

SQL — 2ème partie

Dominique Gonzalez
Université Lille3-Charles de Gaulle

SQL — 2ème partie
par Dominique Gonzalez

Publié lundi 11 octobre 2010 à 18h16
Copyright © 2010 D.Gonzalez

Ce document est soumis à la licence GNU FDL. Permission vous est donnée de distribuer, modifier des copies de ces pages tant que cette note apparaît clairement.

Table des matières

1. Créer votre propre base.....	1
2. Modification de base, transactions, tables et vues.....	3
2.1. Transactions.....	3
2.1.1. Pourquoi utiliser les transactions ?	3
2.1.2. Comment utiliser les transactions ?	4
2.2. Modifier le contenu	4
2.2.1. Créer une table	4
2.2.2. Insertion de lignes.....	4
2.2.3. Modification de lignes.....	5
2.2.4. Suppression de lignes.....	6
2.2.5. Suppression d'une table.....	6
2.2.6. Modification de la structure d'une table	6
2.3. Vues	6
3. Les droits.....	9
4. Les index	11
4.1. Présentation.....	11
4.2. Création d'un index	11
4.3. Suppression d'un index.....	11
4.4. Inconvénients possibles des index.....	11
4.5. Exercices.....	12
5. Réponses aux exercices sur modification de base, etc.....	13
6. Réponses aux exercices sur les droits.....	15
7. Réponses aux exercices sur les index	17
Index	19

Chapitre 1. Créer votre propre base

Télécharger le fichier `disques2009.sql`¹ qui contient toutes les instructions SQL nécessaires à la création de la base.

Utiliser d'abord la commande :

```
CREATE DATABASE base ;
```

où `base` est un nom à votre choix (qui ne doit pas déjà exister).

Connectez vous ensuite à votre base

```
\c base
```

Puis tapez (ou recopiez) la commande :

```
\i disques2009.sql
```

Cela aura pour effet de créer les tables et de les remplir.

1. <http://www.grappa.univ-lille3.fr/~gonzalez/enseignement/2010-2011/bd/poly/exemples/disques2009.sql>

Chapitre 2. Modification de base, transactions, tables et vues

Vous trouverez les réponses des exercices au Chapitre 5.

L'utilisateur qui a créé les tables a tous les droits sur ces tables. Si rien n'est précisé aucun autre utilisateur ne peut faire la moindre modification sur les tables (contenu ou structure). Ce qui fait que vous ne pourrez tester les commandes de ce chapitre que sur une base de données que vous aurez créée personnellement (voir Chapitre 1).

On étudiera plus tard (Chapitre 3) la possibilité de donner (ou enlever) aux autres utilisateurs les droits de modifier la base de données.

2.1. Transactions

2.1.1. Pourquoi utiliser les transactions ?

Le principe de base des transactions est de grouper un ensemble de commandes de façon à être sûr que TOUTES les commandes seront exécutées, ou sinon qu'AUCUNE ne sera exécutée.

Dans quel but ? Pour pouvoir résoudre un certain de problèmes différents. Par exemple :

- Imaginez une transaction bancaire : la banque doit débiter un compte pour en créditer un autre. Facile à réaliser avec deux commandes UPDATE. Mais que faire si une fois la première commande exécutée, on se rend compte que la deuxième ne peut pas l'être ? Il faut annuler la première. Regrouper les deux commandes dans une transaction permet d'automatiser ce comportement : si tout se passe bien les deux se feront, si un quelconque problème survient aucune ne se fera.
- D'une manière plus générale, une suite de commandes quelconques qui entraîne des modifications de la base ne doit surtout pas être arrêtée en plein milieu, ce qui risquerait de laisser la base dans un état inconsistant. Une transaction transforme cette suite de commandes en un ensemble insécable (on dira aussi *atomique*) : ou TOUT se fait, ou RIEN ne sera fait.
- Les transactions permettent aussi de gérer les accès concurrents dans une base en réseau : si deux (ou plus) utilisateurs utilisent en même temps la même base pour exécuter chacun une série de commandes du style :
 - `commande1` : recherche d'une information dans une table (par exemple : calcul d'un stock disponible) ;
 - `commande2` : modification de la table en fonction de l'information reçue (par exemple retrait d'un certain nombre d'articles dans la limite du stock disponible).

