# Administration BDD avec MySQL

## Les débuts avec MySQL

### Installer MySQL Server

**[A COMPLETER]**

### Se connecter

#### Connection

```
> mysql -h localhost -u root -pmotdepassetopsecret

# ou

> mysql --host=localhost --user=root --password=motdepassetopsecret

# ou un mélange des paramètres courts et longs si ça vous amuse

> mysql -h localhost --user=root -pmotdepassetopsecret

# ou

> mysql -h localhost -u root -p
Enter password :
```

#### Déconnection

```
> quit

# ou

> exit
```

### Syntaxe SQL et premières commandes

#### "Hello World !"

```
SELECT 'Hello World !';
```

#### Syntaxe

Pour signifier à MySQL qu'une instruction est terminée, il faut mettre le caractère `;` . Tant qu'il ne rencontre pas ce caractère, le client MySQL pense que vous n'avez pas fini d'écrire votre commande et attend gentiment que vous continuiez.

Ce caractère de fin d'instruction obligatoire va vous permettre :

- d'écrire une instruction sur plusieurs lignes ;
- d'écrire plusieurs instructions sur une seule ligne.

#### Commentaires

En SQL, les commentaires sont introduits par `--` (deux tirets). Cependant, MySQL déroge un peu à la règle SQL et accepte deux syntaxes :

- `#` : tout ce qui suit ce caractère sera considéré comme commentaire.
- `--` : la syntaxe normale est acceptée uniquement si les deux tirets sont suivis d'une espace au moins.

**Chaînes de caractères**

Lorsque vous écrivez une chaîne de caractères dans une commande SQL, il faut absolument l'entourer de guillemets simples (donc des apostrophes).

```
SELECT 'Bonjour les B3 !';
```

Si vous désirez utiliser un caractère spécial dans une chaîne, il vous faudra l'échapper avec `\` . Par exemple, si vous entourez votre chaîne de caractères de guillemets simples, mais voulez utiliser un tel guillemet à l'intérieur de votre chaîne :

```
SELECT 'Salut l'ami';  -- Pas bien !
```

```
SELECT 'Salut l\'ami'; -- Bien !
```

**Quelques autres caractères spéciaux :**

| Caractère | Signification |
|---|---|
| \n | retour à la ligne |
| \t | tabulation |
| \ | antislash |
| % | pourcent |
| _ | souligné |

Notez que, pour échapper un guillemet simple (et uniquement ce caractère), vous pouvez également l'écrire deux fois. Cette façon d'échapper les guillemets correspond d'ailleurs à la norme SQL. Je vous encourage par conséquent à essayer de l'utiliser au maximum.

```
SELECT 'Salut l'ami';  -- ne fonctionne pas !
```

```
SELECT 'Salut l\'ami'; -- fonctionne !
```

```
SELECT 'Salut l''ami'; -- fonctionne aussi et correspond à la norme !
```

**Un peu de math**

```
mysql> SELECT (5+3)*2;

+---------+
| (5+3)*2 |
+---------+
|      16 |
```

```
+---------+
1 row in set (0.00 sec)
```

MySQL est sensible à la priorité des opérations, comme vous pourrez le constater en tapant cette commande :

```
mysql> SELECT (5+3)*2, 5+3*2;
+---------+-------+
| (5+3)*2 | 5+3*2 |
+---------+-------+
|      16 |    11 |
+---------+-------+
1 row in set (0.00 sec)
```

### Utilisateur

Tapez ces commandes dans mysql, en remplaçant `student` par le nom d'utilisateur que vous avez choisi, et `mot_de_passe` par le mot de passe que vous voulez lui attribuer :

```
CREATE USER 'student'@'localhost' IDENTIFIED BY 'mot_de_passe';
GRANT ALL PRIVILEGES ON bachelor3.* TO 'student'@'localhost';
```

**Explications**

- `CREATE USER 'student'` : cette commande crée l'utilisateur student.
- `@'localhost'` : définit à partir d'où l'utilisateur peut se connecter. Dans notre cas, 'localhost', donc il devra être connecté à partir de cet ordinateur.
- `IDENTIFIED BY 'mot_de_passe'` : définit le mot de passe de l'utilisateur.
- `GRANT ALL PRIVILEGES` : cette commande permet d'attribuer tous les droits (c'est-à-dire insertions de données, sélections, modifications, suppressions…).
- `ON bachelor3.*` : définit les bases de données et les tables sur lesquelles ces droits sont acquis. Donc ici, on donne les droits sur la base "bachelor3" (qui n'existe pas encore, mais ce n'est pas grave, nous la créerons plus tard), pour toutes les tables de cette base (grâce à *).
- `TO 'student'@'localhost'` : définit l'utilisateur (et son hôte) auquel on accorde ces droits.

Pour vous connecter à mysql avec ce nouvel utilisateur, il faut donc taper la commande suivante (après s'être déconnecté, bien sûr) :

```
mysql -u student -p
```

### Encodage, jeux de caractères et interclassement

**[A COMPLETER]**

---

## Distinguez les différents types de données

Nous avons vu dans l'introduction qu'une base de données contenait des tables, elles-mêmes organisées en colonnes, dans lesquelles sont stockées des données.

En SQL (et dans la plupart des langages informatiques), les données sont séparées en plusieurs types (par exemple : texte, nombre entier, date…). Lorsque l'on définit une colonne dans une table de la base, il faut donc lui donner un type, et toutes les

données stockées dans cette colonne devront correspondre au type de la colonne. Nous allons donc voir les différents types de données existant dans MySQL.

### Types numériques

On peut subdiviser les types numériques en deux sous-catégories : les nombres entiers, et les nombres décimaux.

### Nombres entiers

Les types de données qui acceptent des nombres entiers comme valeurs sont désignés par le mot-clé `INT`, et ses déclinaisons `TINYINT`, `SMALLINT`, `MEDIUMINT` et `BIGINT`. La différence entre ces types est le nombre d'octets (donc la place en mémoire) réservés à la valeur du champ. Voici un tableau reprenant ces informations, ainsi que l'intervalle dans lequel la valeur peut être comprise pour chaque type :

| Type | Nombre d'octets | Minimum | Maximum |
|---|---|---|---|
| TINYINT | 1 | -128 | 127 |
| SMALLINT | 2 | -32768 | 32767 |
| MEDIUMINT | 3 | -8388608 | 8388607 |
| INT | 4 | -2147483648 | 2147483647 |
| BIGINT | 8 | -9223372036854775808 | 9223372036854775807 |

> *Si vous essayez de stocker une valeur en dehors de l'intervalle permis par le type de votre champ, MySQL stockera la valeur la plus proche. Par exemple, si vous essayez de stocker 12457 dans un `TINYINT`, la valeur stockée sera 127 ; ce qui n'est pas exactement pareil. Réfléchissez donc bien aux types de vos champs.*

#### L'attribut `UNSIGNED`

Vous pouvez également préciser que vos colonnes sont `UNSIGNED`, c'est-à-dire que l'on ne précise pas s'il s'agit d'une valeur positive ou négative (on aura donc toujours une valeur positive). Dans ce cas, la longueur de l'intervalle reste la même, mais les valeurs possibles sont décalées, le minimum valant 0. Pour les `TINYINT`, on pourra par exemple aller de 0 à 255.

#### Limiter la taille d'affichage et l'attribut `ZEROFILL`

Il est possible de préciser le nombre de chiffres minimum **à l'affichage** d'une colonne de type `INT` (ou un de ses dérivés). Il suffit alors de préciser ce nombre entre parenthèses : `INT(x)`.

Cette taille d'affichage est généralement utilisée en combinaison avec l'attribut `ZEROFILL`. Cet attribut ajoute des zéros à gauche du nombre lors de son affichage, il change donc le caractère par défaut par '0'. Donc si vous déclarez une colonne comme étant :

```
INT(4) ZEROFILL
```

Vous aurez l'affichage suivant :

| Nombre stocké | Nombre affiché |
|---|---|

| | |
|---|---|
| 45 | 0045 |
| 4156 | 4156 |
| 785164 | 3785164 |

### Nombres décimaux

Cinq mots-clés permettent de stocker des nombres décimaux dans une colonne :

- `DECIMAL`,
- `NUMERIC`,
- `FLOAT`,
- `REAL`,
- `DOUBLE`.

#### NUMERIC et DECIMAL

`NUMERIC` et `DECIMAL` sont équivalents et acceptent deux paramètres : la **précision** et **l'échelle**.

- La précision définit le nombre de chiffres significatifs stockés, donc les 0 à gauche ne comptent pas. En effet, 0024 est équivalent à 24. Il n'y a donc que deux chiffres significatifs dans 0024.
- L'échelle définit le nombre de chiffres après la virgule.

Dans un champ `DECIMAL(5,3)`, on peut donc stocker des nombres de 5 chiffres significatifs au maximum, dont 3 chiffres sont après la virgule.

*Par exemple : 12.354, -54.258, 89.2 ou -56.*

> *`DECIMAL(4)` équivaut à écrire `DECIMAL(4, 0)`*

#### FLOAT, DOUBLE et REAL

Le mot-clé `FLOAT` peut s'utiliser sans paramètres, auquel cas 4 octets sont utilisés pour stocker les valeurs de la colonne. Il est cependant possible de spécifier une précision et une échelle, de la même manière que pour `DECIMAL` et `NUMERIC`.

Quant à `REAL` et `DOUBLE`, ils ne supportent pas de paramètres. `DOUBLE` est normalement plus précis que `REAL` (stockage dans 8 octets, contre stockage dans 4 octets), mais ce n'est pas le cas avec MySQL qui utilise 8 octets dans les deux cas. Il est donc conseillé d'utiliser `DOUBLE` pour éviter les surprises en cas de changement de **SGBDR**.

### Types alphanumériques

#### Chaînes de type texte

#### CHAR et VARCHAR

Pour stocker un texte relativement court (moins de 255 caractères), vous pouvez utiliser les types `CHAR` et `VARCHAR`. Si vous entrez un texte plus long que la taille maximale définie pour le champ, celui-ci sera tronqué.

Petit tableau explicatif, en prenant l'exemple d'un CHAR ou d'un VARCHAR de 5 caractères au maximum :

| Texte | CHAR(5) | Mémoire requise | VARCHAR(5) | Mémoire requise |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| '' | ' ' | 5 octets | ' ' | 1 octet |
| 'tex' | 'tex ' | 5 octets | 'tex ' | 4 octets |
| 'texte' | 'texte' | 5 octets | 'texte' | 6 octets |
| 'texte trop long' | 'texte' | 5 octets | 'texte' | 6 octets |

Vous voyez donc que, dans le cas où le texte fait la longueur maximale autorisée, un `CHAR(x)` prend moins de place en mémoire qu'un `VARCHAR(x)`. Préférez donc le `CHAR(x)` dans le cas où vous savez que vous aurez toujours `x` caractères (par exemple si vous stockez un code postal). Par contre, si la longueur de votre texte risque de varier d'une ligne à l'autre, définissez votre colonne comme un `VARCHAR(x)`.

