Travaux Pratiques de SQL

Basé sur les notes de cours d'Alain Buys

25 octobre 2017

Table des matières

Notes de version

- Première version : chapitres 1 à 13.
- (23 Octobre 2017) : ajout du dernier chapitre.
- (25 Octobre 2017) : révisions orthographiques.

Chapitre 1

Introduction

Ce document constitue une adaptation textuelle des slides d'Alain Buys afin de faciliter la compréhension de l'étudiant et de préciser certains aspects de MySQL, le système d'interaction de base de données relationnelles employé dans le cadre des travaux pratiques donnés à l'UMONS dans le cadre du cours de Bases de Données I.

Nous travaillons avec des bases de données relationnelles (le modèle relationnel permettant d'exprimer au travers de l'algèbre relationnelle la majorité des requêtes). Le langage employé afin d'interroger et manipuler les bases de données s'appelle **SQL**, acronyme de Structured Query Language (lit. langage de requêtes structurées). Soulignons qu'il ne s'agit pas d'un langage procédural. Il ne peut donc pas être employé à la manière des langages de programmation tels Java et Python.

Dans le cadre de ce document, nous ne nous intéresserons qu'au système de gestion de base de données (abrégé SGBD) MySQL. Pour plus d'informations à son propos, veuillez directement consulter le site `https://dev.mysql.com/`.

1.1 Terminologie

Une base de données relationnelle est une collection de relations ou de tables à deux dimensions.

- **Rangée** : il s'agit d'une rangée de la table, ce qui correspond à un tuple de la relation en question.
- **Colonne** : une colonne d'une table contient toutes les valeurs pour l'attribut correspondant.
- **Clés primaires et extérieures** : permettent de préciser des contraintes sur les tables. D'autres contraintes existent aussi, notamment une contrainte d'unicité.

1.2 Types d'instructions en SQL

- `SELECT` : sert à l'extraction de données ;
- `INSERT, UPDATE` et `DELETE` : font partie du langage de manipulation de données (Data Manipulation Language, DML) ;
- `CREATE, ALTER, DROP, RENAME` et `TRUNCATE` : font partie du langage de définition de données (Data Definition Language, DDL) ;

- `GRANT` et `REVOKE` : font partie du langage de contrôle de données (Data Control Language DCL).

Chapitre 2

Instructions SQL de base

2.1 La commande SELECT

Cette commande permet de sélectionner des rangées, de projeter des rangées et de joindre des rangées de différentes tables. La syntaxe est la suivante [1] :

```
SELECT [DISTINCT] nomColonne1 [[AS] alias1], nomColonne2
   [[AS] alias2], ...
FROM nomTable;
```

Il s'agit de la manière la plus simple de faire appel à la clause `SELECT`. Expliquons en détail tout ce qui précède ;

- Les mots `SELECT`, `DISTINCT` et `FROM` sont des mots-clefs du langage SQL.
  - La clause `SELECT` spécifie les colonnes à extraire des tables. Si le symbole * est spécifié, alors toutes les colonnes sont extraites.
  - La clause `FROM` spécifie la table dont sont extraites les données.
  - La clause `DISTINCT` évite l'affichage de doublons dans la réponse d'une requête.
- Une requête peut prendre plusieurs lignes et s'achève systématiquement par " ;".
- Les mots-clefs ne peuvent être raccourcis.

Exemple 1. Voici un exemple de clause `SELECT` :

```
- SELECT * FROM dept;
- SELECT deptno "Department Number", loc Location FROM dept;
```

Dans le premier exemple, toutes les données de la table *dept* sont affichées. Dans le second exemple, seules les colonnes *deptno* et *loc* sont affichées et à l'affichage, leur nom est remplacé par les alias Department Number et Location (notez la présence de guillemets encadrant Department Number, ils sont nécessaires car il y a plus d'un mot dans l'alias).

Le contenu des tables affichées est aligné de manière différente selon la nature du contenu ; les dates et chaînes de caractère sont alignées par défaut sur la gauche, alors que les nombres sont alignés par défaut sur la droite.

Il est également possible d'utiliser des opérateurs arithmétiques sur les nombres et les dates, suivant l'ordre de précédence $*$, $/$ puis $+$, $-$ (sauf en présence de parenthèses).

1. Les éléments entre crochets sont des éléments optionnels.

Exemple 2. On emploie les opérateurs arithmétiques pour calculer le salaire annuel d'un employé.

```
SELECT name, sal, 12*sal+100 FROM emp;
```

Il est important de remarquer que si une opération contient la valeur `NULL`, alors le résultat retourné sera également `NULL`.

Il n'y a pas d'opérateur de concaténation de chaînes de caractères dans le langage SQL ; on emploie à cette fin la fonction `concat(chaîne`$_1$`, ..., chaîne`$_n$`)`

Exemple 3. On peut ajouter manuellement des chaînes de caractère en tant qu'argument de la fonction `concat` :

```
SELECT concat('VAN DEN ', name) AS "Full Name" FROM emp;
```

2.2 Commandes MySQL

L'environnement MySQL possède son propre ensemble de commandes. Elles ne permettent toutefois pas de manipuler les tables par opposition au langage SQL. Notons que contrairement aux instructions SQL, les commandes de MySQL peuvent être abrégées.

2.2.1 DESCRIBE

Peut être abrégé en `DESC`. Cette commande décrit la structure d'une table. Elle affiche une table contenant les champs suivants :

- **Field** : correspond au nom de l'attribut de la table dont les spécifications sont données dans la rangée ;
- **Type** : indique le type de donnée contenu dans la colonne ;
- **Null** : indique si la colonne peut ou non contenir la valeur null ;
- **Key** : indique si l'attribut a une contrainte de type clef primaire ou clef étrangère (foreign key) ;
- **Default** : indique si une valeur par défaut a été spécialisée pour cet attribut.