Dans ce cas rien ne permet de supposer que tout se passera bien. Il est tout à fait possible que la séquence de commandes se déroule de la façon suivante : `commande1` pour le premier client, puis `commande1` pour le deuxième client, puis `commande2` pour le premier client, puis `commande2` pour le deuxième client.

Dans ce cas la `commande2` du deuxième client a toutes les chances d'entraîner des erreurs car elle est basée sur un résultat calculé avant l'exécution de la `commande2` du premier client.

L'utilisation de transaction transforme chaque suite (`commande1`, `commande2`) en un bloc non interruptible (*atomique*) qui assure que l'information du deuxième ne sera fournie que quand le premier aura modifié la table.

- Enfin il peut être utile de se ménager une *porte de sortie* quand on exécute une commande modifiant la table : on peut se tromper, changer d'avis, vouloir seulement faire un essai, etc. Il est toujours utile de disposer de l'équivalent du *undo* de la plupart des logiciels.

La seule solution est alors d'utiliser une transaction : une fois les commandes exécutées, on peut décider de tout abandonner (comme pour un *undo*) ou au contraire d'accepter tout.

Vous pourrez trouver d'autres explications sur l'utilité des transactions dans les pages suivantes :

- Comment protéger des ensembles d'opérations par des transactions ?¹
- À quoi servent les transactions ?²

1. <http://hcesbronlavau.developpez.com/Transactions/>

2. <http://sqlpro.developpez.com/cours/sqlaz/techniques/#L1>

2.1.2. Comment utiliser les transactions ?

Une *transaction* commence par « BEGIN ; » :

- Si elle se termine par « ROLLBACK ; », aucune des modifications faites ne sera prise en compte.
- Si elle se termine par « COMMIT ; », toutes les modifications sont prises en compte.
- Ne pas taper « BEGIN ; » avant de commencer vos modifications revient à taper chacune de vos commandes en mode *autocommit* : chaque instruction isolée est une transaction en elle-même, une fois tapée elle est exécutée comme si elle était précédée d'un « BEGIN ; » et suivie d'un « COMMIT ; », et elle ne peut plus être défaite.

Toutes les commandes qui vont suivre dans ce chapitre vont modifier votre base de données. Nul n'est à l'abri d'une erreur, il vous est donc fortement conseillé d'inclure vos commandes dans des transactions, pour vous permettre d'annuler éventuellement toute fausse manœuvre.

Dans notre cas il ne se produira rien de catastrophique : cette base n'a pas d'importance vitale, et si vous y faites des erreurs, il vous reste toujours la possibilité de la recréer complètement (Chapitre 1).

Dans la *vraie* vie c'est rarement le cas. C'est donc une bonne habitude à prendre...

2.2. Modifier le contenu

2.2.1. Créer une table

La commande `CREATE TABLE` permet de créer une table. On la fait suivre d'une liste (entre parenthèses) des champs accompagnés de leur type.

Exemple 2-1. CREATE TABLE

La table `CHANSON` a été créée par l'instruction :

```
CREATE TABLE chanson (  
    cha_num INTEGER NOT NULL PRIMARY KEY,  
    cha_titre CHARACTER VARYING,  
    cha_genre INTEGER REFERENCES genre(gen_num),  
    cha_texte CHARACTER VARYING,  
    cha_libre BOOLEAN,  
    cha_modif TIMESTAMP WITHOUT TIME ZONE DEFAULT CURRENT_TIMESTAMP);
```

Quelques détails :

- `PRIMARY KEY` désigne la clef primaire.
- `REFERENCES genre(gen_num)` signifie que le champ `cha_genre` est une clef étrangère (ou `FOREIGN KEY`) faisant référence au champ `gen_num` de la table `GENRE`.
- `DEFAULT` désigne une valeur par défaut : si ce champ n'est pas rempli lors d'une commande `INSERT` (voir Section 2.2.2) c'est cette valeur qui sera utilisée.

2.2.2. Insertion de lignes

La commande `INSERT` permet de créer une nouvelle ligne dans une table.

