce faire, modifier /var/lib/pgsql/9.4/data/pg_hba.conf : </note>

```
vi /var/lib/pgsql/9.4/data/pg_hba.conf
```

Ajouter ou modifier les lignes comme indiqué ci-dessous:

```
[...]
# TYPE  DATABASE        USER            ADDRESS                 METHOD
.
# "local" is for Unix domain socket connections only
local   all             all                                     md5
# IPv4 local connections:
host    all             all             127.0.0.1/32            md5
host    all             all             192.168.1.0/24          md5
# IPv6 local connections:
host    all             all             ::1/128                 md5
[...]
```

Redémarrer le service de postgresql pour appliquer les modifications:

Sur CentOS6:

```
Service postgresql-9.4 restart
```

Sur CentOS7:

```
systemctl restart postgresql-9.4
```

Configurer PostgreSQL-Configurer TCP/IP

<note warning>Par défaut, la connexion TCP / IP est désactivé, de tel sorte que les utilisateurs d'un autre ordinateur ne peuvent pas accéder à postgresql.</note> Pour permettre aux utilisateurs de se connecter à un autre ordinateur. Modifier le fichier /var/lib/pgsql/9.4/data/postgresql.conf :

```
vi /var/lib/pgsql/9.4/data/postgresql.conf
```

Vous trouverez le résultat suivant:

```
[...]
#listen_addresses = 'localhost'
[...]
#Port = 5432
[...]
```

Décommenter les deux lignes et définir l'adresse IP de votre serveur PostgreSQL ou définir '*' pour écouter de tous les clients comme indiqué ci-dessous:

```
listen_addresses = '*'
port = 5432
```

Redémarrer le service PostgreSQL pour enregistrer les modifications:

Sur CentOS 6:

```
service postgresql-9.4  restart
```

Sur CentOS 7:

```
systemctl restart postgresql-9.4
```

LE CLIENT PostgreSQL

Utilisation

Le nom de base de données et l'utilisateur par défaut sont “postgres”. Basculer vers l'utilisateur postgres pour effectuer des opérations de PostgreSQL

liées:

```
#su - postgres
```

Pour vous connecter à PostgreSQL, entrez la commande:

```
$psql
```

Exemple de sortie:

```
$psql (9.4.0)
Type "help" for help.
postgres = #
```

<note warning>Pour quitter rapidement PosgreSQL, tapez **\q** suivit par **exit** pour retourner au terminal.</note>

Ensuite, ajouter un mot de passe à l'utilisateur “postgres”:

```
su - postgres
bash-4.2$ psql
```

..et définir le mot de passe postgres avec la commande suivante:

```
postgres=# \password postgres
enter new password: centos
Enter it again: centos
postgres=# \q
```

Créer un nouvel utilisateur et base de données

Par exemple, créons un nouvel utilisateur appelé “fenestros” avec mot de passe “centos” et base de données appelée “formation”. Basculer vers l'utilisateur postgres:

```
#su - postgres
```

Créer un utilisateur fenestros.

```
$ Createuser fenestros
```

Créer la base de données:

```
$ Createdb formation
```

<note warning>L'utilisateur créé précédement ce situe dans la base de donnée de postgres et non dans la base de LINUX.</note>

Maintenant, connectez-vous à l'invite de psql, et définir un mot de passe et l'accès Grant à la base de données formation pour fenestros:

```
$ psql
psql (9.4.0)
Type "help" for help.
postgres = # alter user fenestros with encrypted password 'centos';
ALTER ROLE
postgres = # grant all privileges on database formation to fenestros;
GRANT
postgres = #
```

Pour visualiser l'ensemble des bases de données, examinez le catalogue pg_database:

```
SELECT datname FROM pg_database;
```

on obtient un résultat similaire:

```
datname
------------
-template1
-template0
-postgres
-formation
```

```
    (4 rows)
```

Supprimer les utilisateurs et les bases de données

Pour supprimer la base de données, passer à l'utilisateur postgres:

```
        su - postgres
```

Entrez la commande:

```
        $ dropdb formation
```

Pour supprimer un utilisateur, entrer la commande suivante:

```
        $ dropuser fenestros
```

Créer et Supprimer des Tables

Maintenant que vous savez comment se connecter au système de base de données PostgreSQL, nous allons commencer à aller sur la façon de remplir certaines tâches de base. Tout d'abord, recréer la base de données formation puis nous allons créer une table “famille” pour stocker des données. La syntaxe de base de cette commande est quelque chose comme ceci:

```
         $ Createdb formation
         postgres=# CREATE TABLE nom_table (
         postgres=# column_name1 col_type ( field_length ) column_constraints ,
         postgres=# column_name2 col_type (field_length),
         postgres=# column_name3 col_type (field_length),
         postgres=# );
```