2.2.2 Autres commandes

- `show tables` : fournit une table des tables contenues dans la base de données sélectionnée ;
- `Source` (abrégé \.) : permet d'exécuter un script de commandes SQL ;
- `Edit` (abrégé \e) : édite le buffer SQL ;
- `System` (abrégé \!) : suivi d'une commande, il la passe directement à l'OS (sur Unix).

Chapitre 3

Sélection et classement des données

Afin d'introduire des conditions sur la sélection effectuée, on utilise une clause `WHERE` :

3.1 Clause `WHERE`

```
SELECT [DISTINCT] col1 [alias1], ...
  FROM nomTable
        [WHERE condition(s)];
```

Exemple 4. On doit écrire une requête qui sélectionne d'une table d'employés ceux dont le travail est CLERK :

```
SELECT ename, job,
 FROM emp
 WHERE job='CLERK';
```

Notez les choses suivantes :

- Les chaînes de caractère et les dates doivent être spécifiés entre apostrophes ;
- En général, les comparaisons de chaînes de caractères sont sensibles à la casse, mais pas en MySQL ;
- Le format des dates doit correspondre. Le format par défaut est AAAA-MM-JJ.

On exprime une condition comme suit :

```
WHERE expression operateur valeur
```

3.1.1 Opérateurs de comparaison

Les opérateurs de comparaison sont :

- = : égalité ;
- > : plus grand ;
- >= : plus grand ou égal ;
- <: plus petit ;
- <= : plus petit ou égal ;

- `<>` : différent.
- `BETWEEN` x `AND` y : vérifie si l'expression est entre les valeurs x et y ;
- `LIKE` : permet d'établir une condition sur les caractères :
  1. "`_`" désigne un caractère arbitraire,
  2. "`%`" désigne un nombre quelconque de caractères arbitraires,
  3. "`\`" permet d'employer les deux caractères précédents comme des caractères usuels.
- `IS NULL` : vérifie si une valeur est nulle ou non.

Soulignons que ces opérateurs sont utilisables sur tous types de données, mis à part `LIKE` qui ne sert que pour les chaînes de caractères.

Exemple 5. On veut la liste des employés dont le nom contient la lettre s en deuxième position :

```
SELECT name FROM employee
  WHERE name LIKE '_s%';
```

3.1.2 Opérateurs logiques

Les trois opérateurs logiques sont `AND`, `OR` et `NOT`. Ils ont une précédence moindre que les opérateurs de comparaison, sinon par ordre décroissant de précédence, on a `NOT`, `AND` et finalement `OR`. L'opérateur de négation peut être associé à d'autres opérateurs, parfois même en utilisant la syntaxe de la langue anglaise.

Exemple 6. On recherche la liste des employés sont les commissions sont différentes de `NULL` :

```
SELECT name FROM employee
  WHERE comm IS NOT NULL;
```

Cette syntaxe est acceptée ! On se serait plutôt attendu à `comm NOT IS NULL`, de manière similaire aux autres associations de `NOT` avec des opérateurs de comparaison.

3.2 La clause `ORDER BY`

La clause `ORDER BY`. Elle permet de classer les données à l'affichage par ordre croissant (par défaut, elles sont classées par ordre décroissant si DESC est spécifié). La syntaxe est la suivante :

```
SELECT [DISTINCT] nomCol1, nomCol2, ...
  FROM nomTable
       [WHERE condition(s)]
       [ORDER BY {nomCol|expression} [ASC|DESC] ];
```

Exemple 7. On désire trier les employés par date d'embauche :

```
SELECT name, job, hiredate FROM employee
  ORDER BY hiredate;
```

On désire trier par salaire annuel (à l'aide d'un alias) :

```
SELECT name, job, sal*12 annSal FROM employee
  ORDER BY annSal;
```

On désire trier les employés selon le département et le salaire (colonnes multiples) :

```
SELECT name, sal, deptno FROM employee
  ORDER BY deptno, sal;
```

Chapitre 4

Fonctions sur une rangée

4.1 Syntaxe

```
nomFonction(colonne ou expression, [arguments])
```

Ces fonctions retournent une valeur par rangée. Elles permettent de manipuler les chaînes de caractères, les nombres, les dates et d'effectuer des conversions de type. Elles peuvent apparaître dans les clauses `SELECT`, `WHERE` et `ORDER BY`.

4.2 Fonctions sur les types usuels

Listons quelques fonctions utiles sur les types récurrents ;

4.2.1 Fonctions sur les caractères

- `LOWER(colonne ou expression)` : retourne la chaîne en miniscules ;
- `UPPER(colonne ou expression)` : retourne la chaîne en majuscules ;
- `CONCAT(param1, param2, ...)` : retourne la concaténation des paramètres ;
- `SUBSTRING(colonne ou expression, m [, n])` : retourne la sous-chaîne à partir de la position m, longue de n caractères. Si $m < 0$, on compte à partir de la fin ;
- `LENGTH(colonne ou expression)` : retourne la longueur de la chaîne ;
- `INSTR(colonne ou expression, c)` : retourne la position du caractère c ;

4.2.2 Fonctions sur les nombres

- `ROUND(nombre ou expression)` : arrondit selon la règle usuelle ;
- `TRUNCATE(nombre ou expression)` : omet les décimales ;
- `MOD(a, b)` : retourne le reste de la division entière de a par b ;

4.2.3 Fonctions sur les dates

A compléter.

4.2.4 Table "dual"

Il s'agit d'une table d'une rangée et d'une seule colonne qui permet d'afficher un texte ou le résultat d'une fonction.

Exemple 8. Obtenir la date courante du système :

```
SELECT sysdate()
 FROM dual;
```

4.3 Conversions

4.3.1 Conversions implicites

Les conversions de nombre décimaux vers les chaînes de caractères sont implicites, de même pour les dates vers les chaînes de caractères.