Par exemple, pour insérer une nouvelle ligne dans la table `ARTISTE`, et par la même occasion tester les transactions, essayez la suite de commandes que voici :

```

BEGIN ; /* début de transaction */
/* insertion d'une ligne dans la table ARTISTE */
INSERT INTO artiste
VALUES (9999 , 'Arnaud' , 'Bodinoz' , NULL ) ;
SELECT *
FROM artiste
WHERE art_num>9000 ; /* liste, pour vérifier l'insertion */ ;
ROLLBACK ; /* on annule la transaction */

SELECT *
FROM artiste
WHERE art_num>9000; /* la nouvelle ligne n'est plus là */

BEGIN ; /* on recommence... */
INSERT INTO artiste
VALUES (9999 , 'Arnaud' , 'Bodinoz' , NULL ) ;
SELECT *
FROM artiste
WHERE art_num>9000 ; /* on vérifie à nouveau */
COMMIT ; /* on confirme la transaction */

SELECT *
FROM artiste
WHERE art_num>9000 ; /* la nouvelle ligne est toujours là */

```

La mise à jour n'est prise en compte que si l'intégrité des données est satisfaite.

Par exemple, si on veut insérer une nouvelle ligne dans la table ARTISTE avec le numéro d'artiste non renseigné :

```

INSERT INTO artiste /* cela produit une erreur */
VALUES (null,'Sophie', 'Fonfec', NULL) ;
SELECT *
FROM artiste
WHERE art_nom = 'Fonfec' ; /* l'insertion n'a pas eu lieu */

```

Même quand on ne souhaite pas remplir toutes les colonnes on peut insérer une nouvelle ligne dans la table ARTISTE sans utiliser la clause NULL pour les valeurs non renseignées.

```

BEGIN ;
/* on crée une nouvelle ligne en ne remplissant que 2 champs */
INSERT INTO artiste (art_num, art_nom)
VALUES (10000, 'Steppenwolf') ;
SELECT *
FROM artiste
WHERE art_nom = 'Steppenwolf' ;
COMMIT ;

```

2.2.3. Modification de lignes

La commande UPDATE permet de mettre à jour dans une table une ou plusieurs colonnes pour une ou plusieurs lignes.

Pour mettre à jour le prix d'achat du disque dont le titre est Cannonball, en l'augmentant de 10%, il faudra taper :

```

BEGIN ; /* ça DOIT être un réflexe avant un UPDATE */
SELECT *
FROM disque
WHERE dis_titre = 'Cannonball' ;
UPDATE disque /* modification */
SET dis_prixachat = dis_prixachat * 1.1
WHERE dis_titre = 'Cannonball' ;
SELECT * /* vérification */
FROM disque
WHERE dis_titre = 'Cannonball' ;
COMMIT ; /* confirmation */

```

1. Mettre à la date d'aujourd'hui (`CURRENT_TIMESTAMP`) les dates de modification des fiches d'artistes dont ce champ n'est pas renseigné.
2. Mettre à `Anonyme` les droits des personnes enregistrées comme `Membre` dans la table `AUTORISATIONS`.

2.2.4. Suppression de lignes

La commande `DELETE` permet de supprimer une ou plusieurs lignes d'une table. Si elle n'est pas suivie d'une restriction (`WHERE`, avec la même syntaxe que pour `SELECT`), c'est tout le contenu de la table qui est effacé.

3. Supprimer de la table `ARTISTE` l'artiste `Steppenwolf`.

2.2.5. Suppression d'une table

La commande `DROP TABLE` permet de détruire une table (lignes et structure).

Pour supprimer totalement la table `AUTORISATIONS` :

```
DROP TABLE autorisations ;
```

Pour la recréer voir Chapitre 1.

2.2.6. Modification de la structure d'une table

La commande `ALTER TABLE` permet de modifier la structure de la table, même si elle contient des données.

Les principales façons de l'utiliser sont :

```
ALTER TABLE nom RENAME TO nouveau_nom ;
```

renommer une table

```
ALTER TABLE nom RENAME colonne TO nouvelle_colonne ;
```

renommer une colonne d'une table

```
ALTER TABLE nom ADD colonne type ;
```

ajouter une colonne à une table

```
ALTER TABLE nom DROP colonn ;
```

supprimer une colonne

```
ALTER TABLE nom ALTER colonne TYPE type ;
```

changer le type d'une colonne

```
ALTER TABLE nom OWNER TO nouveau_propriétaire ;
```

changer le propriétaire d'une table

2.3. Vues

Une vue est une table virtuelle, c'est-à-dire dont les données ne sont pas stockées dans une table de la base de données, et dans laquelle il est possible de rassembler des informations provenant de plusieurs tables. On parle de « vue » car il s'agit simplement d'une représentation des données dans le but d'une exploitation visuelle. Les données présentes dans une vue sont définies grâce à une clause `SELECT`.