> *Dans le tableau ci-dessus, tous les caractères sont des caractères simples, sans accents. Dans un texte comprenant des accents et encodé en UTF-8, les caractères accentués occupent deux octets en mémoire. Il est donc possible, pour un CHAR(5), d'occuper plus de 5 octets en mémoire (mais impossible d'y stocker plus que 5 caractères).*

**TEXT**

Pour stocker des textes de plus de 255 caractères, il suffit d'utiliser le type `TEXT`, ou un de ses dérivés `TINYTEXT`, `MEDIUMTEXT` ou `LONGTEXT`. La différence entre ceux-ci est la place qu'ils permettent d'occuper en mémoire. Petit tableau habituel :

| Type | Longueur maximale | Mémoire occupée |
|---|---|---|
| TINYTEXT | $2^8$ octets | Longueur de la chaîne + 1 octet |
| TEXT | $2^{16}$ octets | Longueur de la chaîne + 2 octets |
| MEDIUMTEXT | $2^{24}$ octets | Longueur de la chaîne + 3 octets |
| LONGTEXT | $2^{32}$ octets | Longueur de la chaîne + 4 octets |

**Chaînes de type binaire**

Comme les chaînes de type texte que l'on vient de voir, une chaîne binaire n'est rien d'autre qu'une suite de caractères. Cependant, si les textes sont affectés par l'encodage et l'interclassement, ce n'est pas le cas des chaînes binaires. Une chaîne binaire n'est rien d'autre qu'une suite d'octets. Aucune interprétation n'est faite sur ces octets. Ceci a deux conséquences principales :

- Une chaîne binaire traite directement l'octet, et pas le caractère que l'octet représente. Donc par exemple, une recherche sur une chaîne binaire sera toujours sensible à la casse, puisque "A" (code binaire : 01000001) sera toujours différent de "a" (code binaire : 01100001).
- Tous les caractères sont utilisables, y compris les fameux caractères de contrôle non affichables définis dans la table ASCII.

Par conséquent, les types binaires sont parfaits pour stocker des données "brutes" comme des images par exemple, tandis que les chaînes de texte sont parfaites pour stocker du texte.

Les types binaires sont définis de la même façon que les types de chaînes de texte. `VARBINARY(x)` et `BINARY(x)` permettent de stocker des chaînes binaires de *x* caractères

au maximum (avec une gestion de la mémoire identique à `VARCHAR(x)` et `CHAR(x)`). Pour les chaînes plus longues, il existe les types `TINYBLOB`, `BLOB`, `MEDIUMBLOB` et `LONGBLOB`, avec les mêmes limites de stockage que les types `TEXT`.

### Types temporels

Les cinq types temporels de MySQL sont `DATE`, `DATETIME`, `TIME`, `TIMESTAMP` et `YEAR`.

#### DATE, TIME et DATETIME

Comme son nom l'indique, `DATE` sert à stocker une date. `TIME` sert quant à lui à stocker une heure, et `DATETIME` stocke une date **ET** une heure !

#### DATE

Pour entrer une date, il faut donner d'abord l'année (deux ou quatre chiffres), ensuite le mois (deux chiffres) , et, pour finir, le jour (deux chiffres), sous forme de nombre ou de chaîne de caractères.

Voici quelques exemples d'expressions correctes (A représente les années, M les mois et J les jours) :

- `AAAA-MM-JJ'` (c'est sous ce format-ci qu'une `DATE` est stockée dans MySQL)
- `'AAMMJJ'`
- `'AAAA/MM/JJ'`
- `'AA+MM+JJ'`
- `'AAAA%MM%JJ'`
- `AAAAMMJJ` (nombre)
- `AAMMJJ` (nombre)

L'année peut donc être donnée avec deux ou quatre chiffres. Dans ce cas, le siècle n'est pas précisé, et c'est MySQL qui va décider de ce qu'il utilisera, selon ces critères :

- si l'année donnée est entre 00 et 69, on utilisera le XXIe siècle, on ira donc de 2000 à 2069 ;
- par contre, si l'année est comprise entre 70 et 99, on utilisera le XXe siècle, donc entre 1970 et 1999.

> *MySQL supporte des `DATE` allant de '1001-01-01' à '9999-12-31'*

#### DATETIME

Très proche de `DATE`, ce type permet de stocker une heure, en plus d'une date. Si l'on utilise une chaîne de caractères, il faut séparer la date et l'heure par une espace.

Quelques exemples corrects (H représente les heures, M les minutes et S les secondes) :

- `'AAAA-MM-JJ HH:MM:SS'` (c'est sous ce format-ci qu'un DATETIME est stocké dans MySQL)
- `'AA*MM*JJ HH+MM+SS'`
- `AAAAMMJJHHMMSS` (nombre)

> *MySQL supporte des `DATETIME` allant de '1001-01-01 00:00:00' à '9999-12-31 23:59:59'.*

#### TIME

Le type `TIME` est un peu plus compliqué, puisqu'il permet non seulement de stocker une heure précise, mais aussi un intervalle de temps. On n'est donc pas limité à 24 heures, et il est même possible de stocker un nombre de jours ou un intervalle négatif. Comme dans `DATETIME`, il faut d'abord donner l'heure, puis les minutes, puis les secondes, chaque partie pouvant être séparée des autres par le caractère `:` . Dans le cas où l'on précise également un nombre de jours, alors les jours sont en premier et séparés du reste par une espace. Exemples :

- `'HH:MM:SS'`
- `'HHH:MM:SS'`
- `'MM:SS'`
- `'J HH:MM:SS'`
- `'HHMMSS'`
- `HHMMSS` (nombre)

> *MySQL supporte des `TIME` allant de '-838:59:59' à '838:59:59*

**YEAR**

`YEAR` est un type intéressant pour retenir que l'année, car il ne prend qu'un seul octet en mémoire. Cependant, un octet ne pouvant contenir que 256 valeurs différentes, `YEAR` est fortement limité : on ne peut y stocker que des années entre 1901 et 2155. Cela dit, cela devrait suffire à la majorité d'entre vous pour au moins les cent prochaines années.

On peut entrer une donnée de type `YEAR` sous forme de chaîne de caractères ou d'entiers, avec 2 ou 4 chiffres. Si l'on ne précise que deux chiffres, le siècle est ajouté par MySQL selon les mêmes critères que pour `DATE` et `DATETIME`, **à une exception près** : si l'on entre 00 (un entier donc), il sera interprété comme la valeur par défaut de `YEAR` 0000. Par contre, si l'on entre `'00'` (une chaîne de caractères), elle sera bien interprétée comme l'année 2000. Plus de précisions sur les valeurs par défaut des types temporels dans quelques instants !

**TIMESTAMP**

Par définition, le timestamp d'une date est le nombre de secondes écoulées depuis le 1er janvier 1970, 0 h 0 min 0 s (TUC) et la date en question. Les timestamps étant stockés sur 4 octets, il existe une limite supérieure : le 19 janvier 2038 à 3 h 14 min 7 s. Par conséquent, vérifiez bien que vous êtes dans l'intervalle de validité avant d'utiliser un timestamp.

**La date par défaut**

Lorsque MySQL rencontre une date/heure incorrecte, ou qui n'est pas dans l'intervalle de validité du champ, la valeur par défaut est stockée à la place. Il s'agit de la valeur "zéro" du type. On peut se référer à cette valeur par défaut en utilisant `'0'` (caractère), 0 (nombre) ou la représentation du "zéro" correspondant au type de la colonne (voir tableau ci-dessous).

| Type | Date par défaut ("zéro") |
|---|---|
| DATE | '0000-00-00' |
| DATETIME | '0000-00-00 00:00:00' |
| TIME | '00:00:00' |
| YEAR | 0000 |

| TIMESTAMP | 00000000000000 |
|---|---|

> *Une exception toutefois : si vous insérez un `TIME` qui dépasse l'intervalle de validité, MySQL ne le remplacera pas par le "zéro", mais par la plus proche valeur appartenant à l'intervalle de validité (-838:59:59 ou 838:59:59).*

---

## Créez une base de données

### Conseils

N'utilisez jamais, **d'espaces ou d'accents** dans vos noms de bases, tables ou colonnes. Au lieu d'avoir une colonne "date de naissance", préférez "date_de_naissance" ou "date_naissance". Et au lieu d'avoir une colonne "prénom", utilisez "prenom". Évitez également d'utiliser des `mots réservés` comme nom de colonnes/tables/bases. Par `"mot réservé"`, on entends un mot-clé SQL, donc un mot qui sert à définir quelque chose dans le langage SQL. documentation officielle

### Conventions

Une convention largement répandue veut que les commandes et mots-clés SQL soient écrits complètement en **MAJUSCULES**. Il est plus facile de relire une commande de 5 lignes lorsque l'on peut différencier au premier coup d'œil les commandes SQL des noms de tables et de colonnes.

## Création et suppression d'une base de données

### Création

La commande SQL pour créer une base de données est la suivante :

```
CREATE DATABASE nom_base;
```

Je vous rappelle qu'il faut également définir l'encodage utilisé (l'UTF-8, dans notre cas). Voici donc la commande complète à taper pour créer votre base :

```
CREATE DATABASE bachelor3 CHARACTER SET 'utf8';
```

Lorsque nous créerons nos tables dans la base de données, automatiquement, elles seront encodées également en UTF-8.

### Suppression

Soyez très prudent, car vous effacez tous les fichiers créés par MySQL qui servent à stocker les informations de votre base.

```
DROP DATABASE bachelor3;
```

Si vous essayez cette commande alors que la base de données bachelor3 n'existe pas, MySQL vous affichera une erreur :

```
mysql> DROP DATABASE bachelor3;
ERROR 1008 (HY000) : Can't drop database 'bachelor3'; database doesn't exist
```

Pour éviter ce message d'erreur, si vous n'êtes pas sûr que la base de données existe, vous pouvez utiliser l'option IF EXISTS, de la manière suivante :

```
DROP DATABASE IF EXISTS bachelor3;
```

Si la base de données existe, vous devriez alors avoir un message du type :

```
Query OK, 0 rows affected (0.00 sec)
```

Si elle n'existe pas, vous aurez :

```
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

### Utilisation d'une base de données

Mais pour pouvoir agir sur cette base, vous devez d'abord la sélectionner. Une fois de plus, la commande est très simple :

```
USE bachelor3
```

Notez que vous pouvez spécifier la base de données sur laquelle vous allez travailler lors de la connexion à MySQL. Il suffit d'ajouter le nom de la base à la fin de la commande de connexion :

```
mysql -u sdz -p bachelor3
```

---

## Créez des tables

### Définition des colonnes

#### Type de colonne

Avant de choisir le type des colonnes, il faut choisir les colonnes que l'on va définir. On va donc créer une table *Eleves*. Qu'est-ce qui caractérise un éleve ? Son nom, son sexe, sa date de naissance et une éventuelle colonne commentaires qui peut servir de fourre-tout.

Examinons donc les colonnes afin d'en choisir le type au mieux :

- **Nom** : ce sera un champ de type alphanumérique. On choisira donc un `VARCHAR(30)`.
- **Sexe** : ici, deux choix possibles (homme ou femme). Nous utiliserons une colonne `CHAR(1)`, contenant soit `'H'` (homme), soit `'F'` (femme).
- **Date de naissance** : il s'agit d'une date, donc soit un `DATETIME`, soit une `DATE`.
- **Commentaires** : un type `alphanumérique`, mais on n'a ici aucune idée de la longueur. Ce sera donc un champ `TEXT`.

#### NULL or NOT NULL ?

Il faut maintenant déterminer si l'on autorise les colonnes à ne pas stocker de valeur :

- **Nom** : Cette colonne ne peut pas être `NULL` car un nom est propre à chaques élèves.

- **Sexe** : Cette colonne peut être **`NULL`** si l'élève ne souhaite pas se prononcer sur son sexe.
- **Date de naissance** : Cette colonne ne peut pas être **`NULL`**.
- **Commentaires** : Cette colonne peut très bien ne rien contenir.

**En résumé**

| Caractéristique | Nom de la colonne | Type | NULL |
|---|---|---|---|
| Nom | nom | `VARCHAR(40)` | NON |
| Sexe | sexe | `CHAR(1)` | OUI |
| Date de naissance | date_naissance | `DATETIME` | NON |
| Commentaire | commentaire | `VARCHAR(30)` | OUI |

> *Ne pas oublier de donner une taille aux colonnes qui en nécessitent une, comme les `VARCHAR(x)`, les `CHAR(x)`, les `DECIMAL(n, d)` ...*

## Introduction aux clés primaires

### Identité

On va donner à chaque élève un numéro d'identité. La colonne que l'on ajoutera s'appellera donc id, et il s'agira d'un `INT`, toujours positif donc `UNSIGNED`. Comme il est peu probable que l'on dépasse les 65000 élèves, on utilisera `SMALLINT`.

Ce champ ne pourra bien sûr pas être `NULL`, sinon il perdrait toute son utilité.

### Clé primaire

La clé primaire d'une table est une **contrainte d'unicité**, composée d'une ou plusieurs colonnes. La clé primaire d'une ligne **permet d'identifier de manière unique cette ligne dans la table**. Si l'on parle de la ligne dont la clé primaire vaut x, il ne doit y avoir aucun doute quant à la ligne dont on parle. Lorsqu'une table possède une clé primaire (et il est extrêmement conseillé de définir une clé primaire pour chaque table créée), celle-ci **doit** être définie. Cette définition correspond exactement au numéro d'identité dont nous venons de parler. Nous définirons donc id comme la clé primaire de la table Elèves, en utilisant les mots-clés `PRIMARY KEY(id)`. Lorsque vous insérerez une nouvelle ligne dans la table, MySQL vérifiera que vous insérez bien un id, et que cet id n'existe pas encore dans la table. Si vous ne respectez pas ces deux contraintes, MySQL n'insérera pas la ligne et vous renverra une erreur.