Exemple 9. Obtenir la date d'embauche d'un employé et son salaire sous forme d'une unique chaîne de caractères :

```
SELECT concat(hiredate, ' ', sal)
 FROM emp;
```

4.3.2 Conversions explicites

Afin d'expliciter une conversation d'un type vers un autre, on adopte la syntaxe `CONVERT(d, type)` :

Exemple 10. Obtenir sous forme de nombre décimal la date d'embauche des employés :

```
SELECT CONVERT(hiredate, decimal)
 FROM emp;
```

4.3.3 Fonctions diverses

`IFNULL`

La fonction `IFNULL` permet de convertir une valeur nulle en une valeur réelle. C'est un outil utile pour éviter que `NULL` ne nuise à des opérations arithmétiques (textSQLIS NULL convient mieux aux opérations booléennes).

Exemple 11. Calculer le revenu total d'un employé (pour les employés qui ne touchent pas de commissions, le champ est nul) :

```
SELECT name, sal + IFNULL(comm, 0) FROM emp;
```

`CASE`

La fonction `CASE` permet de distinguer des cas selon une valeur donnée :

```
CASE value WHEN cvalue1 THEN result1
                [WHEN cvalue 2 then result2 ...]
                [ELSE result] END
```

Chapitre 5

Données de tables multiples

On peut sélectionner des données provenant de multiples tables.

```
SELECT table1.col, table2.col, ...
   FROM table1, table2
                WHERE table1.col = table2.col;
```

Notez la présence d'une condition dans la forme syntaxique ; si elle est omise, on obtient un produit cartésien des tables. Les noms des colonnes doivent être explicité que si deux tables partagent des noms de colonnes.

La condition donnée ci-haut étant une égalité, on parle d'équi-jonction (on fait correspondre à une clef primaire une clef étrangère). Si la condition n'est pas une égalité, on parle simplement de non-équi-jonction. Il est possible d'employer les opérateurs logiques dans la condition. Il est également possible d'employer des alias dans la clause `FROM` pour s'en servir dans le reste de l'instruction (en réalité tout alias peut-être utilisé au lieu du nom réel d'une expression dans le reste de l'instruction).

Exemple 12. On désire de savoir où se situe le département de chaque employé :

```
SELECT e.ename, e.deptno, d.loc
  FROM emp e, dept d
        WHERE e.deptno = d.deptno;
```

Il est également possible de joindre une table avec elle-même, ce qui est intéressant si une clef étrangère de la table est dans elle-même. On parle alors de self-jonction. Pour ce faire, il suffit de donner deux alias différents à une même table.

Exemple 13. On veut obtenir le nom du manager de chaque employé

```
SELECT e.ename, m.ename
  FROM emp e, emp m
        WHERE e.mgr = m.empno;
```

Chapitre 6

Fonctions de groupe

6.1 Quelques fonctions de groupe

Il s'agit de fonctions qui agissent sur un groupe de rangées. La syntaxe est la suivante :

```
SELECT [col1, ...] groupFunction([DISTINCT | ALL]col)
  FROM nomTable
       [WHERE condition]
       [GROUP BY col]
       [ORDER BY col];
```

Le mot-clef `DISTINCT` permet d'ignorer les doubles, le mot clef `ALL` quant à lui est la valeur par défaut. En général, les valeurs nulles sont ignorées. Énumérons certaines de ces fonctions :

- `AVG` : retourne la moyenne des valeurs de la colonne sélectionnée (nécessite un paramètre numérique) ;
- `COUNT` : compte les rangées pour lesquelles l'expression en paramètre n'a pas une valeur nulle, compte toutes les rangées si on lui passe le paramètre * ;
- `SUM` : retourne la somme des valeurs de la colonne sélectionnée (nécéssite un paramètre numérique) ;
- `SDTDEV` et `VARIANCE` : retourne l'écart-type ou la variance respectivement du contenu de la colonne sélectionnée (nécessite un paramètre numérique) ;
- `MIN` et `MAX` : retourne le minimum ou les maximum de la colonne passé en paramètre. Fonctionne également avec des dates et des chaînes de caractères.

6.2 La clause `GROUP BY`

La clause `GROUP BY` permet de spécifier la manière dont doit opérer les fonctins de groupe.

Exemple 14. On désire obtenir le salaire moyen de chaque département. A cette fin, on utilise une clause `GROUP BY` :

```
SELECT deptno, AVG(sal)
 FROM emp
 GROUP BY deptno;
```

On peut également cumuler plus d'un champ dans la clause `GROUP BY` :

```
SELECT deptno, job, AVG(sal)
 FROM emp
 GROUP BY deptno, job;
```

Notons qu'il est également possible d'utiliser une fonction de groupe dans une clause `ORDER BY`. Alors, l'exemple précédent devient :

```
SELECT deptno, AVG(sal)
 FROM emp
 GROUP BY deptno
 ORDER BY AVG(sal);
```

6.3 La clause `HAVING` : conditions sur les groupes

Pour introduire des conditions sur les groupes, on ne peut pas écrire :

```
SELECT deptno, AVG(sal)
 FROM emp
 WHERE AVG(sal) > 2000
 GROUP BY deptno;
```

A la place, on utilise la clause `HAVING` qui permet d'introduire des conditions sur les groupes. Le bloc erronné précédent devient :

```
SELECT deptno, AVG(sal)
 FROM emp
 GROUP BY deptno
 HAVING AVG(sal) > 2000;
```

Chapitre 7

Sous-requêtes

7.1 Sous-requêtes élémentaires

```
SELECT col1, col2, ...
 FROM table
 WHERE expr OPERATOR
   (SELECT col
           FROM table);
```

On peut utiliser des fonctions de groupe dans une sous-requête. De même on peut utiliser une sous-requête dans une clause `HAVING`.

Exemple 15. On désire avoir les noms des employés dont le salaire excède le salaire moyen :

```
SELECT ename FROM emp
        WHERE sal > (SELECT AVG(sal) FROM emp);
```

Opérateurs de comparaison sur plusieurs rangées

Une sous-requête est invalide à partir du moment où on veut comparer un élément à un ensemble d'éléments. De même, si aucune rangée n'est retournée par la sous-requête, la sous-requête est invalide.