Comme vous pouvez le voir, nous donnons à la table un nom, puis on définit les colonnes que nous voulons, ainsi que le type de colonne et la longueur maximale des données de terrain. Nous pouvons également ajouter des contraintes de table pour chaque colonne. Pour nos besoins, nous allons créer un tableau simple comme ceci:

```
         postgres=# CREATE TABLE famille (
```

```
postgres=# code_famille integer primary key,
postgres=# nom text,
postgres=# nombre integer,
postgres=# );
```

Insérer les valeurs dans une table:

```
INSERT INTO famille (code_famille, nom, nombre) VALUES ('1', 'toto', '5');
INSERT INTO famille (code_famille, nom, nombre) VALUES ('2', 'michael', '8');
INSERT INTO famille (code_famille, nom, nombre) VALUES ('3', 'jabeur', '7');
```

Pour visualiser les entrées de cette table, exécuter la commande:

```
SELECT * FROM famille;
```

Vous obtiendrez un résultat similaire:

```
code_famille  |  nom    |  nombre
--------------+---------+----------
   - 1        | toto    |    5
   - 2        | michael |    8
   - 3        | jabeur  |    7
(3 rows)
```

Syntaxe SQL

Le programme PSQL dispose d'un certain nombre de commande interne qui ne sont pas des commandes SQL. Elles commencent avec le caractères “\”. Vous pouvez obtenir de l'aide sur la syntaxe de nombreuses commandes SQL de PostgreSQL en exécutant:

```
formation=# \h
```

1.structure lexicale

Une commande est composée d'une séquence de jetons terminés par un point-virgule. La fin du flux en entrée termine aussi une commande; les jetons

valides dépendent de la syntaxe particulière de la commande. Un jeton peut être un mot clé, un identificateur, un identificateur entre guillemets, une constante ou un symbole de caractère spécial. Les jetons sont normalement séparés par des espaces blancs (espace, tabulation, nouvelle ligne) mais n'ont pas besoin de l'être s'il n'y a pas d'ambiguïté. Par exemple, ce qui suit est valide pour une entrée SQL :

```
SELECT * FROM MA_TABLE;
UPDATE MA_TABLE SET A =5;
INSERT INTO MA_TABLE VALUES (3,'salut ici');
```

2.Valeurs par défaut

Une valeur par défaut peut être attribuée à une colonne. Quand une nouvelle ligne est créée et qu'aucune valeur n'est indiquée pour certaines de ses colonnes, celles-ci sont remplies avec leurs valeurs par défaut respectives. Une commande de manipulation de données peut aussi demander explicitement que la valeur d'une colonne soit positionnée à la valeur par défaut, sans qu'il lui soit nécessaire de connaître cette valeur. Si aucune valeur n'est déclarée explicitement, la valeur par défaut est la valeur NULL. Dans la définition d'une table, les valeurs par défaut sont listées après le type de données de la colonne, par exemple:

```
CREATE TABLE produits(
no_produit integer,
nom text,
prix numeric DEFAULT 9.99,
);
```

3.Contraintes

Les types de données sont un moyen de restreindre la nature des données qui peuvent être stockées dans une table. Pour beaucoup d'applications, la contrainte fournie par ce biais est trop grossière. Une colonne qui contient le prix d'un produit ne doit accepter que des valeurs positives. Mais il n'existe pas de type de données standard qui n'accepte que des valeurs positives. Un autre problème peut provenir de la volonté de contraindre les données d'une colonne par rapport aux autres colonnes ou lignes. Par exemple, dans une table contenant des informations de produit, il ne peut y avoir qu'une ligne par numéro de produit. La contrainte de vérification est la contrainte la plus générique qui soit. Elle permet d'indiquer que la valeur d'une colonne particulière doit satisfaire une expression booléenne (valeur de vérité). Par exemple, pour obliger les prix des produits à être positifs, on peut utiliser :

```
CREATE TABLE produits (
no_produit integer,
```

```
        nom text,
        prix numeric CHECK (prix > 0),
        );
```

5.Modification des tables

Lorsqu'une table est créée et qu'une erreur a été commise ou que les besoins de l'application changent, il est alors possible de la supprimer et de la récréer. Cela n'est toutefois pas pratique si la table contient déjà des données ou qu'elle est référencée par d'autres objets de la base de données. C'est pourquoi PostgreSQL™ offre une série de commandes permettant de modifier une table existante. Il est possible:

1. d'ajouter des colonnes ;
2. de supprimer des colonnes ;
3. d'ajouter des contraintes ;
4. de supprimer des contraintes ;
5. de modifier des valeurs par défaut ;
6. de modifier les types de données des colonnes ;
7. de renommer des colonnes ;
8. de renommer des tables.

Toutes ces actions sont réalisées à l'aide de la commande ALTER TABLE:

```
        ALTER TABLE produits ADD COLUMN description text;
```

La nouvelle colonne est initialement remplie avec la valeur par défaut précisée (NULL en l'absence de clause DEFAULT).