```
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
```

### Auto-incrémentation

Il faut donc, pour chaque élève, décider d'une valeur pour `id`. Donc, si l'on déclare qu'une colonne doit s'auto-incrémenter (grâce au mot-clé **`AUTO_INCREMENT`**), plus besoin de chercher quelle valeur on va y mettre lors de la prochaine insertion.

## Syntaxe de CREATE TABLE

Par souci de clarté, l'explication de la syntaxe de `CREATE TABLE` sera divisé en deux. La première partie vous donne la syntaxe globale de la commande, et la deuxième partie s'attarde sur la description des colonnes créées dans la table.

**Création de la table**

```
CREATE TABLE [IF NOT EXISTS] Nom_table (
    colonne1 description_colonne1,
    colonne2 description_colonne2,
    colonne3 description_colonne3,
    ...,
    [PRIMARY KEY (colonne_clé_primaire)]
)
ENGINE=INNODB;
```

Le `IF NOT EXISTS` est facultatif (d'où l'utilisation de crochets [ ]), et a le même rôle que dans la commande `CREATE DATABASE` : si une table de ce nom existe déjà dans la base de données, la requête renverra un warning plutôt qu'une erreur si `IF NOT EXISTS` est spécifié.

**Définition des colonnes**

Pour définir une colonne, il faut donner son nom en premier, puis sa description. La description est constituée au minimum du type de la colonne. Exemple :

```
nom VARCHAR(30),
sexe CHAR(1)
```

C'est aussi dans la description que l'on précise si la colonne peut contenir `NULL` ou pas (par défaut, `NULL` est autorisé). Exemple :

```
nom VARCHAR(30) NOT NULL,
date_naissance DATETIME NOT NULL
```

L'auto-incrémentation se définit également à cet endroit.

```
id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT [PRIMARY KEY]
```

Enfin, on peut donner une valeur par défaut au champ. Si lorsque l'on insère une ligne, aucune valeur n'est précisée pour le champ, c'est la valeur par défaut qui sera utilisée.

```
nom VARCHAR(30) NOT NULL DEFAULT 'Hamilton'
```

> *Une valeur par défaut DOIT être une constante. Ce ne peut pas être une fonction.*

**Application : création de *Eleves***

```
CREATE TABLE Eleves (
    id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
    nom VARCHAR(30) NOT NULL,
    sexe CHAR(1),
    date_naissance DATETIME NOT NULL,
    commentaires TEXT,
    PRIMARY KEY (id)
)
ENGINE=INNODB;
```

#### Vérifications

Voici deux commandes vous permettant de vérifier que vous avez bien créé une table *Eleves* avec les cinq colonnes que vous vouliez.

```
SHOW TABLES;      -- liste les tables de la base de données

DESCRIBE Eleves;  -- liste les colonnes de la table avec leurs caractéristiques
```

### Suppression d'une table

La commande pour supprimer une table est la même que celle pour supprimer une base de données.

```
DROP TABLE Eleves;
```

---

## Modifiez une table

### Syntaxe de la requête

Lorsque l'on modifie une table, on peut vouloir lui ajouter, retirer ou modifier quelque chose. Dans les trois cas, c'est la commande `ALTER TABLE` qui sera utilisée, une variante existant pour chacune des opérations :

```
ALTER TABLE nom_table ADD ... -- permet d'ajouter quelque chose (une colonne par exemple)

ALTER TABLE nom_table DROP ... -- permet de retirer quelque chose

ALTER TABLE nom_table CHANGE ...
ALTER TABLE nom_table MODIFY ... -- permettent de modifier une colonne
```

#### Avant de continuer

Dans la seconde partie de ce chapitre, nous allons faire quelques modifications sur notre table *Eleves*, mais en attendant, nous allons utiliser la table suivante afin de tester les différentes possibilités d'`ALTER TABLE` :

```
CREATE TABLE Test (
    id INT NOT NULL,
    nom VARCHAR(10) NOT NULL,
        PRIMARY KEY(id)
);
```

### Ajout et suppression d'une colonne

#### Ajout

```
mysql> ALTER TABLE nom_table
    -> ADD [COLUMN] nom_colonne description_colonne;
```

Le `[COLUMN]` est facultatif, donc, si à la suite de `ADD` , vous ne précisez pas ce que vous voulez ajouter, MySQL considérera qu'il s'agit d'une colonne.

`description_colonne` correspond à la même chose que lorsque l'on crée une table.

Ajoutons une colonne *date_insertion* à notre table de test. Il s'agit d'une date, donc une colonne de type `Date` :

```
mysql> ALTER TABLE Test
    -> ADD COLUMN date_insertion DATE NOT NULL;
```

La commande `DESCRIBE Test;` vous permettra de vérifier les changements apportés.

**Suppression**

```
mysql> ALTER TABLE nom_table
    -> DROP [COLUMN] nom_colonne;
```

Comme pour les ajouts, le mot `COLUMN` est facultatif. Par défaut, MySQL considérera que vous parlez d'une colonne.

Si l'on reprend notre précédent exemple :

```
mysql> ALTER TABLE Test
    -> DROP COLUMN date_insertion; -- Suppression de la colonne date_insertion
```

### Modification de colonne

**Changement du nom de la colonne**

```
mysql> ALTER TABLE nom_table
    -> CHANGE ancien_nom nouveau_nom description_colonne;
```

Si l'on reprend notre précédent exemple :

```
mysql> ALTER TABLE Test
    -> CHANGE nom prenom VARCHAR(10) NOT NULL;
```

Attention, la description de la colonne doit être complète, sinon elle sera également modifiée. Si vous ne précisez pas `NOT NULL` dans la commande précédente, *prenom* pourra contenir `NULL`, alors que, du temps où elle s'appelait *nom*, cela lui était interdit.

**Changement du type de données**

Les mots-clés `CHANGE` et `MODIFY` peuvent être utilisés pour changer le type de donnée de la colonne, mais aussi changer la valeur par défaut ou ajouter/supprimer une propriété `AUTO_INCREMENT`. Si vous utilisez `CHANGE`, vous pouvez renommer la colonne en même temps. Si vous ne désirez pas la renommer, il suffit d'indiquer deux fois le même nom.

Voici les syntaxes possibles :

```
ALTER TABLE nom_table
CHANGE ancien_nom nouveau_nom nouvelle_description;

ALTER TABLE nom_table
MODIFY nom_colonne nouvelle_description;
```

Des exemples pour illustrer :

```
ALTER TABLE Test
CHANGE prenom nom VARCHAR(30) NOT NULL; -- Changement du type + changement du nom

ALTER TABLE Test
CHANGE id id BIGINT NOT NULL; -- Changement du type sans renommer

ALTER TABLE Test
MODIFY id BIGINT NOT NULL AUTO_INCREMENT; -- Ajout de l'auto-incrémentation

ALTER TABLE Test
MODIFY nom VARCHAR(30) NOT NULL DEFAULT 'Blabla'; -- Changement de la description
(même type mais ajout d'une valeur par défaut)
```

Il existe d'autres possibilités et combinaisons pour la commande `ALTER TABLE` que vous trouverez dans la documentation officielle.

---

## Insérez des données

### Syntaxe de INSERT

Deux possibilités s'offrent à nous lorsque l'on veut insérer une ligne dans une table : soit donner une valeur pour chaque colonne de la ligne, soit ne donner les valeurs que de certaines colonnes, auquel cas il faut bien sûr préciser de quelles colonnes il s'agit.

#### Insertion sans préciser les colonnes

Notre table *Eleves* est composée de six colonnes : *id*, *nom*, *sexe*, *date_naissance* et *commentaires*.

Voici donc la syntaxe à utiliser pour insérer une ligne dans *Eleves*, sans renseigner les colonnes pour lesquelles on donne une valeur (implicitement, MySQL considère que l'on donne une valeur pour chaque colonne de la table).

```
INSERT INTO Eleves
VALUES (1, 'Hamilton', 'H', '2010-04-05 13:43:00', 'Elève impliqué');
```

**Deuxième exemple** : cette fois-ci, on ne connaît pas le sexe et on n'a aucun commentaire à faire sur l'élève.

```
INSERT INTO Eleves
VALUES (2, 'Verstappen', NULL, '2010-03-24 02:23:00', NULL);
```

**Troisième et dernier exemple** : on donne `NULL` comme valeur d'*id*, ce qui en principe est impossible puisque *id* est défini comme `NOT NULL` et comme clé primaire. Cependant, l'auto-incrémentation fait que MySQL va calculer tout seul quel *id* il faut donner à la ligne (ici : 3).

```
INSERT INTO Eleves
VALUES (NULL , 'Calderón', 'F', '2010-09-13 15:02:00', NULL);
```

Vous avez maintenant trois élèves dans votre table :

| Id | Nom | Sexe | Date de naissance | Commentaires |
|---|---|---|---|---|
| 1 | Hamilton | H | 2010-04-05 13:43:00 | Elève impliqué |
| 2 | Verstappen | NULL | 2010-03-24 02:23:00 | NULL |
| 3 | Calderón | F | 2010-09-13 15:02:00 | NULL |

Pour vérifier, vous pouvez utiliser la requête suivante :

```
SELECT * FROM Eleves;
```

#### Insertion en précisant les colonnes

Dans la requête, nous allons donc écrire explicitement à quelle(s) colonne(s) nous donnons une valeur. Ceci va permettre deux choses :

- On ne doit plus donner les valeurs dans l'ordre de création des colonnes, mais dans l'ordre précisé par la requête.
- On n'est plus obligé de donner une valeur à chaque colonne ; plus besoin de NULL lorsque l'on n'a pas de valeur à mettre.