Une vue est une définition dynamique dans le sens suivant : si on modifie, ou si on ajoute ou si on supprime des enregistrements, chaque exécution de la vue comprendra ces modifications.

La norme SQL propose un ensemble important de restrictions pour la modification ou l'insertion ou la modification des données dans les vues. Les systèmes de gestion de base de données ont aussi chacun leur implantation de ce concept et chacun leurs contraintes et restrictions. En particulier, peu d'opérations sont autorisées dès qu'une vue porte sur plusieurs tables ; aucune n'est possible si la vue comporte des opérateurs d'agrégation.

En PostgreSQL les vues ne sont que consultables par des instructions SELECT. Aucune autre opération n'est possible. Par contre, c'est la notion propre à PostgreSQL de règles qui assure cette fonctionnalité. Cette notion s'avère plus souple et puissante que les restrictions communément appliquées aux SGBD classiques.

La commande CREATE VIEW permet de créer une vue relative à une ou plusieurs tables de la base. Elle est suivie de AS puis d'une requête qui la définit.

4. Créer une vue `disqueartiste` contenant les titres des disques, les noms, prénoms et groupes des artistes, ainsi que les identifiants des disques et des artistes. Utiliser la vue pour interroger. Mettre à jour `ARTISTE` et utiliser la vue pour interroger.

Chapitre 3. Les droits

Vous trouverez les réponses des exercices au Chapitre 6.

Une table ayant été créée, son propriétaire peut accorder (`GRANT`) ou retirer (`REVOKE`) à un autre utilisateur (ou à tous : `PUBLIC`) les droits suivants :

- `SELECT` : Accès en lecture à toutes les colonnes d'une table ou d'une vue.
- `INSERT` : Insérer des données dans toutes les colonnes d'une table.
- `UPDATE` : Mettre à jour dans toutes les colonnes d'une table.
- `DELETE` : Supprimer des enregistrements d'une table.
- `ALL` : Donner tous les privilèges.

La syntaxe en est :

```
GRANT droit ON table TO user ;  
REVOKE droit ON table FROM user ;
```

À tout moment vous pouvez afficher les droits actuellement donnés par la commande « `\dp` » ou « `\z` »¹.

1. Accorder le droit de `SELECT` sur la table `GENRE` à un autre *user*.
2. Accorder les droits de `INSERT` et `UPDATE` sur `GENRE` à un autre *user*.
3. Accorder tous les droits sur la table `DISQUE` à un autre *user*.
4. Créer une vue relative au titre, à l'artiste (prénom+nom+groupe) et à l'année des disques. Puis accorder les droits de `SELECT` sur la vue à un autre *user*.
5. Créer une vue relative à toutes les informations (titre de la chanson, titre du disque, artiste du disque, position sur le disque, année du disque) des chansons interprétées par David BOWIE. Puis accorder tous les droits sur la vue à un autre *user*.
6. Retirer le droit `INSERT` à l'autre *user*, sur la table `GENRE`.

1. Ces commandes ne sont pas du SQL, mais des commandes de l'interface `psql`. Elles peuvent différer d'une version à l'autre. Vous pouvez obtenir la liste de toutes les commandes disponibles en tapant « `\?` ».

Chapitre 4. Les index

Vous trouverez les réponses des exercices au Chapitre 7.

4.1. Présentation

Les index nous permettent de retrouver rapidement les données contenues dans une table. Utilisons un exemple pour illustrer l'utilité des index : supposons que nous sommes intéressés à lire des informations sur la culture de piments dans un livre de jardinage. Au lieu de commencer la lecture dès le début jusqu'à parvenir à la section sur les piments, il est beaucoup plus rapide d'aller à l'index en fin de livre, repérer les pages contenant les informations au sujet des piments, puis aller directement à ces pages. Aller à l'index en premier lieu nous permet d'épargner du temps et c'est une méthode largement plus efficace pour repérer les informations dont nous avons besoin.