Des contraintes de colonne peuvent être définies dans la même commande à l'aide de la syntaxe habituelle :

```
        ALTER TABLE produits ADD COLUMN description text CHECK(description <>'');
```

6.Droits

Quand un objet est créé, il se voit affecter un propriétaire. Le propriétaire est normalement le rôle qui a exécuté la requête de création. Pour la plupart des objets, l'état initial est que seul le propriétaire (et les superutilisateurs) peuvent faire quelque chose avec cet objet. Pour permettre aux autres rôles de l'utiliser, des droits doivent être donnés.

Il existe un certain nombre de droits différents : SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, CREATE, CONNECT, TEMPORARY, EXECUTE et USAGE. Les droits applicables à un objet particulier varient selon le type d'objet (table, fonction,...). Le droit de modifier ou de détruire un objet est le privilège du seul propriétaire.

La commande GRANT est utilisée pour accorder des privilèges. Par exemple, si jabeur est un utilisateur et michael une table, le privilège d'actualiser la table michael peut être accordé à jabeur avec :

```
GRANT UPDATE ON michael TO jabeur;
```

Écrire ALL à la place d'un droit spécifique accorde tous les droits applicables à ce type d'objet. Le nom d'utilisateur spécial PUBLIC peut être utilisé pour donner un privilège à tous les utilisateurs du système. De plus, les rôles de type “group” peuvent être configurés pour aider à la gestion des droits quand il y a beaucoup d'utilisateurs dans une base. Pour révoquer un privilège, on utilise la commande REVOKE:

```
REVOKE ALL ON michael FROM PUBLIC;
```

Les privilèges spéciaux du propriétaire de l'objet(le droit d'exécuter DROP, GRANT, REVOKE, etc.) appartiennent toujours implicitement au propriétaire. Ils ne peuvent être ni accordés ni révoqués. Mais le propriétaire de l'objet peut choisir de révoquer ses propres droits ordinaires pour mettre une table en lecture seule pour lui-même et pour les autres.

7.Partitionnement

Le partitionnement fait référence à la division d'une table logique volumineuse en plusieurs parties physiques plus petites. Le partitionnement comporte de nombreux avantages :

1#les performances des requêtes peuvent être significativement améliorées dans certaines situations, particulièrement lorsque la plupart des lignes fortement accédées d'une table se trouvent sur une seule partition ou sur un petit nombre de partitions. Le partitionnement se substitue aux colonnes principales des index, réduisant ainsi la taille des index et facilitant la tenue en mémoire des parties les plus utilisées de l'index;

2#lorsque les requêtes ou les mises à jour accèdent à un important pourcentage d'une seule partition, les performances peuvent être grandement améliorées par l'utilisation avantageuse de parcours séquentiels sur cette partition plutôt que d'utiliser un index et des lectures aléatoires réparties sur toute la table;

3#les chargements et suppressions importantes de données peuvent être obtenus par l'ajout ou la suppression de partitions, sous réserve que ce besoin ait été pris en compte lors de la conception du partitionnement. ALTER TABLE NO INHERIT et DROP TABLE sont bien plus rapides qu'une opération de masse;

5.LAB#- Utilisation

Créer une base de données **projet** avec le mot de passe **centos**:

```
$createdb projet
password:
```

<note warning>Le mot de passe ne sera pas visible</note>

Ensuite, entrer la commande suivante pour se connecter à **projet**:

```
$psql projet
password:
```

Créer une table**stagiaire**:

```
projet=# CREATE TABLE stagiaire(
projet=# Code integer,
projet=# Nom varchar,
projet=# Prenom varchar,
projet=# Sujet varchar
         );
```

Insérer les valeurs dans la table:

```
INSERT INTO stagiaire (Code, Nom, Prenom, Sujet) VALUES ('1', 'LEVI', 'Michael', 'PostgresQL');
INSERT INTO stagiaire (Code, Nom, Prenom, Sujet) VALUES ('2', 'MESKINI', 'Jabeur', 'PostgresQL');
INSERT INTO stagiaire (Code, Nom, Prenom, Sujet) VALUES ('3', 'TATINOU KENFACK', 'Stephan', 'Puppet');
INSERT INTO stagiaire (Code, Nom, Prenom, Sujet) VALUES ('4', 'MKACHER', 'Ines', 'Puppet');
INSERT INTO stagiaire (Code, Nom, Prenom, Sujet) VALUES ('5', 'LOGA', 'Patrick', 'JBoss');
INSERT INTO stagiaire (Code, Nom, Prenom, Sujet) VALUES ('6', 'KEFSI', 'Mourad', 'JBoss');
INSERT INTO stagiaire (Code, Nom, Prenom, Sujet) VALUES ('7', 'DIET', 'Antonin', 'MongoDB');
INSERT INTO stagiaire (Code, Nom, Prenom, Sujet) VALUES ('8', 'SAYAVONGSA', 'Rathasath', 'MongoDB');
```