```
INSERT INTO Eleves (nom, sexe, date_naissance)
    VALUES ('Senna', 'H', '2010-04-05 13:43:00');
INSERT INTO Eleves (nom, commentaires, date_naissance)
    VALUES ('Schumacher', 'Elève turbulent', '2010-03-24 02:23:00');
INSERT INTO Eleves (date_naissance, commentaires, nom, sexe)
    VALUES ('2010-09-13 15:02:00', 'Peut mieux faire', 'Floersch', 'F');
```

#### Insertion multiple

Si vous avez plusieurs lignes à introduire, il est possible de le faire en une seule requête de la manière suivante :

```
INSERT INTO Eleves (nom, sexe, date_naissance, commentaire)
VALUES ('Vettel', NULL, '2008-12-06 05:18:00', 'Elève très respectueux'),
        ('Prost', 'M', '2008-09-11 15:38:00', NULL),
        ('Visser', 'F', '2010-08-23 05:18:00', NULL);
```

> *Vous êtes obligé de préciser les mêmes colonnes pour chaque entrée, quitte à mettre `NULL` pour certaines.*

### Syntaxe alternative de MySQL

MySQL propose une syntaxe alternative à `INSERT INTO ... VALUES ...` pour insérer des données dans une table.

```
INSERT INTO Eleves
SET nom='Alonso', sexe='M', date_naissance='2010-07-21 15:41:00';
```

Cette syntaxe présente deux avantages :

- Le fait d'avoir l'un à côté de l'autre la colonne et la valeur qu'on lui attribue ( `nom = 'Alonso'` ) rend la syntaxe plus lisible et plus facile à manipuler. En effet, ici, il n'y a que cinq colonnes, mais imaginez une table

avec 20, voire 100 colonnes. Difficile d'être sûr que l'ordre dans lequel on a déclaré les colonnes est bien le même que l'ordre des valeurs qu'on leur donne...
- Elle est très semblable à la syntaxe de `UPDATE`, que nous verrons plus tard et qui permet de modifier des données existantes. C'est donc moins de choses à retenir.

> *Cependant, cette syntaxe alternative présente également des défauts qui sont plus importants que les avantages apportés. C'est pourquoi il est déconseillé de l'utiliser.*

En effet, cette syntaxe présente deux défauts majeurs.

- Elle est propre à MySQL. Ce n'est pas du SQL pur. De ce fait, si vous décidez un jour de migrer votre base vers un autre SGBDR, vous devrez réécrire toutes les requêtes `INSERT` utilisant cette syntaxe.
- Elle ne permet pas l'insertion multiple.

### Utilisation de fichiers externes

Pour éviter d'écrire vous-même toutes les requêtes d'insertion, nous allons voir comment on peut utiliser un fichier texte pour interagir avec notre base de données.

#### Exécuter des commandes SQL à partir d'un fichier

Une solution pour éviter d'écrire toutes les commandes à la main dans la console est d'écrire les requêtes dans un fichier texte, puis de dire à MySQL d'exécuter les requêtes contenues dans ce fichier :

```
SOURCE monFichier.sql;

# ou

\. monFichier.sql;
```

Ces deux commandes sont équivalentes et vont exécuter le fichier `monFichier.sql`.

> ***Attention** : si vous ne lui indiquez pas le chemin, MySQL va aller chercher votre fichier dans le dossier où vous étiez lors de votre connexion*

```
SOURCE Users\michel\ynov\monFichier.sql;

SOURCE C:/"Document and Settings"/ynov/monFichier.sql; --Si vous utilisez Windows, utilisez "/" au lieu de "\"
```

#### Insérer des données à partir d'un fichier formaté

Par fichier formaté, on entend un fichier qui suit certaines règles de format. Un exemple typique serait les fichiers .csv. Ces fichiers contiennent un certain nombre de données et sont organisés en tables. Chaque ligne correspond à une entrée, et les colonnes de la table sont séparées par un caractère défini (souvent une virgule ou un point-virgule). Ceci, par exemple, est un format csv :

```
nom;prenom;date_naissance
Charles;Myeur;1994-12-30
```

```
Bruno;Debor;1978-05-12
Mireille;Franelli;1990-08-23
```

Ce type de fichier est facile à produire et à lire avec un logiciel de type tableur (Microsoft Excel, ExcelViewer, Numbers...). La bonne nouvelle est qu'il est aussi possible de lire ce type de fichier avec MySQL, afin de remplir une table avec les données contenues dans le fichier.

La commande SQL permettant cela est `LOAD DATA INFILE` :

```
LOAD DATA [LOCAL] INFILE 'nom_fichier'
INTO TABLE nom_table
[FIELDS
    [TERMINATED BY '\t']
    [ENCLOSED BY '']
    [ESCAPED BY '\\' ]
]
[LINES
    [STARTING BY '']
    [TERMINATED BY '\n']
]
[IGNORE nombre LINES]
[(nom_colonne,...)];
```

Le mot-clé `LOCAL` sert à spécifier si le fichier se trouve côté client ou côté serveur (auquel cas, on ne met pas `LOCAL` dans la commande). Si le fichier se trouve du côté du serveur, il est obligatoire, pour des raisons de sécurité, qu'il soit dans le répertoire de la base de données, c'est-à-dire dans le répertoire créé par MySQL à la création de la base de données, qui contient les fichiers dans lesquels sont stockées les données de la base.

Les clauses `FIELDS` et `LINES` permettent de définir le format de fichier utilisé. `FIELDS` se rapporte aux colonnes, et `LINES` aux lignes. Ces deux clauses sont facultatives.

Si vous précisez une clause `FIELDS`, il faut lui donner au moins une des trois "sous-clauses".

- `TERMINATED BY`, qui définit le caractère séparant les colonnes, entre guillemets bien sûr. `'\t'` correspond à une tabulation. C'est le caractère par défaut.
- `ENCLOSED BY`, qui définit le caractère entourant les valeurs dans chaque colonne (vide par défaut).
- `ESCAPED BY`, qui définit le caractère d'échappement pour les caractères spéciaux. Si par exemple vous définissez vos valeurs comme entourées d'apostrophes, mais que certaines valeurs contiennent des apostrophes, il faut échapper ces apostrophes "internes" afin qu'elles ne soient pas considérées comme un début ou une fin de valeur. Par défaut, il s'agit du `\` habituel. Remarquez qu'il faut lui-même l'échapper dans la clause.

De même pour `LINES`, si vous l'utilisez, il faut lui donner une ou deux sous-clauses.

- `STARTING BY`, qui définit le caractère de début de ligne (vide par défaut).
- `TERMINATED BY`, qui définit le caractère de fin de ligne (`'\n'` par défaut, mais attention, les fichiers générés sous Windows ont souvent `'\r\n'` comme

caractère de fin de ligne).

La clause `IGNORE nombre LINES` permet d'ignorer un certain nombre de lignes. Par exemple, si la première ligne de votre fichier contient les noms des colonnes, vous ne voulez pas l'insérer dans votre table. Il suffit alors d'utiliser `IGNORE 1 LINES`.

Enfin, vous pouvez préciser le nom des colonnes présentes dans votre fichier. Attention, évidemment, à ce que les colonnes absentes acceptent `NULL` ou soient auto-incrémentées.

On reprend l'example précédent, en imaginant que nous ayons une table *Personne* contenant les colonnes *id* (clé primaire auto-incrémentée), *nom*, *prenom*, *date_naissance* et *adresse* (qui peut être `NULL`).

```
nom;prenom;date_naissance
Charles;Myeur;1994-12-30
Bruno;Debor;1978-05-12
Mireille;Franelli;1990-08-23
```

Si ce fichier est enregistré sous le nom personne.csv, il vous suffit d'exécuter la commande suivante pour enregistrer ces trois lignes dans la table *Personne*, en spécifiant si nécessaire le chemin complet vers personne.csv :

```
LOAD DATA LOCAL INFILE 'personne.csv'
INTO TABLE Personne
FIELDS TERMINATED BY ';'
LINES TERMINATED BY '\n' -- ou '\r\n' selon l'ordinateur et le programme utilisés pour créer le fichier
IGNORE 1 LINES
(nom,prenom,date_naissance);
```

## Remplissage de la base

Nous allons utiliser les deux techniques présentés pour remplir un peu notre base.

### Exécution de commandes SQL

Voici pour l'example le contenu d'un fichier `remplissageEleve.sql` :

```
INSERT INTO Eleves (nom, sexe, date_naissance, commentaires) VALUES
('Bottas', 'M', '2008-02-20 15:45:00' , NULL),
('Villeneuve', 'M','2009-05-26 08:54:00'  , NULL),
('Laffite', 'F','2007-04-24 12:54:00' , NULL),
('Ogier', 'M','2009-05-26 08:56:00' , NULL),
('Quartararo', 'M','2008-02-20 15:47:00' , 'Elève brillant !'),
('Hubert', NULL,'2009-05-26 08:50:00' , NULL),
('Grosjean', 'M', '2008-03-10 13:45:00', NULL),
('Stroll', NULL,'2007-04-24 12:59:00' , NULL),
('Lauda', 'M', '2009-05-26 09:02:00', NULL),
('Hunt', 'M','2007-04-24 12:45:00' , NULL),
('Norris', 'M','2008-03-10 13:43:00' , NULL),
('Fangio', 'M','2007-04-24 12:42:00' , NULL),
('Sainz', 'M', '2009-03-05 13:54:00', NULL),
('Ricciardo', 'M','2007-04-12 05:23:00' , NULL),
('Leclerc', 'M', '2006-05-14 15:50:00', NULL),
```

```
('Gasly', 'M','2006-05-14 15:48:00' , NULL),
('Russel', 'M', '2008-03-10 13:40:00',, NULL),
('Perez', 'M', '2006-05-14 15:40:00', NULL),
('Ocon', 'M','2009-05-14 06:30:00' , NULL),
('Raikkonen', 'M','2007-03-12 12:05:00' , NULL),
('Mazepin', 'M','2008-02-20 15:45:00' , 'N''as rien à faire ici'),
('De Vries', 'M','2007-03-12 12:07:00' , NULL),
('Stewart', 'M','2006-05-19 16:17:00' , NULL),
('Pourchaire', 'M','2008-04-20 03:22:00' , NULL),
('Kubica ', 'M','2006-05-19 16:56:00' , NULL);
```

Il faut alors taper :

```
SOURCE remplissageEleves.sql;
```

**LOAD DATA INFILE**

À nouveau, voici pour l'example le contenu d'un fichier `eleve.csv` :

```
"Quartararo";"M";"2009-05-14 06:42:00";NULL
"Bradl";"F";"2006-05-19 16:06:00";NULL
"Petrucci";"F";"2009-05-14 06:45:00";NULL
"Marini";"F";"2008-04-20 03:26:00";NULL
"Viñales";"F";"2007-03-12 12:00:00";NULL
"Morbidelli";"F";"2006-05-19 15:59:00";NULL
"Bastianini";"F";"2008-04-20 03:20:00";"Ne fais pas ses devoirs"
"Lecuona";"F";"2007-03-12 11:54:00";NULL
"Nakagami";"F";"2006-05-19 16:16:00";NULL
"Savadori";"F";"2007-04-01 18:17:00";NULL
"Binder";"F";"2009-03-24 08:23:00";NULL
"Crutchlow";"F";"2009-03-26 01:24:00";NULL
"Mir";"F";"2006-03-15 14:56:00";NULL
"Espargaro";"F";"2008-03-15 12:02:00";NULL
"Rins";"F";"2009-05-25 19:57:00";NULL
"Miller";"F";"2007-04-01 03:54:00";NULL
"Bagnaia";"F";"2006-03-15 14:26:00";"Insomniaque"
"Marquez";"M";"2007-04-02 01:45:00";NULL
"Oliveira";"M";"2008-03-16 08:20:00";NULL
"Martin";"M";"2008-03-15 18:45:00";"Surpoids"
"Marquez";"M";"2009-05-25 18:54:00";NULL
"Bradl";"M";"2007-03-04 19:36:00";NULL
"Navarro";"M";"2008-02-20 02:50:00";NULL
"Rossi";"M";"2009-03-26 08:28:00";NULL
"Zarco";"F";"2009-03-26 07:55:00";NULL
```

> ***Attention, le fichier doit se terminer par un saut de ligne !***

Il faut alors exécuter la commande suivante :

```
LOAD DATA LOCAL INFILE 'eleve.csv'
INTO TABLE Eleves
FIELDS TERMINATED BY ';' ENCLOSED BY '"'
LINES TERMINATED BY '\n' -- ou '\r\n' selon l'ordinateur et le programme utilisés pour
```

```
créer le fichier
(nom, sexe, date_naissance, commentaires);
```

---

## Sélectionnez des données

### Syntaxe de `SELECT`

`SELECT` permet d'afficher des données directement : des chaînes de caractères, des résultats de calculs, etc.