Exemple 16. La sous-requête suivante est fausse car on compare à un nombre une table (qui contient plus d'un élément) :

```
SELECT ename FROM emp
        WHERE sal > (SELECT AVG(sal)
                            FROM emp
                            GROUP BY deptno);
```

Notons que la sous-requête de l'exemple précédent retourne plusieurs rangées. Les opérateurs `IN`, `ANY`, `ALL` peuvent être employés, il s'agit de comparateurs de comparaison sur plusieurs rangées. Il est possible de combiner `NOT` avec ces trois opérateurs. De plus, ils doivent toujours se trouver à gauche de la sous-requête (autrement dit, la sous-requête doit être à droite de l'opérateur).

- `ALL` : la condition est vraie si elle est vraie pour toutes les valeurs retournées.
- `ANY` : la condition est vraie si elle est vraie pour au moins une des valeurs retournées.
- `IN` : a le même fonctionnement que lorsqu'utilisé sur des listes ; retourne vrai si l'expression à gauche correspond à une des valeurs retournées.

Exemple 17. On désire une liste des employés dont le salaire excède le salaire moyen de chaque département :

```
SELECT ename, sal
  FROM emp
        WHERE sal > ALL (SELECT AVG(sal)
                                FROM emp
                                GROUP BY deptno);
```

7.2 Sous-requêtes complexes

7.2.1 Conditions sur plusieurs colonnes

Il est possible d'avoir des conditions qui portent sur un groupe de colonnes. On emploie dès lors une sous-requête à colonnes multiples. La syntaxe est la suivante :

```
SELECT col1, col2, ...
 FROM table
 WHERE (col, col, ...) OPERATOR
   (SELECT col, col, ...
           FROM table
                 [WHERE condition]);
```

Exemple 18. On désire obtenir la liste des employés qui ont le même travail et département que Rogers.

```
SELECT ename
  FROM emp
        WHERE (job, deptno) = (SELECT job, deptno
                                FROM emp
                                WHERE ename = 'ROGERS');
```

7.2.2 Valeurs nulles dans une sous-requête

A compléter

7.2.3 Sous-requête dans une clause `FROM`

Il est possible d'utiliser une sous-requête dans une clause `FROM`. Considérons l'exemple suivant :

Exemple 19. On désire le nom des employés qui sont payés mieux que la moyenne de leur département. Pour ce faire, on emploie dans la clause `FROM` une sous-requête permettant

d'obtenir la table qui contient le numéro des département et leur salaire moyen. Notons l'emploi d'alias pour référencer la table "fabriquée" dans la clause `FROM` ainsi que pour ses colonnes.

```
SELECT a.ename, a.sal, a.deptno, b.salavg
  FROM emp a (SELECT deptno, AVG(sal) salavg
              FROM emp
              GROUP BY deptno) b
  WHERE a.deptno = b.deptno
  AND a.sal>b.salavg
```

Lorsqu'on référence une telle table, il est important de donner des alias à ses attributs qui n'ont pas de noms (tels que les résultats de fonctions de groupe ou de l'évaluation d'une expression). Bien sûr, il faut également un alias pour la table même.

Chapitre 8

Manipulation de données

Les instructions DML (Data Manipulation Language) permettent les choses suivantes :

- Ajouter des rangées à une table ;
- Enlever des rangées à une table ;
- Modifier des rangées.

8.1 Insertion de données

Il est possible d'insérer des données de deux manières : en spécifiant explicitement les noms de colonnes ou en fournissant un nombre d'arguments qui correspond au nombre d'attributs (de colonnes) de la table modifiée.

```
INSERT INTO nomTable [(col [, col ...])]
  VALUES (value [, value ...]);
```

Exemple 20. On désire ajouter un nouveau département à notre base de donnée. On peut le faire en spécifiant les noms de colonnes :

```
INSERT INTO dept (deptno, dename, loc)
  VALUES (50, 'DEVELOPMENT', 'DETROIT');
```

Ou alors en donnant trois arguments dont le type correspond à celui des attributs :

```
INSERT INTO dept
  VALUES (50, 'DEVELOPMENT', 'DETROIT');
```

Pour insérer des valeurs nulles, il suffit d'omettre le nom de la colonne dans la liste (si on suit la première manière de faire) ou alors simplement de spécifier `NULL` (il faut éviter de mettre des guillemets ou apostrophes, sous peine d'ajouter une chaîne de caractère qui correspond à "NULL"). Il est également possible d'ajouter des valeurs de fonctions, comme `sysdate()`.

On peut copier des données d'une autre table grâce à une sous-requête. Dans ce cas, le mot-clef `VALUES` n'est pas spécifié.

Exemple 21. Copier dans une table MANAGERS les employés qui occupent ce poste :

```
INSERT INTO MANAGER (id, name, salary, hiredate)
  SELECT empno, ename, sal, hiredate
        FROM emp
        WHERE job='MANAGER';
```

8.2 Mise à jour des données d'une table

Pour modifier les données d'une (ou plusieurs) rangées, on emploie la clause `UPDATE` avec la syntaxe suivante :

```
UPDATE nomTable
  SET col = value [, col = value, ...]
        WHERE [condition];
```

Si la condition est omise, toutes les rangées sont modifiées.

Exemple 22. On désire transférer l'employé dont le numéro d'identification est 7865 au département 20 :

```
UPDATE emp
  SET deptno = 20
        WHERE empno = 7865;
```

Il n'est pas toujours possible d'utiliser des sous-requêtes avec MySQL. En particulier, on ne peut pas utiliser une sous-requête dans une clause `UPDATE` si la table de la clause `FROM` de la sous-requête est cette même table. Donnons un faux-exemple :

Exemple 23. L'erreur sus-mentionnée arrive dans les cas similaires à :

```
UPDATE t1 SET col2 = (SELECT MAX(t1) FROM t1);
```

Une requête similaire déclenche en MySQL l'erreur 1093.