Exécuter la commande:

```
projet=# SELECT * FROM Stagiaire;
```

Vous obtiendrez un résultat similaire:

```
Code  |      nom         |  prenom    |  sujet
------+------------------+----------- +------------
- 1   | LEVI             |  Michael   | PostgresQL
  2   | MESKINI          |  Jabeur    | PostgresQL
  3   | TATINOU KENFACK  |  Stephan   | Puppet
  4   | MKACHER          |  Ines      | Puppet
  5   | LOGA             |  Patrick   | JBoss
  6   | KEFSI            |  Mourad    | JBoss
  7   | DIET             |  Antonin   | MongoDB
  8   | SAYAVONGSA       |  Rathasath | MongoDB
```

1.*Ajouter une colonne*

La commande d'ajout d'une colonne ressemble à :

```
  ALTER TABLE stagiaire ADD COLUMN  num_tel integer;
```

La nouvelle colonne est initialement remplie avec la valeur par défaut précisée (NULL en l'absence de clause DEFAULT). Ajouter ensuite les numéros de tel:

```
UPDATE stagiaire SET num_tel='12345678' WHERE prenom='Michael';
UPDATE stagiaire SET num_tel='23456781' WHERE prenom='Jabeur';
UPDATE stagiaire SET num_tel='34567812' WHERE prenom='Stephan';
UPDATE stagiaire SET num_tel='45678123' WHERE prenom='Ines';
UPDATE stagiaire SET num_tel='56781234' WHERE prenom='Patrick';
UPDATE stagiaire SET num_tel='67812345' WHERE prenom='Mourad';
UPDATE stagiaire SET num_tel='78123456' WHERE prenom='Antonin';
```

```
UPDATE stagiaire SET num_tel='81234567' WHERE prenom='Rathasth';
```

Vous obtiendrez un résultat similaire:

```
Code  |      nom         |  prenom    |  sujet     |  num_tel
------+------------------+----------- +------------+---------
- 1   | LEVI             |  Michael   | PostgresQL | 12345678
  2   | MESKINI          |  Jabeur    | PostgresQL | 23456781
  3   | TATINOU KENFACK  |  Stephan   | Puppet     | 34567812
  4   | MKACHER          |  Ines      | Puppet     | 45678123
  5   | LOGA             |  Patrick   | JBoss      | 56781234
  6   | KEFSI            |  Mourad    | JBoss      | 67812345
  7   | DIET             |  Antonin   | MongoDB    | 78123456
  8   | SAYAVONGSA       |  Rathasath | MongoDB    | 81234567
```

2.*Ajouter une contrainte*

```
ALTER TABLE stagiaire ADD PRIMARY KEY (code);
ALTER TABLE stagiaire ADD CHECK (nom <> '' OR prenom <> '');
ALTER TABLE stagiaire ADD CONSTAINT nom UNIQUE (prenom);
ALTER TABLE stagiaire ALTER COLUMN sujet SET NOT NULL;
```

L'ajout d'une contrainte NOT NULL ne peut pas être écrite sous forme d'une contrainte de table, la syntaxe suivante est utilisée :

```
ALTER TABLE nom_table ALTER COLUMN nom_colonne SET NOT NULL
```

3.*Supprimer une contrainte*

```
ALTER TABLE stagiaire DROP CONSTRAINT nom;
```

4.*Modifier le type de données d'une colonne*

```
ALTER TABLE stagiaire ALTER COLUMN num_tel TYPE varchar(8);
```

5.*Renommer une colonne*

```
ALTER TABLE stagiaire RENAME COLUMN num_tel TO telephone;
```

6.*Renommer une table*

```
ALTER TABLE stagiaire RENAME TO stagiaires ;
```

Pour voir le résultat,exécuter la commande:

```
SELECT * FROM stagiaires;
```

Vous obtiendrez ceci:

```
Code  |      nom         |  prenom   |  sujet         |  telephone
------+------------------+-----------+----------------+------------
- 1   | LEVI             |  Michael  | PostgresQL     | 12345678
  2   | MESKINI          |  Jabeur   | PostgresQL     | 23456781
  3   | TATINOU KENFACK  |  Stephan  | Puppet         | 34567812
  4   | MKACHER          |  Ines     | Puppet         | 45678123
  5   | LOGA             |  Patrick  | JBoss          | 56781234
  6   | KEFSI            |  Mourad   | JBoss          | 67812345
  7   | DIET             |  Antonin  | MongoDB        | 78123456
  8   | SAYAVONGSA       |  Rathasath | MongoDB        | 81234567
```

(source WIKIPEDIA et POSTGRESQL.)