Le principe est le même quant à retrouver des données d'une table de bases de données. Sans index, le système de base de données balaie la table entière (ce processus est appelé *table scan* (balayage de table)) pour trouver les informations recherchées. Avec l'index créé, le système de la base de données peut d'abord repérer dans l'index l'emplacement des données, puis aller directement à ces emplacements afin d'obtenir les données requises. C'est beaucoup plus rapide.

4.2. Création d'un index

L'instruction `CREATE INDEX` construit un index sur la table spécifiée. Les index sont principalement utilisés pour améliorer les performances de la base de données (bien qu'une utilisation inappropriée puisse produire l'effet inverse).

La syntaxe de base est :

```
CREATE INDEX nom d'index
ON nom de table (nom de colonne) ;
```

Les champs clé pour l'index sont spécifiés à l'aide de noms des colonnes ou par des expressions écrites entre parenthèses. Plusieurs champs peuvent être spécifiés si la méthode d'indexation supporte les index multi-colonnes.

Un champ d'index peut être une expression calculée à partir des valeurs d'une ou plusieurs colonnes de la ligne de table. Cette fonctionnalité peut être utilisée pour obtenir un accès rapide à des données obtenues par transformation des données basiques. Par exemple, un index calculé sur `upper(col)` autorise la clause `WHERE upper(col) = 'JIM'` à utiliser un index.

4.3. Suppression d'un index

La syntaxe est :

```
DROP INDEX nom d'index ;
```

4.4. Inconvénients possibles des index

La gestion des index nécessite des calculs complexes qui peuvent s'avérer gourmands en temps de calcul. En cas d'ajouts massifs dans une table indexée (et encore plus si elle possède plusieurs index), le temps nécessaire à la mise à jour de l'index peut être prohibitif.

Il peut être alors préférable de supprimer les index, d'ajouter les enregistrements, et de recréer les index.

Attention

Ces inconvénients peuvent survenir en cas d'ajout d'un très grand nombre d'enregistrements, de l'ordre de plusieurs milliers. Même pour quelques centaines d'enregistrements ajoutés, ces inconvénients sont souvent négligeables.

Attention

Ces ralentissements sont proportionnels au nombre d'index existant sur la table. Il est donc important de savoir se limiter et de ne créer que les index nécessaires.

4.5. Exercices

1. Calculer le temps d'exécution de la requête :

```
SELECT cha_num,cha_titre FROM chanson WHERE cha_titre ='Lady Stardust' ;
```

Créer ensuite un index sur le titre des chansons, et recommencez l'opération.

2. Supprimer l'index précédent.

Chapitre 5. Réponses aux exercices sur modification de base, etc.

Vous trouverez les énoncés correspondant au Chapitre 2.

Solution de l'exercice 1

```
BEGIN ;
SELECT COUNT(*)
  FROM artiste
 WHERE art_modif IS NULL ;

UPDATE artiste
  SET art_modif = CURRENT_TIMESTAMP
 WHERE art_modif IS NULL ;

SELECT COUNT(*)
  FROM artiste
 WHERE art_modif IS NULL ;
COMMIT ;
```

Solution de l'exercice 2

```
BEGIN ;
SELECT *
  FROM autorisations
 WHERE aut_droits IN (SELECT dro_num
                      FROM droits
                      WHERE dro_nom='Membre') ;

UPDATE autorisations
  SET aut_droits = (SELECT dro_num
                   FROM droits
                   WHERE dro_nom='Anonyme')
 WHERE aut_droits IN (SELECT dro_num
                     FROM droits
                     WHERE dro_nom='Membre') ;

SELECT *
  FROM autorisations
 WHERE aut_droits IN (SELECT dro_num
                      FROM droits
                      WHERE dro_nom='Membre') ;

COMMIT ;
```

Solution de l'exercice 3

```
BEGIN ;
DELETE
  FROM artiste
 WHERE art_nom = 'Steppenwolf' ;
COMMIT ;
```