```
SELECT 'Hello World !';
SELECT 3+2;
```

`SELECT` permet également de sélectionner des données à partir d'une table. Pour cela, il faut ajouter une clause à la commande `SELECT` : la clause `FROM` qui définit de quelle structure (dans notre cas, une table) viennent les données.

```
SELECT colonne1, colonne2, ...
FROM nom_table;
```

Par exemple, si l'on veut sélectionner le nom, le sexe et la date de naissance présents dans la table *Eleves*, on utilisera :

```
SELECT nom, sexe, date_naissance
FROM Eleves
```

#### Sélectionner toutes les colonnes

Si vous désirez sélectionner toutes les colonnes, vous pouvez utiliser le caractère `*` dans votre requête :

```
SELECT *
FROM Eleves;
```

Il est cependant déconseillé d'utiliser `SELECT *` trop souvent. Donner explicitement le nom des colonnes dont vous avez besoin présente deux avantages :

- d'une part, vous êtes certain de ce que vous récupérez ;
- d'autre part, vous récupérez uniquement ce dont vous avez vraiment besoin, ce qui permet d'économiser des ressources.

Le désavantage est bien sûr que vous avez plus à écrire.

### La clause WHERE

La clause `WHERE` ("où" en anglais) permet de restreindre les résultats selon des critères de recherche :

```
SELECT *
FROM Eleves
WHERE sexe='F';
```

#### Les opérateurs de comparaison

Les opérateurs de comparaison sont les symboles que l'on utilise pour définir les critères de recherche (le `=` dans notre exemple précédent). Huit opérateurs simples peuvent être utilisés :

| Opérateur | Signification |
|---|---|
| = | égal |
| < | inférieur |
| <= | inférieur ou égal |
| > | supérieur |
| >= | supérieur ou égal |
| <> ou != | différent |
| <=> | égal (valable pour NULL aussi) |

Exemple :

```
SELECT *
FROM Eleves
WHERE date_naissance < '2008-01-01'; -- Eleves nés avant 2008

SELECT *
FROM Eleves
WHERE nom <> 'Mazepin'; -- Tous les Eleves sauf Mazepin (désolé ^^)
```

**Combinaisons de critères**

Imaginons que l'on veuille *Leclerc* et *Sainz* en même temps. Pour cela, il suffit de combiner les critères. Il faut donc des opérateurs logiques :

| Opérateur | Symbole | Signification |
|---|---|---|
| AND | && | ET |
| OR | \|\| | OU |
| XOR | | OU Exclusif |
| NOT | ! | NON |

**AND**

Je veux sélectionner toutes les femmes nées en 2010. Je veux donc sélectionner les élèves qui sont à la fois des femmes ET nées en 2010. J'utilise l'opérateur AND :

```
SELECT *
FROM Eleves
WHERE sexe='F'
    AND YEAR(date_naissance)='2010';
-- OU
SELECT *
FROM Eleves
```

```
WHERE sexe='F'
    && YEAR(date_naissance)='2010';
```

**OR**

Je désire donc obtenir les hommes OU les élèves nés en 2008 :

```
SELECT *
FROM Eleves
WHERE sexe='H'
    OR YEAR(date_naissance)='2008';
-- OU
SELECT *
FROM Eleves
WHERE sexe='H'
    || YEAR(date_naissance)='2008';
```

> *Il est conseillé d'utiliser plutôt `OR` que `||`, car, dans la majorité des SGBDR (et dans la norme SQL), l'opérateur `||` sert à la concaténation, c'est-à-dire à rassembler plusieurs chaînes de caractères en une seule. Il vaut donc mieux prendre l'habitude d'utiliser `OR`, au cas où vous changeriez un jour de SGBDR (ou tout simplement parce que c'est une bonne habitude).*

**NOT**

Sélection de tous les élèves femme sauf né en 2009.

```
SELECT *
FROM Eleves
WHERE sexe='F'
    AND NOT YEAR(date_naissance)='2009';
-- OU
SELECT *
FROM Eleves
WHERE sexe='F'
    AND ! YEAR(date_naissance)='2009';
```

Les parenthèses sont indispensables dans le cas du `!` , sinon il porte uniquement sur *date_naissance*. Oublier les parenthèses revient à faire `WHERE (NOT date_naissance) = 'YEAR(date_naissance)='2009''` , ce qui n'a pas de sens.

**XOR**

Sélection des ékèves qui sont soit des hommes, soit né en 2005 (mais pas les deux) :

```
SELECT *
FROM Eleves
WHERE sexe='H'
    XOR YEAR(date_naissance)='2005';
```

**Sélection complexe**

Lorsque vous utilisez plusieurs critères et que vous devez donc combiner plusieurs opérateurs logiques, il est extrêmement important de bien structurer la requête. En particulier, il faut placer des parenthèses au bon endroit.

Petit exemple simple :
Critères : rouge `AND` vert `OR` bleu

Qu'accepte-t-on ?

- Ce qui est rouge et vert, et ce qui est bleu ?
- Ou ce qui est rouge, et soit vert, soit bleu ?

Dans le premier cas, [rouge, vert] et [bleu] seraient acceptés. Dans le deuxième, c'est [rouge, vert] et [rouge, bleu] qui seront acceptés, et non [bleu].

En fait, le premier cas correspond à (rouge `AND` vert) `OR` bleu, et le deuxième cas à rouge `AND` (vert `OR` bleu).

Avec des parenthèses, pas moyen de se tromper sur ce que l'on désire sélectionner.

**EXEMPLE**

Je voudrais les élèves qui sont soit nés après 2009, soit des hommes ou des femmes, mais dans le cas des femmes, elles doivent être nées avant juin 2007.

L'astuce, c'est de penser en niveaux. Nous allons découper la requête.

On cherche :

- les élèves nés après 2009 ;
- les élèves hommes et femmes (uniquement nées avant juin 2007 pour les femmes).

L'opérateur logique sera `OR` puisqu'il faut que les élèves répondent à un seul des deux critères pour être sélectionnés.

On continue à découper. Le premier critère ne peut plus être subdivisé, contrairement au deuxième. On cherche :

- les élèves nés après 2009 ;
- les élèves :
  - hommes ;
  - et femmes nées avant juin 2007.

Nous avons bien défini les différents niveaux, il n'y a plus qu'à écrire la requête avec les bons opérateurs logiques :

```
SELECT *
FROM Eleves
WHERE date_naissance > '2009-12-31'
     AND
    ( sexe='H'
        OR
        ( sexe='F' AND date_naissance < '2007-06-01' )
    );
```

**Le cas de NULL**

Le marqueur `NULL` est un peu particulier. En effet, vous ne pouvez pas tester directement colonne = `NULL`. Essayons donc :

```
SELECT *
FROM Eleves
WHERE nom = NULL; -- sélection des élèves sans nom
```

```
SELECT *
FROM Eleves
WHERE commentaires <> NULL; -- sélection des élèves pour lesquels un commentaire existe
```

Ces deux requêtes ne renvoient pas les résultats que l'on pourrait espérer. En fait, elles ne renvoient aucun résultat. C'est donc ici qu'intervient notre opérateur de comparaison un peu spécial `<=>` qui permet de reconnaître `NULL`. Une autre possibilité est d'utiliser les mots-clés `IS NULL`, et si l'on veut exclure les `NULL` : `IS NOT NULL`. Nous pouvons donc réécrire nos requêtes, correctement cette fois-ci :

```
SELECT *
FROM Elevs
WHERE sexe <=> NULL; -- sélection des élèves dont le sexe n'est pas prononcé

-- OU

SELECT *
FROM Elevs
WHERE sexe IS NULL;

SELECT *
FROM Elevs
WHERE commentaires IS NOT NULL; -- sélection des élèves pour lesquels un commentaire existe
```

Cette fois-ci, cela fonctionne parfaitement :

| id | nom | sexe | date_naissance | commentaires |
|---|---|---|---|---|
| 4 | Hamilton | H | 2009-08-03 05:12:00 | NULL |
| 9 | Verstappen | NULL | 2010-08-23 05:18:00 | NULL |

## Tri des données

Pour trier vos données, il suffit d'ajouter `ORDER BY tri` à votre requête (après les critères de sélection de `WHERE` s'il y en a) et de remplacer "tri" par la colonne sur laquelle vous voulez trier vos données bien sûr.

Par exemple, pour trier la date de naissance :

```
SELECT *
FROM Elèves
WHERE sexe='H'
ORDER BY date_naissance;
```

Vos données sont triées, les plus vieux hommes sont récupérés en premier, les plus jeunes à la fin.

### Tri ascendant ou descendant

Pour déterminer le sens du tri effectué, SQL possède deux mots-clés : `ASC` pour ascendant, et `DESC` pour descendant. Par défaut, si vous ne précisez rien, c'est un

tri ascendant qui est effectué.
Si, par contre, vous utilisez le mot `DESC`, l'ordre est inversé : plus grand nombre d'abord, date la plus récente d'abord, et ordre anti-alphabétique pour les caractères.

```
SELECT *
FROM Eleves
WHERE sexe='F'
    AND nom IS NOT NULL
ORDER BY nom DESC;
```

#### Trier sur plusieurs colonnes

Il est également possible de trier sur plusieurs colonnes. Par exemple, si vous voulez que les résultats soient triés par nom et, dans chaque nom, triés par date de naissance, il suffit de donner les deux colonnes correspondantes à `ORDER BY` :

```
SELECT *
FROM Eleves
ORDER BY nom, date_naissance;
```

> *L'ordre dans lequel vous donnez les colonnes est important, le tri se fera d'abord sur la première colonne donnée, puis sur la seconde, etc.*

Vous pouvez trier sur autant de colonnes que vous voulez.

### Éliminer les doublons

Il peut arriver que MySQL vous donne plusieurs fois le même résultat parce que certaines informations sont présentes plusieurs fois dans la table.

Petit exemple : vous voulez savoir quelles sont les noms que vous possédez dans votre établissement :

```
SELECT sexe
FROM Eleves;
```

En effet, vous allez bien récupérer toutes les noms que vous possédez, mais si vous avez 500 femmes, vous allez récupérer 500 lignes 'femmes'.

Pour cela il y a une solution : le mot-clé `DISTINCT`. Ce mot-clé se place juste après `SELECT` et permet d'éliminer les doublons.

```
SELECT DISTINCT sexe
FROM Eleves;
```

Ceci devrait vous ramener deux lignes avec les deux sexes qui se trouvent dans la table.

> *Pour éliminer un doublon, il faut que toute la ligne **sélectionnée** soit égale à une autre ligne du jeu de résultats. Ne seront donc prises en compte que les colonnes que vous avez précisées dans votre `SELECT`. Uniquement sexe donc, dans notre exemple.*

### Restreindre les résultats

En plus de restreindre une recherche en lui donnant des critères grâce à la clause `WHERE`, il est possible de restreindre le nombre de lignes récupérées. Cela se fait grâce à la clause `LIMIT`.

**Syntaxe**

`LIMIT` s'utilise avec deux paramètres :

- le nombre de lignes que l'on veut récupérer ;
- le décalage, introduit par le mot-clé `OFFSET`, qui indique à partir de quelle ligne on récupère les résultats. Ce paramètre est facultatif. S'il n'est pas précisé, il est mis à 0.