8.3 Suppression des données d'une table

Pour supprimer une (ou plusieurs) rangées, on emploie la clause `DELETE` avec la syntaxe suivante :

```
DELETE nomTable
        WHERE [condition];
```

Si la conditions est omise, toutes les rangées sont supprimées. Il n'est pas possible de supprimer une rangée qui sert de clef étrangère dans une autre table.

8.4 Transactions

- Collection d'instructions DML qui forment une unité logique ;
- Instruction DDL (Data Definition Language) ;

- Instruction DCL (Data Control Language).

Ces intructions commencent dès la première instruction SQL exécutable. Elle se terminent après un `COMMIT` (les modifications sont enregistrées) ou un `ROLLBACK` (les modifications depuis le point de sauvegarde ou depuis le dernier enregistrement sont annulées). Les instructions DDL et DCL ont un `COMMIT` automatique et terminent donc également les transactions. Les deux dernières raisons de termination de transaction sont la sortie de l'application par l'utilisateur ou un éventuel crash système.

MySQL est lancé par défaut en mode `AUTOCOMMIT`. Chaque modification est enregistrée immédiatement sur le disque par MySQL. Il est possible de configurer MySQL en mode non-autocommit, si des tables qui supportent des transactions sont employées (par exemple InnoDB, BDB). Pour ce faire, il faut employer la commande `SET AUTOMMIT=0`.

- `COMMIT` : termine la transaction en rendant les modifications permanentes ;
- `SAVEPOINT` nom : crée un repère dans la transaction.
- `ROLLBACK` : termine la transaction en supprimant les modifications. S'il est précisé un point de sauvegarde (`ROLLBACK TO SAVEPOINT nom`, où nom désigne le nom du point de sauvegarde), les modifications sont annulées jusqu'à ce repère.

Avant un `COMMMIT` et `ROLLBACK`, il est possible de revenir à l'état précédent et de vérifier es résultats des instructions DML à l'aide de la clause `SELECT`. De plus, les modifications ne sont pas visibles aux autres utilisateurs et les rangées affectées par les modifications ne peuvent être modifiées par d'autres utilisateurs.

Chapitre 9

Création et gestion de tables

9.1 Création d'une table

Le nom d'une table doit vérifier les propriétés suivantes :

- Le nome doit commencer par une lettre ;
- Le nom doit avoir un et soixante-quatre caratères parmi A-Z, a-z, 0-9, $, #, ... ;
- Le nom ne doit correspondre à aucun mot réservé.

La syntaxe pour créer une table est la suivante :

```
CREATE TABLE [schema] nomTable
   (nomColonne1 typeDeDonnees1 [DEFAULT valeurParDefaut] [,
      ...])
        [ENGINE = innodb];
```

Le schéma est à spécifier si la table est créée pour un autre utilisateur. La valeur par défaut spécifiée est utilisé dans les `INSERT`. En MySQL, il doit s'agir d'une constante et non pas d'une fonction ni d'une colonne.

Exemple 24. On désire créer une table dans laquelle on va stocker les département de notre société, plus précisément leur nom, numéro et localisation.

```
CREATE TABLE dept
  (deptno INTEGER(2),
        dname VARCHAR(14),
        loc VARCHAR(13));
```

Introduisons quelques types de données récurrents (il en existe une pléthore d'autres) :

- `INTEGER(p)` : un entier de p chiffres ;
- `DECIMAL(p, s)` : un réel de p chiffres et de s décimales ;
- `VARCHAR(s)` : chaîne de longueur variable de maximum s caractères ;
- `CHAR(s)` : chaîne de longueur fixe, de s caractères ;
- `DATE` : représente une date.

Il est également possible de créer une table à partir d'une sous-requête. La syntaxe est la suivante :

```
CREATE TABLE nomTable [ENGINE=innodb] AS sousRequete;
```

Les valeurs par défaut ne sont pas transmises de cette façon.

Le nom donné aux colonnes de la nouvelle table sont ceux des alias de la sous-requête. Illustrons cela par un exemple :

Exemple 25. À partir d'une table d'employés appelé emp, on désire créer une table EMPLOYEE qui est une copie de la table emp, mais dont les noms de colonne sont différents.

```
CREATE TABLE EMPLOYEE ENGINE=innodb AS
  SELECT ename Name, empno id, deptno Dep
        FROM emp;
```

Les colonnes de la table créée seront appelées Name, id et Dep, conformément aux alias utilisés.

9.2 Modification d'une table

Il est possible d'ajouter une colonne à une table :

```
ALTER TABLE nomTable
  ADD nomColonne typeDonnee [DEFAULT valeur] [,...];
```

Il est également possible de modifier le type de valeur que prend une colonne :

```
ALTER TABLE table
  MODIFY nomColonne typeDonnee [DEFAULT valeur] [, ...];
```

Exemple 26. On désire faire passer le nombre de caractères maximal possible pour le nom d'un employé à 20.

```
ALTER TABLE EMPLOYEE
  MODIFY Name VARCHAR(20);
```

9.3 Suppression de tables et de leur contenu

9.3.1 L'instruction `DROP TABLE`

Cette instruction supprime une table de la base de données. Il faut faire attention parce que les tables supprimées de la sorte sont irrécupérables (à cause du `COMMIT` implicite aux commandes de type DDL).

```
DROP TABLE nomTable;
```

9.3.2 L'instruction `TRUNCATE`

Cette instruction vide une table de son contenu. Encore une fois, suite au `COMMIT` implicite de cette commande, les données supprimées de cette manière sont irrécupérables.

```
TRUNCATE TABLE nomTable;
```

Exemple 27. Supposons que l'instruction `AUTOCOMMIT=0` ait été entrée. Supposons également que nous disposons d'une table emp qui contient 17 employés et d'une table dept qui contient 4 départements. Combien d'entrées sont affichées au terme de ces instructions ?

```
TRUNCATE TABLE emp;
DELETE FROM dept;
ROLLBACK;
SELECT * FROM emp;
SELECT * FROM dept;
```

Dans ce premier cas, seules les quatre entrées de la table dept seront affichées. En effet, la suppression des données de emp par l'instruction `TRUNCATE` est définitive mais la supression des données de dept par l'instruction `DELETE` est annulée par le `ROLLBACK`.