Solution de l'exercice 4

```
CREATE VIEW disqueartiste
  AS SELECT dis_titre AS da_titre, art_prenom AS da_prenom,
           art_nom AS da_nom, art_groupe AS da_groupe,
           dis_num AS da_disnum, art_num AS da_artnnum
  FROM disque, artiste
 WHERE dis_artiste = art_num ;

SELECT *
  FROM disqueartiste
 WHERE da_nom LIKE 'Ar%' ;

UPDATE artiste
```

Chapitre 5. Réponses aux exercices sur modification de base, etc.

```
    SET art_nom = 'Arnaud'  
WHERE art_nom = 'Arno' ;  
SELECT *  
    FROM disqueartiste  
WHERE da_nom LIKE 'Ar%' ;
```

Chapitre 6. Réponses aux exercices sur les droits

Vous trouverez les énoncés correspondant au Chapitre 3.

Solution de l'exercice 1

```
GRANT SELECT
  ON genre
  TO nom de l'autre user ;
```

Solution de l'exercice 2

```
GRANT INSERT,UPDATE
  ON genre
  TO nom de l'autre user ;
```

Solution de l'exercice 3

```
GRANT ALL ON disque TO nom de l'autre user ;
```

Solution de l'exercice 4

```
CREATE VIEW disqvue
  AS SELECT TRIM(art_prenom||' '||art_nom||' '||art_groupe) AS artiste,
           dis_titre AS titre,
           dis_annee AS annee
  FROM disque,artiste
  WHERE dis_artiste = art_num ;
GRANT SELECT ON disqvue
  TO nom de l'autre user ;
```

Solution de l'exercice 5

```
CREATE VIEW davidBowie
  AS SELECT cha_titre AS "titre de la chanson",
           dis_titre AS "titre du disque",
           TRIM(d.art_prenom||' '||d.art_nom||' '||d.art_groupe) AS "artiste",
           int_numero AS "position sur le disque",
           dis_annee AS "année du disque"
  FROM disque,artiste d,artiste b,chanson,interprete
  WHERE cha_num = int_chanson
        AND int_disque = dis_num
        AND dis_artiste = d.art_num
        AND int_artiste = b.art_num
        AND LOWER(b.art_prenom) = 'david'
        AND LOWER(b.art_nom) = 'bowie'
        AND LOWER(b.art_groupe) = " " ;
GRANT ALL
  ON davidBowie
  TO nom de l'autre user ;
```

Solution de l'exercice 6

```
REVOKE INSERT ON genre FROM nom de l'autre user ;
```


Chapitre 7. Réponses aux exercices sur les index

Vous trouverez les énoncés correspondant au Chapitre 4.

Solution de l'exercice 1

Écrire une ligne de commandes commençant par

```
SELECT NOW();
```

contenant ensuite 10 fois

```
SELECT cha_num,cha_titre FROM chanson WHERE cha_titre ='Lady Stardust';
```

et se terminant par

```
SELECT NOW();
```

Il vous reste à faire la différence des deux temps, et à diviser par 10... Vous avez alors la durée nécessaire à l'exécution de cette requête sans index.

Pour créer l'index :

```
CREATE INDEX idx_nom ON chanson (cha_titre) ;
```

En recommençant la première opération vous pourrez calculer la durée nécessaire à l'exécution de cette requête avec index. Cette dernière durée doit être beaucoup plus petite que la première.

Solution de l'exercice 2

```
DROP INDEX idx_nom ;
```


Index

- accès concurrents, 3
- ajouts massifs, 11
- ALTER
 - TABLE, 6
- atomique, 3
- autocommit, 4
- balayage de table, 11
- BEGIN, 4
- clef
 - primaire, 4
 - étrangère, 4
- COMMIT, 4
- CREATE
 - DATABASE, 1
 - INDEX, 11
 - TABLE, 4
 - VIEW, 7
- culture, 11
- DEFAULT, 4
- DELETE, 6, 9
- droits, 3
- DROP
 - INDEX, 11
 - TABLE, 6
- FOREIGN KEY, 4
- GRANT, 9
- index, 11
- INSERT, 4, 9
- jardinage, 11
- performance, 11
- piment, 11
- PRIMARY KEY, 4
- REFERENCES, 4
- REVOKE, 9
- ROLLBACK, 4
- SELECT, 9
- table scan, 11
- temps de calcul, 11
- transaction, 3
- undo, 3
- UPDATE, 5, 9
- VALUES, 4