```
LIMIT nombre_de_lignes [OFFSET decalage];
```

Exemple :

```
SELECT *
FROM Eleves
ORDER BY id
LIMIT 6 OFFSET 0;

SELECT *
FROM Eleves
ORDER BY id
LIMIT 6 OFFSET 3;
```

Avec la première requête, vous devriez obtenir six lignes, les six plus petits id, puisque nous n'avons demandé aucun décalage (`OFFSET 0`).

| id | nom | sexe | date_naissance | commentaires |
|---|---|---|---|---|
| 1 | Hamilton | H | 2009-08-03 05:12:00 | NULL |
| 2 | Verstappen | NULL | 2010-10-03 16:44:00 | Elève turbulent |
| 3 | Calderón | F | 2009-06-13 08:17:00 | NULL |
| 4 | Russel | H | 2008-12-06 05:18:00 | Passage en année supérieur |
| 5 | Norris | NULL | 2008-09-11 15:38:00 | NULL |
| 6 | Gasly | H | 2010-08-23 05:18:00 | NULL |

Par contre, dans la deuxième, vous récupérez toujours six lignes, mais vous devriez commencer au quatrième plus petit id, puisque l'on a demandé un décalage de trois lignes.

| id | nom | sexe | date_naissance | commentaires |
|---|---|---|---|---|
| 4 | Ricciardo | H | 2009-08-03 05:12:00 | NULL |
| 5 | Latifi | NULL | 2010-10-03 16:44:00 | NULL |
| 6 | Floersch | F | 2009-06-13 08:17:00 | NULL |
| 7 | Schumacher | H | 2008-12-06 05:18:00 | NULL |

| | | | | |
|---|---|---|---|---|
| 8 | Tsunoda | NULL | 2008-09-11 15:38:00 | Elève dissipé |
| 9 | Bottas | H | 2010-08-23 05:18:00 | NULL |

Exemple :

```
SELECT *
FROM Eleves
ORDER BY id
LIMIT 10;
```

Cette requête est donc équivalente à :

```
SELECT *
FROM Eleves
ORDER BY id
LIMIT 10 OFFSET 0;
```

---

# Élargissez les possibilités de la clause WHERE

## Recherche approximative

Pour ce genre de recherches, l'opérateur **`LIKE`** est très utile, car il permet de faire des recherches en utilisant des "jokers", c'est-à-dire des caractères qui représentent n'importe quel caractère.

Deux jokers existent pour **`LIKE`** :

- `'%'` : qui représente n'importe quelle chaîne de caractères, quelle que soit sa longueur (y compris une chaîne de longueur 0) ;
- `'_'` : qui représente un seul caractère.

### Quelques exemples :

- `'b%'` cherchera toutes les chaînes de caractères commençant par 'b' ("brocoli", "bouli", "b").
- `'b_'` cherchera toutes les chaînes de caractères contenant deux lettres dont la première est'b' ("ba", "bf", "b8").
- `'%ch%ne'` cherchera toutes les chaînes de caractères contenant 'ch' et finissant par 'ne' ("chne", "chine", "échine", "le pays le plus peuplé du monde est la Chine").
- `'p_rl_'` cherchera toutes les chaînes de caractères commençant par un "p" suivi d'un caractère, puis de "rl" et enfin se terminant par un caractère ("parle", "perla", "perle").

## Rechercher '%' ou '_'

Comment faire si vous cherchez une chaîne de caractères contenant `%` ou `_` ? Évidemment, si vous écrivez `LIKE '%'` ou `LIKE '_'`, MySQL vous donnera absolument toutes les chaînes de caractères dans le premier cas, et toutes les chaînes de 1 caractère dans le deuxième. Il faut donc signaler à MySQL que vous ne désirez pas

utiliser `%` ou `_` en tant que joker, mais bien en tant que caractère de recherche. Pour ça, il suffit de mettre le caractère d'échappement `\`, dont je vous ai déjà parlé, devant le `%` ou le `_`.

**Exemple :**

```
SELECT *
FROM Eleves
WHERE commentaires LIKE '%\%%';
```

## Exclure une chaîne de caractères

C'est logique, mais je précise quand même (et puis, cela fait un petit rappel) : l'opérateur logique `NOT` est utilisable avec `LIKE`. Si l'on veut rechercher les animaux dont le nom ne contient pas la lettre a, on peut donc écrire :

```
SELECT *
FROM Eleves
WHERE nom NOT LIKE '%a%';
```

## Recherche dans les numériques

Vous pouvez bien entendu utiliser des chiffres dans une chaîne de caractères. Après tout, ce sont des caractères comme les autres. Par contre, utiliser `LIKE` sur un type numérique ( `INT` , par exemple), c'est déjà plus étonnant. Et pourtant, MySQL le permet. **Attention**, cependant, il s'agit bien d'une particularité MySQL, qui prend souvent un malin plaisir à étendre la norme SQL pure.

`LIKE '1%'` sur une colonne de type numérique trouvera donc des nombres comme 10, 1000, 153.

```
SELECT *
FROM Eleves
WHERE id LIKE '1%';
```

## Recherche dans un intervalle

Il est possible de faire une recherche sur un intervalle à l'aide uniquement des opérateurs de comparaison >= et <=. Par exemple, on peut rechercher les animaux qui sont nés entre le 5 janvier 2008 et le 23 mars 2009 de la manière suivante :

```
SELECT *
FROM Eleves
WHERE date_naissance <= '2009-03-23'
    AND date_naissance >= '2008-01-05';
```

Cela fonctionne très bien. Cependant, SQL dispose d'un opérateur spécifique, pour les intervalles, qui pourrait vous éviter les erreurs d'inattention classiques (< au lieu de >, par exemple) en plus de rendre votre requête plus lisible et plus performante : `BETWEEN minimum AND maximum` (between signifie "entre" en anglais). La requête précédente peut donc s'écrire :

```sql
SELECT *
FROM Eleves
WHERE date_naissance BETWEEN '2008-01-05' AND '2009-03-23';
```

`BETWEEN` peut s'utiliser avec des dates, mais aussi avec des nombres ( `BETWEEN 0 AND 100` ) ou avec des chaînes de caractères ( `BETWEEN 'a' AND 'd'` ), auquel cas c'est l'ordre alphabétique qui sera utilisé (toujours insensible à la casse sauf si l'on utilise des chaînes binaires : `BETWEEN BINARY 'a' AND BINARY 'd'` ). Bien évidemment, on peut aussi exclure un intervalle avec `NOT BETWEEN` .

## Set de critères

Le dernier opérateur à utiliser dans la clause `WHERE` que nous verrons est `IN` .

Imaginons que vous vouliez récupérer les informations des élèves répondant aux noms d'Hamilton, Verstappen, Vettel, Norris, Russel, Schumacher et Senna. Jusqu'à maintenant, vous auriez sans doute fait quelque chose comme ça :

```sql
SELECT *
FROM Eleves
WHERE nom = 'Hamilton'
    OR nom = 'Verstappen'
    OR nom = 'Vettel'
    OR nom = 'Norris'
    OR nom = 'Russel'
    OR nom = 'Schumacher'
    OR nom = 'Senna';
```

`IN` vous permet de faire des recherches dans une liste de valeurs. Parfait pour nous, donc, qui voulons rechercher les élèves correspondant à une liste de noms. Voici la manière d'utiliser `IN` :

```sql
SELECT *
FROM Eleves
WHERE nom IN ('Hamilton', 'Verstappen', 'Vettel', 'Norris', 'Russel', 'Schumacher',
'Senna');
```

---

## Supprimez et modifiez des données

### Sauvegarde d'une base de données

MySQL dispose donc d'un outil spécialement dédié à la sauvegarde des données sous forme de fichiers texte : `mysqldump` .

Cette fonction de sauvegarde s'utilise à partir de la console. Vous devez donc être déconnecté de MySQL pour la lancer.

La manière classique de faire la sauvegarde d'une base de données est de taper la commande suivante :

```
mysqldump -u user -p --opt nom_de_la_base > sauvegarde.sql
```

- `mysqldump` : il s'agit du client permettant de sauvegarder les bases. Rien de spécial à signaler.
- `--opt` : c'est une option de `mysqldump` qui lance la commande avec une série de paramètres qui font que la commande s'effectue très rapidement.
- `nom_de_la_base` : vous l'avez sans doute deviné, c'est ici qu'il faut indiquer le nom de la base que l'on veut sauvegarder.
- `> sauvegarde.sql` : le signe `>` indique que l'on va donner la destination de ce qui va être généré par la commande `sauvegarde.sql`. Il s'agit du nom du fichier qui contiendra la sauvegarde de notre base. Vous pouvez bien sûr l'appeler comme bon vous semble.

Si vous effacez votre base par mégarde, il vous faut d'abord recréer la base de données (avec `CREATE DATABASE nom_base`), puis exécuter la commande suivante (dans la console) :

```
mysql nom_base < chemin_fichier_de_sauvegarde.sql
```

> *`mysqldump` possède de nombreuses options qui sont détaillés dans la documentation de MySQL .*

### Suppression

La commande utilisée pour supprimer des données est DELETE. **Cette opération est irréversible, soyez très prudent !** On utilise la clause `WHERE` de la même manière qu'avec la commande `SELECT` pour préciser quelles lignes doivent être supprimées.

```
DELETE FROM nom_table
WHERE critères;
```

Par exemple : Kimi Raikkonen est parti de l'école. Nous allons donc le retirer de la base de données.

```
DELETE FROM Eleves
WHERE nom = 'Raikkonen';
```

Si vous désirez supprimer toutes les lignes d'une table, il suffit de ne pas préciser de clause `WHERE`.

```
DELETE FROM Eleves;
```

### Modification

La modification des données se fait grâce à la commande `UPDATE`, dont la syntaxe est la suivante :

```
UPDATE nom_table
SET col1 = val1 [, col2 = val2, ...]
[WHERE ...];
```

Par exemple, dans le cas où il y ai une faute de frappe et que Verstappen soit Verstappen dans la bdd. Il faut donc modifier son nom :

```
UPDATE Eleve
SET nom='Verstappen'
```

```
WHERE id=21;
```

Vérifiez d'abord chez vous que l'élève portant le numéro d'identification 21 est bien Verstappen. La clé primaire (*id*) est utilisée pour identifier la ligne à modifier, car c'est la seule manière d'être sûr que l'on ne modifiera que la ligne que l'on désire.

> *Tout comme pour la commande `DELETE`, si vous omettez la clause `WHERE` dans un `UPDATE`, la modification se fera sur toutes les lignes de la table. Soyez prudent !*

La requête suivante changerait donc le commentaire pour tous les élèves stockés dans la table Eleves :