On se pose la même question mais face à cet exemple (on suppose à nouveau que emp contient 17 rangées et dept quatre) :

```
DELETE FROM emp;
TRUNCATE TABLE dept;
ROLLBACK;
SELECT * FROM emp;
SELECT * FROM dept;
```

Dans ce second cas, aucune entrée n'est affichée. En effet, la suppression par le `DELETE` est rendue définitive par le `COMMIT` implicite dans le `TRUNCATE` de la ligne suivante.

Chapitre 10

Contraintes

10.1 Les différentes contraintes

Les contraintes sont des règles concernant les champs d'une table. S'il y a dépendance d'une table avec une autre (par exemple par une clef étrangère), ces contraintes empêchent d'effacer une table ou une rangée. Elles sont au nombre de quatre :

- `NOT NULL` : empêche le champ affecté par cette contrainte de prendre la valeur `NULL`.
- `UNIQUE` : oblige les valeurs dans une colonne (ou une combinaison de colonnes si la contrainte est posée sur plusieurs colonnes à la fois) à être deux-à-deux distinctes.
- `PRIMARY KEY` : combine les contraintes `NOT NULL` et `UNIQUE`.
- `FOREIGN KEY` : oblige les valeurs dans un champ à correspondre à une clef primaire (ou à une (des) colonne(s) sous la contrainte `UNIQUE`) d'une table de référence. On parle de clef étrangère ou de clef extérieure en français.

Pour voir les contraintes sur les tables, il faut consulter la table `information_schema.table_constrain` non contenue dans la base de données mais dans `information`.

10.2 Définition de contraintes

10.2.1 A la création d'une table

Il est possible de préciser deux types de contraintes ; des contraintes au niveau d'une colonne ou au niveau de la table. La syntaxe est la suivante :

```
CREATE TABLE [schema] nomTable
  (nomColonne typeDonnee [DEFAULT valeur][contrainteColonne],
        [...]
        contrainteTable [, ...]);
```

La syntaxe pour une contrainte de colonne est :

```
nomColonne [CONSTRAINT nomContrainte] typeContrainte
```

La contrainte `NOT NULL` ne peut être spécifiée qu'au niveau d'une colonne.

La syntaxe pour une contrainte au niveau de la table est :

```
[CONSTRAINT nomContrainte] typeContrainte(colonne, ...)
```

La contrainte `UNIQUE` n'empêche pas la valeur d'être nulle. Elle ainsi que les contraintes de type `KEY` doivent être spécifée au niveau de la table.

Donnons un exemple pour illustrer ce qui précède.

Exemple 28. On désire créer une table d'employés qui seront distingués par un numéro (num), leur nom (name), leur poste (job) et le numéro de leur département (deptno). Un employé est caractérisé par son numéro (il s'agira de la clef primaire) et doit posséder un numéro de département (il ne peut être nul).

```
CREATE TABLE employee
  (num INTEGER(5),
        name VARCHAR(12),
        job VARCHAR(12),
        mgr INTEGER(5),
        deptno INTEGER(2) NOT NULL,
        CONSTRAINT emp_num_pk PRIMARY KEY(num));
```

10.2.2 Ajout de contraintes

Il est possible d'ajouter des contraintes à une table. La syntaxe est la suivante :

```
ALTER TABLE nomTable
  ADD [CONSTRAINT nomContrainte] typeContrainte [(colonne,
     ...)];
```

Contrairement aux autres contraintes de table, la contrainte `NOT NULL` doit être ajoutée de la manière suivante, en respécifiant le type de données de la colonne :

```
ALTER TABLE nomTable
  MODIFY nomColonne typeDonnee [DEFAULT valeur] NOT NULL;
```

Il ne faut pas oublier de spécifier à nouveau la valeur par défaut, le cas échéant, cette propriété est perdue. Si la table contient déjà des données, alors il ne sera pas possible de spécifier `NOT NULL`. Cela est dû au fait qu'il pourrait y avoir des problèmes si la table contient déjà des valeurs nulles.

La clause `ALTER TABLE` est utilisée, pour rappel, afin d'ajouter, retirer ou modifier des colonnes d'une table existante. C'est cette même clause qui est utilisée pour ajouter ou laisser tomber des contraintes à une table existante.

10.3 La contrainte `FOREIGN KEY`

La contrainte `FOREIGN KEY` a des mots-clefs supplémentaires. La syntaxe est la suivante :

```
CONSTRAINT nomContrainte FOREIGN KEY (colonne) REFERENCES
   nomTableRef (colRef) [ON DELETE CASCADE]
```

Si `ON DELETE CASCADE` est spécifié, alors si une des rangées de la table *référencée* est supprimée, alors toutes celles qui contiennent la valeur référencée sont supprimées. Si `ON DELETE CASCADE` n'est pas spécifié, alors la suppression d'une rangée de la table référencée alors qu'au moins une des rangées de la table (qui référence) référence la rangée en question est interdit.

Exemple 29. Imaginons qu'on ajoute cette ligne au code de l'exemple 28 :

```
CONSTRAINT employee_deptno_fk FOREIGN KEY (deptno)
   REFERENCES dept(deptno);
```

Ici, `ON DELETE CASCADE` n'a pas été spécifié. Il est donc impossible de supprimer un département tant qu'il y a encore un employé qui y appartient. Si `ON DELETE CASCADE` avait été spécifié, la suppression d'un département entraînerait la suppression de tous les employés qui y travaillent de la base de données.