```
UPDATE Eleves
SET commentaires='modification de toutes les lignes';
```

---

## Index

Un index est une structure qui reprend la liste ordonnée des valeurs auxquelles il se rapporte. Les index sont utilisés pour accélérer les requêtes (notamment les requêtes impliquant plusieurs tables, ou les requêtes de recherche), et sont indispensables à la création de clés, étrangères et primaires, qui permettent de garantir l'intégrité des données de la base.

### Qu'est-ce qu'un index ?

> ***Structure de données** qui reprend la **liste ordonnée** des valeurs auxquelles il se rapporte.*

Lorsque vous créez un index sur une table, MySQL stocke cet index sous forme d'une structure particulière, contenant les valeurs des colonnes impliquées dans l'index. Cette structure stocke les valeurs **triées** et permet d'accéder à chacune de manière efficace et rapide.

Les données d'*Eleves* ne sont pas stockées suivant un ordre intelligible pour nous. Par contre, l'index sur l'*id* est trié simplement par ordre croissant. Cela permet de grandement accélérer toute recherche faite sur cet *id*.
Imaginons en effet que nous voulions récupérer toutes les lignes dont l'*id* est inférieur ou égal à 5. Sans index, MySQL doit parcourir toutes les lignes une à une. Par contre, grâce à l'index, dès qu'il tombe sur la ligne dont l'*id* est 6, il sait qu'il peut s'arrêter, puisque toutes les lignes suivantes auront un *id* supérieur ou égal à 6. Dans cet exemple, on ne gagne que quelques lignes, mais imaginez une table contenant des millions de lignes. Le gain de temps peut être assez considérable.
Par ailleurs, avec les *id* triés par ordre croissant, pour rechercher un *id* particulier, MySQL n'est pas obligé de simplement parcourir les données ligne par ligne. Il peut utiliser des algorithmes de recherche puissants (comme la recherche dichotomique), toujours afin d'accélérer la recherche.

Mais pourquoi ne pas simplement trier la table complète sur la base de la colonne *id* ? Pourquoi créer et stocker une structure spécialement pour l'index ? Tout simplement parce qu'il peut y avoir plusieurs index sur une même table, et que l'ordre des lignes, pour chacun de ces index, n'est pas nécessairement le même. Par exemple, nous pouvons créer un second index pour notre table *Eleves*, sur la colonne *date_naissance*.

**Intérêt des index**

Vous devriez avoir compris maintenant que tout l'intérêt des index est d'accélérer les requêtes qui utilisent des colonnes indexées comme critères de recherche. Par conséquent, si vous savez que, dans votre application, vous ferez énormément de recherches sur la colonne *X*, ajoutez donc un index sur cette colonne, vous ne vous en porterez que mieux.
Les index permettent aussi d'assurer l'intégrité des données de la base. Pour cela, il existe plusieurs types d'index différents, et deux types de "clés". Lorsque je parle de garantir l'intégrité de vos données, cela signifie garantir la qualité de vos données, vous assurer que vos données ont du sens. Grâce aux clés et aux index, vous pouvez par exemple avoir la garantie que tous les clients auxquels vous faites référence dans la table *Commande* existent bien dans la table *Client*.

**Désavantages**

Si tout ce que fait un index, c'est accélérer les requêtes utilisant les critères de recherche correspondants, autant en mettre partout et en profiter à chaque requête ! Sauf qu'évidemment, ce n'est pas si simple. Les index ont deux inconvénients.

- Ils prennent de la place en mémoire
- Ils ralentissent les requêtes d'insertion, modification et suppression, puisqu'à chaque fois, il faut remettre l'index à jour en plus de la table.

Par conséquent, n'ajoutez pas d'index lorsque ce n'est pas vraiment utile.

**Index sur plusieurs colonnes**

Reprenons l'exemple d'une table appelée Client qui reprend les informations des clients d'une société. Elle se présente comme suit :

| id | nom | prenom | init_2e_prenom | email |
|---|---|---|---|---|
| 1 | Dupont | Charles | T | charles.dupont@email.com |
| 2 | François | Damien | V | fdamien@email.com |
| 3 | Dupont | Guillaume | A | guillaumevdb@email.com |
| 4 | Dupont | Valérie | G | dupont.valerie@email.com |
| 5 | François | Martin | D | mdmartin@email.com |

Vous avez bien sûr un index sur la colonne *id*, mais vous constatez que vous faites énormément de recherches par nom, prénom et initiale du second prénom. Vous pourriez donc faire trois index, un pour chacune de ces colonnes. Mais, si vous faites souvent des recherches sur les trois colonnes à la fois, il vaut encore mieux faire un seul index, sur les trois colonnes (*nom, prenom, init_2e_prenom*). L'index contiendra donc les valeurs des trois colonnes et sera trié par nom, ensuite par prénom, et enfin par initiale (l'ordre des colonnes a donc de l'importance !).

Donc lorsque vous chercherez "Dupont Valérie C.", grâce à l'index, MySQL trouvera rapidement tous les "Dupont", parmi lesquels il trouvera toutes les "Valérie" (toujours en se servant du même index), parmi lesquelles il trouvera celle (ou celles) dont le second prénom commence par "C".

**Tirer parti des "index par la gauche"**

Tout cela est bien beau si l'on fait souvent des recherches à la fois sur le nom, le prénom et l'initiale. Mais comment fait-on si l'on fait aussi souvent des recherches uniquement sur le nom, ou uniquement sur le prénom, ou sur le nom et le prénom en même temps, mais sans l'initiale ? Faut-il créer un nouvel index pour chaque type de recherche ? Eh bien non ! MySQl est beaucoup trop fort : il est capable de tirer parti de votre index sur (*nom, prenom, init_2e_prenom*) pour certaines autres recherches.

#### Index sur des colonnes de type alphanumérique

#### Types CHAR et VARCHAR

Lorsque l'on indexe une colonne de type `VARCHAR` ou `CHAR` (comme la colonne *nom* de la table *Client* par exemple), on peut décomposer l'index de la manière suivante :

https://openclassrooms.com/fr/courses/1959476-administrez-vos-bases-de-donnees-avec-mysql/1962880-index#/id/r-1979790

Ici, nous n'avons que des noms assez courts. La colonne nom peut par exemple être de type `VARCHAR(30)`. Mais imaginez une colonne de type `VARCHAR(150)`, qui contient des titres de livres par exemple. Si l'on met un index dessus, MySQL va indexer jusqu'à 150 caractères. Or, il est fort probable que les 25-30 premiers caractères du titre suffisent à trier ceux-ci. Au pire, un ou deux ne seront pas exactement à la bonne place, mais les requêtes en seraient déjà grandement accélérées. Il serait donc plutôt pratique de dire à MySQL : "Indexe cette colonne, mais base-toi seulement sur les *x* premiers caractères". Et c'est possible évidemment, et c'est même très simple. Lorsque l'on créera l'index sur la colonne *titre_livre*, il suffira d'indiquer un nombre entre parenthèses : *titre_livre(25)* par exemple. Ce nombre étant bien sûr le nombre de caractères (dans le sens de la lecture, donc à partir de la gauche) à prendre en compte pour l'index.

Le fait d'utiliser ces index partiels sur des champs alphanumériques permet de **gagner de la place** (un index sur 150 lettres prend évidemment plus de place qu'un index sur 20 lettres), et si la longueur est intelligemment définie, l'accélération permise par l'index sera la même que si l'on avait pris la colonne entière.

#### Types BLOB et TEXT (et dérivés)

Si vous mettez un index sur une colonne de type `BLOB` ou `TEXT` (ou un de leurs dérivés), MySQL **exige** que vous précisiez un nombre de caractères à prendre en compte. Et heureusement... vu la longueur potentielle de ce que l'on stocke dans de telles colonnes.

### Les différents types d'index

En plus des index "simples", que je viens de vous décrire, il existe trois types d'index qui ont des propriétés particulières. Les index `UNIQUE`, les index `FULLTEXT`, et enfin les index `SPATIAL`.

#### Index UNIQUE

Avoir un index `UNIQUE` sur une colonne (ou plusieurs) permet de s'assurer que jamais vous n'insérerez deux fois la même valeur (ou combinaison de valeurs) dans la table.

Par exemple, vous créez un site internet, et vous voulez le doter d'un espace membre. Chaque membre devra se connecter grâce à un pseudo et un mot de passe. Vous avez donc une table *Membre*, qui contient 4 colonnes : *id, pseudo, mot_de_passe* et *date_inscription*.
Deux membres peuvent avoir le même mot de passe, pas de problème. Par contre, que se

passerait-il si deux membres avaient le même pseudo ? Lors de la connexion, il serait impossible de savoir quel mot de passe utiliser et sur quel compte connecter le membre. Il faut donc absolument éviter que deux membres utilisent le même pseudo. Pour cela, on va utiliser un index `UNIQUE` sur la colonne pseudo de la table *Membre*.

#### Contraintes

Lorsque vous mettez un index `UNIQUE` sur une table, vous ne mettez pas seulement un index, vous ajoutez surtout une **contrainte**. Les contraintes sont une notion importante en SQL. Sans le savoir, ou sans savoir que c'était appelé comme cela, vous en avez déjà utilisé. En effet, lorsque vous empêchez une colonne d'accepter `NULL`, vous lui mettez une **contrainte** `NOT NULL`. De même, les valeurs par défaut que vous pouvez donner aux colonnes sont des contraintes. Vous contraignez la colonne à prendre une certaine valeur si aucune autre n'est précisée.

#### Index FULLTEXT

Un index `FULLTEXT` est utilisé pour faire des recherches de manière puissante et rapide sur un texte. On n'utilise donc ce type d'index que sur les colonnes de type `CHAR`, `VARCHAR` ou `TEXT`.

Une différence importante entre les index `FULLTEXT` et les index classiques (et `UNIQUE`) est que l'on ne peut plus utiliser les fameux "index par la gauche" dont je vous ai parlé précédemment. Donc, si vous voulez faire des recherches "fulltext" sur deux colonnes (parfois l'une, parfois l'autre, parfois les deux ensemble), il vous faudra créer trois index FULLTEXT : (*colonne1*), (*colonne2*) et (*colonne1*, *colonne2*).

### Création et suppression des index

#### Ajout des index lors de la création de la table

Ici, deux possibilités : vous pouvez préciser dans la description de la colonne qu'il s'agit d'un index, ou lister les index par la suite.

#### Index dans la description de la colonne

Je rappelle que la description de la colonne se rapporte à l'endroit où vous indiquez le type de données, si la colonne peut contenir `NULL`, etc. Il est donc possible, à ce même endroit, de préciser si la colonne est un index.

```
CREATE TABLE nom_table (
    colonne1 INT KEY,                  -- Crée un index simple sur colonne1
    colonne2 VARCHAR(40) UNIQUE,       -- Crée un index unique sur colonne2
);
```

Quelques petites remarques ici :

- Avec cette syntaxe, **seul le mot `KEY` peut être utilisé** pour définir un index simple. Ailleurs, vous pourrez utiliser `KEY` ou `INDEX`.
- Pour définir un index `UNIQUE` de cette manière, on n'utilise que le mot-clé `UNIQUE`, sans le faire précéder de `INDEX` ou de `KEY` (comme ce sera le cas avec d'autres syntaxes).
- Il n'est **pas possible de définir des index composites** (sur plusieurs colonnes) de cette manière.
- Il n'est pas non plus possible de créer un index sur une partie de la colonne (les *x* premiers caractères).

#### Liste d'index

L'autre possibilité est d'ajouter les index à la suite des colonnes, en séparant chaque élément par une virgule :

```
CREATE TABLE nom_table (
    colonne1 description_colonne1,
    [colonne2 description_colonne2,
    colonne3 description_colonne3,
    ...,]
    [PRIMARY KEY (colonne_clé_primaire)],
    [INDEX [nom_index] (colonne1_index [, colonne2_index, ...])]
)
[ENGINE=moteur];
```

**Exemple** : si l'on avait voulu créer la table *Eleves* avec un index sur la date de naissance, et un autre sur les 10 premières lettres du nom, on aurait pu utiliser la commande suivante :

```
CREATE TABLE Eleves (
    id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
    nom VARCHAR(40) NOT NULL,
    sexe CHAR(1),
    date_naissance DATETIME NOT NULL,
    commentaires TEXT,
    PRIMARY KEY (id),
    INDEX ind_date_naissance (date_naissance),  -- index sur la date de naissance
    INDEX ind_nom (nom(10))                     -- index sur le nom (le chiffre entre
parenthèses étant le nombre de caractères pris en compte)
)
ENGINE=INNODB;
```

Vous n'êtes pas obligé de préciser un nom pour votre index. Si vous ne le faites pas, MySQL en créera un automatiquement pour vous. Je préfère nommer mes index moi-même plutôt que de laisser MySQL créer un nom par défaut, et respecter certaines conventions personnelles. Ainsi, mes index auront toujours le préfixe "ind" suivi du ou des noms des colonnes concernées, le tout séparé par des "_". Il vous appartient de suivre vos propres conventions, bien sûr, l'important étant de vous y retrouver.

Et pour ajouter des index `UNIQUE` ou `FULLTEXT`, c'est le même principe :

```
CREATE TABLE nom_table (
    colonne1 INT NOT NULL,
    colonne2 VARCHAR(40),
    colonne3 TEXT,
    UNIQUE [INDEX] ind_uni_col2 (colonne2),     -- Crée un index UNIQUE sur la
colonne2, INDEX est facultatif
    FULLTEXT [INDEX] ind_full_col3 (colonne3)   -- Crée un index FULLTEXT sur la
colonne3, INDEX est facultatif
)
ENGINE=MyISAM;
```

**Exemple** : création de la table *Eleves* comme précédemment, en ajoutant un index `UNIQUE` sur (*nom*).