Il est possible d'utiliser qu'une table se référence elle-même via une clef étrangère. Observons l'exemple suivant :

Exemple 30. On dispose d'une table d'employés dans laquelle les managers sont spécifiés. En particulier, un manager est un employé ; on utilise une clef étrangère pour s'assurer que le manager existe déjà dans la liste d'employés. Remarquons que la valeur pour le manager peut être nulle (notamment dans le cas du chef de l'entreprise).

```
ALTER TABLE employee
  ADD CONSTRAINT employee_mgr_fk FOREIGN KEY (mgr)
     REFERENCES employee(num);
```

10.4 Suppression de contraintes

Tout comme l'ajout de contraintes à une table déjà créée, il faut utiliser la clause `ALTER TABLE`. La syntaxe est la suivante :

```
ALTER TABLE nomTable
  DROP [CONSTRAINT] typeContrainte nomContrainte;
```

Dans le cas particulier de la clef primaire, le nom de la contrainte ne doit pas être spécifié. S'il existe une clef étrangère référençant cette clef primaire, la syntaxe précédente échouera. Il faut spécifier en plus le mot-clef `CASCADE` :

```
ALTER TABLE nomTable
  DROP PRIMARY KEY CASCADE;
```

Ceci `"DROP"` toute clef étrangère référençant la clef primaire supprimée.

Exemple 31. On désire enlever la contrainte de clef primaire sur la table des départements dept, toutefois l'usage des numéros de département comme clef étrangère (exemple 30) empêche cette suppression. On emploie alors `CASCADE` :

```
ALTER TABLE dept
  DROP PRIMARY KEY CASCADE;
```

La contrainte sur les numéros de département de la table des employés tombe également en conséquence.

Chapitre 11

Création de vues

11.1 Qu'est-ce qu'une vue ?

Une vue (`VIEW`) est une représentation de sous-ensembles de données d'une ou de plusieurs tables. Une vue est stockée sous la forme d'une instruction `SELECT` dans une table.Le buts de ce mécanisme est de restreindre l'acccès à certaines données, de simplifier les requêtes complexes et marquer une indépendance entre les données et les applications. On emploie la syntaxe suivante :

```
CREATE [OR REPLACE]
  VIEW nomVue [(alias1, ...)]AS sousRequete;
```

La sous-requête ne peut pas contenir de clause `ORDER BY`. Les mots-clefs `OR REPLACE` servent à recréer la vue si elle existe déjà. Les alias sont les noms attribués aux colonnes de la vue.

11.2 Quelques exemples de vues

On parle de vue simple lorsque la vue ne contient qu'une table, aucune fonction, aucun groupe.

Exemple 32. On désire créer une vue pour accéder rapidement à la liste des employés du dixième département.

```
CREATE  VIEW empdp10 AS
  SELECT num, name, job
  FROM employee
  WHERE deptno = 10;
```

On parle de vue complexe dans les autres cas :

Exemple 33. On désire créer une vue pour accéder rapidement aux salaires de chaque département.

```
CREATE  VIEW dept_avg (name, minsal, maxsal, avgsal)
  AS SELECT d.name, MIN(e.sal), MAX(e.sal), AVG(e.sal)
  FROM employee e, dept d
  WHERE e.deptno = d.deptno;
        GROUP BY d.name;
```

11.3 Instructions DML sur une vue

Les instructions DML sont toujours possibles sur des vues simples.
Il n'est pas possible d'enlever des données si une des conditions suivantes est vérifiée :

- la vue contient des fonctions de groupe ;
- la vue contient une clause `GROUP BY` ;
- la vue contient le mot-clef `DISTINCT`.

Il n'est pas possible de modifier des données si :

- une des conditions plus haut est vérifiée ;
- la vue contient des tables définies par des expressions.

Il n'est pas possible d'ajouter des données si :

- Une des conditions plus haut est vérifiée ;
- des colonnes avec la contrainte `NOT NULL` se trouvent dans la table de base mais sont absentes de la vue.

11.4 Suppression de vues

La syntaxe est la suivante :

```
DROP VIEW nomVue;
```

Chapitre 12

Index

Un index améliore la performance de certaines requêtes. il est utilisé par le serveur pour accélérer la recherche de rangées. L'index est maintenu par le serveur et indépendant de la base indexée.

Une index est créé automatiquement via les contraintes `PRIMARY KEY` et `UNIQUE`. Pour créer un index manuellement, la syntaxe est la suivante :

```
CREATE INDEX nomIndex
ON nomTable (nomColonne [, nomColonne2, ...]);
```

La syntaxe pour la suppression d'index est la suivante :

```
DROP INDEX nomIndex;
```

Il vaut mieux créer un index dans les cas suivants :

- Lorsqu'une colonne est utilisée fréquemment dans une clause `WHERE` ou pour joindre des tables ;
- Sur une colonne qui contient un grand éventail de valeurs, des valeurs nulles ;
- Lorsque la table est grande ;
- Lorsque les requête usuelles ne retournent que peu de rangées ;
- Les mises à jour de la table sont peu fréquentes (car une mise à jour de la table implique une mise à jour de l'index).

Un index sur une colonnes peu fréquemment utilisée est inutile.

Chapitre 13

Sécurité

Distinguons le point de vue administrateur (création d'utilisateurs, backups, etc.) et d'utilisateurs (gestion de l'accès à ses propres objets). Les utilisateurs ont certains privilèges, notamment la création d'objets (tables, vues, etc.) et l'accès aux objets (mises à jours, etc.). Il est nécessaire d'utiliser un mot de passe pour accéder à la base de données.

La commande à utiliser pour modifier le mot de passe est :

```
SET PASSWORD [FOR USER] = PASSWORD('motDePasse');
```

La syntaxe pour accorder des privilèges objets est la suivante :

```
GRANT privilege [(colonnes)]
ON objet
TO utilisateur [IDENTIFIED BY 'password']
[WITH GRANT OPTION];
```

Les privilèges qui peuvent être accordés sont, entre autres `SELECT`, `UPDATE`, `CREATE VIEW` (etc.). Si `ON GRANT OPTION` est spécifié, l'utilisateur auquel les privilèges ont été accordés peut les transmettre à d'autres. Les colonnes ne doivent être précisées que si on veut spécifier que le privilège ne porte que sur elles *(si quelqu'un sait préciser l'utilité de ce colonnes, je serais reconnaissant).*

Exemple 34. On désire donner à l'utilisateur *rain* le privilège de `SELECT`, `INSERT` et `UPDATE` sur la table employee. On désire également qu'il puisse transférer ces privilèges.

```
GRANT SELECT, INSERT, UPDATE
ON employee
TO rain
WITH GRANT OPTION;
```

Si on veut limiter les colonnes auxquelles une personne a accès, il faut utiliser une vue.

Exemple 35. On veut donner à l'utilisateur *alys* un accès partiel à la table employee (aux numéros et aux noms des employés). Il faut d'abord créer une vue pour ce faire :

```
CREATE VIEW empview
(name, number) AS SELECT name, num FROM employee;
```

```
GRANT SELECT
ON empview
TO alys;
```

Exemple 36. *rain* veut transmettre tous ses privilèges à *bob*. Plutôt que de les spécifier un à un, elle écrit :

```
GRANT ALL PRIVILEGES
ON employee
TO bob;
```

Notons que seuls les privilèges qu'a *rain* sont transmis. La syntaxe fonctionne également pour transmettre tous les privilèges (dans la mesure où le transmetteur les a tous).

Notons que si les tables sont dans une base de données différentes, il est nécessaire de changer de base pour y accéder (`use` *`nomBaseDonnees;`*)

Pour révoquer les privilèges transmis, il faut utiliser une clause `REVOKE` :

```
REVOKE privilege1 [, privilege2, ...]
ON objet
FROM nomUtilisateur;
```

Avec `ALL`, on peut enlever tous les privilèges accordés à une personne. La suppression de privilège accordés entraîne la suppression des privilèges transmis.

Chapitre 14

Sous-requêtes correlées et opérateurs ensemblistes

14.1 Opérateurs ensemblistes

Les opérateurs d'ensemble sont au nombre de trois. Contrairement à `UNION` (et `UNION ALL` qui autorise les doublons), les opérateurs `INTERSECT` et `MINUS` ne sont pas implémentés. Il se pourrait qu'à leur implémentation, ils portent d'autres noms que ceux-là.

- `UNION` : il permet l'union de deux tables dont les colonnes contiennent le même type de données.
- `MINUS` : il permettrait d'obtenir la première table à laquelle on aurait enlevé les éléments apparaissant dans la seconde. Il serait nécessaire d'avoir des colonnes qui contiennent le même type de données.
- `INTERSECT` : il permettrait d'obtenir l'intersection de deux tables (dont les colonnes contiennent le même type de données).

Exemple 37. Supposons qu'on aie à notre disposition une table *emp_hist* qui contient l'historique de tous les employés. Afin d'avoir une liste complète de tous les employés, on peut exécuter la requête suivante :

```
SELECT ename, job, deptno
FROM emp
UNION
SELECT name, title, deptid
FROM emp_history;
```

En remplaçant dans le code précédent `UNION` par `UNION ALL`, les employés qui sont dans l'entreprise apparaîtront deux fois (puisqu'ils sont également enregistrés dans l'historique). Si on remplace `UNION` par `MINUS` (ce qui n'est pas possible en pratique), on obtiendrai la liste des employés qui ont quitté l'entreprise.

Lors de l'emploi des opérateurs d'ensemble, il faut retenir les choses suivantes :

- Les noms de colonnes affichés sont ceux de la première requête ;
- Les expressions dans les listes doivent correspondre en type et en nombre ;
- Les doublons sont ignorés, sauf dans le cas de `UNION ALL` ;
- Une clause `ORDER BY` ne peut apparaître qu'à la fin de la requête ;

- Il est possible de combiner les opérateurs (il est recommandé d'utiliser des parenthèses) ;
- Il est possible de laisser des champs vides (cf exemple 38.

Exemple 38. On désire lister la date d'embauche des employés de chaque département avec la localisation des départements. Afin d'éviter de répéter la localisation, on utilise l'opérateur `UNION` en laissant dans un `SELECT` un blanc (une chaîne d'un espace) au lieu de la localisation et dans l'autre on fait de même mais pour la date d'embauche.

```
SELECT deptno, " " location, hiredate FROM emp
UNION
SELECT deptno, hiredate, " " FROM emp
ORDER BY deptno, hiredate;
```

Ainsi seront affichés d'abord la ligne avec la localisation puis toutes les dates, sans que soit répétée la localisation.

14.2 Sous-requête corrélées

La sous-requête est exécutée pour chaque rangée de la requête principale. Pour cela, on fait usage d'alias de tables dans la requête principal qu'on emploie ensuite dans la sous-requête. Illustrons cela par un exemple :

Exemple 39. On désire lister les employés qui gagnent plus que le salaire moyen dans leur département.

```
SELECT empno, sal, deptno
FROM emp e
WHERE sal > (SELECT AVG(sal) FROM emp i
  WHERE e.deptno = i.deptno);
```

L'opérateur `EXISTS`

Si la sous-requête contient une rangée, alors elle s'arrête et retourne vrai. Sinon la condition est fausse.

Exemple 40. On désire lister les employés qui ont des employés sous leurs ordres.

```
SELECT empno, deptno
FROM emp e
WHERE EXISTS (SELECT * FROM emp i
  WHERE e.empno = i.mgr);
```

`UPDATE` corrélé

Pour ceci, contentons-nous d'un exemple :

Exemple 41. Ajoutons une colonne *dname* dans la table *emp* (en employant le champ correspondant dans *dept*) :

```
ALTER TABLE emp
ADD (dname VARCHAR(14));
/* ajout d'une nouvelle colonne*/

UPDATE emp e
SET dname = (SELECT dname FROM dept
  WHERE e.deptno=d.deptno);
```