```
CREATE TABLE Eleves (
    id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
    nom VARCHAR(40) NOT NULL,
    sexe CHAR(1),
    date_naissance DATETIME NOT NULL,
    commentaires TEXT,
    PRIMARY KEY (id),
    INDEX ind_date_naissance (date_naissance),
    INDEX ind_nom (nom(10)),
    UNIQUE INDEX ind_uni_nom_espece (nom)  -- Index sur le nom
)
ENGINE=INNODB;
```

**Ajout des index après création de la table**

En dehors du fait que parfois vous ne penserez pas à tout au moment de la création de votre table, il peut parfois être intéressant de créer les index après la table. En effet, je vous ai dit que l'ajout d'index sur une table ralentissait l'exécution des requêtes d'écriture (insertion, suppression, modification de données). Par conséquent, si vous créez une table que vous comptez remplir avec un grand nombre de données immédiatement, grâce à la commande `LOAD DATA INFILE`, par exemple, il vaut bien mieux créer la table, la remplir, et ensuite seulement créer les index voulus sur cette table.

Il existe deux commandes permettant de créer des index sur une table existante : `ALTER TABLE`, que vous connaissez déjà un peu, et `CREATE INDEX`. Ces deux commandes sont équivalentes, utilisez celle qui vous parle le plus.

**Ajout avec ALTER TABLE**

```
ALTER TABLE nom_table
ADD INDEX [nom_index] (colonne_index [, colonne2_index ...]); --Ajout d'un index simple

ALTER TABLE nom_table
ADD UNIQUE [nom_index] (colonne_index [, colonne2_index ...]); --Ajout d'un index UNIQUE

ALTER TABLE nom_table
ADD FULLTEXT [nom_index] (colonne_index [, colonne2_index ...]); --Ajout d'un index FULLTEXT
```

Contrairement à ce qui se passait pour l'ajout d'une colonne, il est ici obligatoire de préciser `INDEX` (ou `UNIQUE`, ou `FULLTEXT`) après `ADD`. Dans le cas d'un index multicolonne, il suffit comme d'habitude d'indiquer toutes les colonnes entre parenthèses, séparées par des virgules.

Reprenons la table Test_tuto, utilisée pour tester `ALTER TABLE`, et ajoutons-lui un index sur la colonne *nom* :

```
ALTER TABLE Test_tuto
ADD INDEX ind_nom (nom);
```

Si vous affichez maintenant la description de votre table *Test_tuto*, vous verrez que, dans la colonne "Key", il est indiqué **MUL** pour la colonne *nom*. L'index a donc bien été

créé.

**Ajout avec CREATE INDEX**

```
CREATE INDEX nom_index
ON nom_table (colonne_index [, colonne2_index ...]);  -- Crée un index simple

CREATE UNIQUE INDEX nom_index
ON nom_table (colonne_index [, colonne2_index ...]);  -- Crée un index UNIQUE


CREATE FULLTEXT INDEX nom_index
ON nom_table (colonne_index [, colonne2_index ...]);  -- Crée un index FULLTEXT
```

**Exemple :** L'équivalent de la commande **ALTER TABLE** que nous avons utilisée pour ajouter un index sur la colonne *nom* est donc :

```
CREATE INDEX ind_nom
ON Test_tuto (nom);
```

**Complément pour la création d'un index UNIQUE - le cas des contraintes**

Vous vous rappelez, j'espère, que les index `UNIQUE` sont ce que l'on appelle des contraintes.

Or, lorsque vous créez un index `UNIQUE`, vous pouvez explicitement créer une contrainte. C'est fait automatiquement bien sûr si vous ne le faites pas, mais ne soyez donc pas surpris de voir apparaître le mot CONSTRAINT, c'est à cela qu'il se réfère.

Pour pouvoir créer explicitement une contrainte lors de la création d'un index `UNIQUE`, vous devez créer cet index soit lors de la création de la table, en listant l'index (et la contrainte) à la suite des colonnes, soit après la création de la table, avec la commande ALTER TABLE.

```
CREATE TABLE nom_table (
    colonne1 INT NOT NULL,
    colonne2 VARCHAR(40),
    colonne3 TEXT,
    CONSTRAINT [symbole_contrainte] UNIQUE [INDEX] ind_uni_col2 (colonne2)
);

ALTER TABLE nom_table
ADD CONSTRAINT [symbole_contrainte] UNIQUE ind_uni_col2 (colonne2);
```

Il n'est pas obligatoire de donner un symbole (un nom en fait) à la contrainte. D'autant plus que dans le cas des index, vous pouvez donner un nom à l'index (ici : `ind_uni_col`).

**Suppression d'un index**

```
ALTER TABLE nom_table
DROP INDEX nom_index;
```

Notez qu'il n'existe pas de commande permettant de modifier un index. Le cas échéant, il vous faudra supprimer, puis recréer votre index avec vos modifications.

---

# Clés primaires et étrangères

## Clés primaires

> *La clé primaire d'une table est une contrainte d'unicité, composée d'une ou plusieurs colonnes, et qui permet d'identifier de manière unique chaque ligne de la table.*

Examinons plus en détail cette définition :

- **Contrainte d'unicité** : ceci ressemble fort à un index `UNIQUE`.
- **Composée d'une ou plusieurs colonnes** : comme les index, les clés peuvent donc être composites.
- **Permet d'identifier chaque ligne de manière unique** : dans ce cas, une clé primaire ne peut pas être `NULL`.

Ces quelques considérations résument très bien l'essence des clés primaires. En gros, une clé primaire est un index `UNIQUE` sur une colonne qui ne peut pas être `NULL`. D'ailleurs, vous savez déjà que l'on définit une clé primaire grâce aux mots-clés PRIMARY KEY. Or, nous avons vu dans le précédent chapitre que `KEY` s'utilise pour définir un index. Par conséquent, lorsque vous définissez une clé primaire, pas besoin de définir en plus un index sur la ou les colonnes qui composent celle-ci, c'est déjà fait ! Et pas besoin non plus de rajouter une contrainte `NOT NULL`.

Pour le dire différemment, une contrainte de clé primaire est donc une combinaison de deux des contraintes que nous avons vues jusqu'à présent : `UNIQUE` et `NOT NULL`.

#### Choix de la clé primaire

Le choix d'une clé primaire est une étape importante dans la conception d'une table. Ce n'est pas parce que vous avez l'impression qu'une colonne, ou un groupe de colonnes, pourrait faire une bonne clé primaire que c'est le cas. Reprenons l'exemple d'une table *Client*, qui contient le nom, le prénom, la date de naissance et l'e-mail des clients d'une société.

Chaque client a bien sûr un nom et un prénom. Est-ce que (`nom`, `prenom`) ferait une bonne clé primaire ? Non : il est évident ici que vous risquez des doublons.
Et si l'on ajoute la date de naissance ? Les chances de doublons sont alors quasi nulles. Mais quasi nul, ce n'est pas nul... Qu'arrivera-t-il le jour où vous voyez débarquer un client qui a les mêmes nom et prénom qu'un autre, et qui est né le même jour ? On refait toute la base de données ? Non, bien sûr. Et l'e-mail alors ? Il est impossible que deux personnes aient la même adresse e-mail, donc la contrainte d'unicité est respectée. Par contre, tout le monde n'est pas obligé d'avoir une adresse e-mail. Difficile donc de mettre une contrainte `NOT NULL` sur cette colonne.

Par conséquent, on est bien souvent obligé d'ajouter une colonne pour jouer le rôle de la clé primaire. C'est cette fameuse colonne *id*, auto-incrémentée que nous avons déjà vue pour la table *Eleves*.

Il y a une autre raison d'utiliser une colonne spéciale auto-incrémentée, de type `INT` (ou un de ses dérivés) pour la clé primaire. En effet, si l'on définit une clé primaire, c'est en partie dans le but d'utiliser au maximum cette clé pour faire des recherches dans la table. Bien sûr, parfois, ce n'est pas possible, parfois vous ne connaissez pas l'*id* du client, et vous êtes obligé de faire une recherche par nom. Cependant, vous verrez bientôt que les clés primaires peuvent servir à faire des

recherches de manière **indirecte** sur la table. Comme les recherches sont beaucoup plus rapides sur des nombres que sur des textes, il est souvent intéressant d'avoir une clé primaire composée de colonnes de type `INT`.

Enfin, il y a également l'argument de l'auto-incrémentation. Si vous devez remplir vous-même la colonne de la clé primaire, étant donné que vous êtes humain, vous risquez de faire une erreur. Avec une clé primaire auto-incrémentée, vous ne risquez rien : MySQL fait tout pour vous. De plus, on ne peut définir une colonne comme auto-incrémentée que si elle est de type `INT` et qu'il existe un index dessus. Dans le cas d'une clé primaire auto-incrémentée, on définit généralement la colonne comme un entier `UNSIGNED`, comme on l'a fait pour la table *Eleves*.

> *Il peut bien sûr n'y avoir qu'**une seule clé primaire par table**. De même, une seule colonne peut être auto-incrémentée (la clé primaire en général).*

#### PRIMARY KEY or not PRIMARY KEY

D'un point de vue technique, avoir une clé primaire sur chaque table n'est pas obligatoire. Vous pourriez travailler toute votre vie sur une base de données sans aucune clé primaire, et ne jamais voir un message d'erreur à ce propos.
Cependant, d'un point de vue **conceptuel**, ce serait une grave erreur. Pensez donc à mettre une clé primaire sur chacune de vos tables.

#### Création d'une clé primaire

La création des clés primaires étant extrêmement semblable à la création d'index simples.

Donc, à nouveau, la clé primaire peut être créée en même temps que la table, ou par la suite.

#### Lors de la création de la table

On peut donc préciser `PRIMARY KEY` dans la description de la colonne qui doit devenir la clé primaire (pas de clé composite, dans ce cas) :

```
CREATE TABLE [IF NOT EXISTS] Nom_table (
    colonne1 description_colonne1 PRIMARY KEY [,
    colonne2 description_colonne2,
    colonne3 description_colonne3,
    ...,]
)
[ENGINE=moteur];
```

**Exemple** : création de la table Eleves en donnant la clé primaire dans la description de la colonne.

```
CREATE TABLE Eleves (
    id SMALLINT AUTO_INCREMENT PRIMARY KEY,
    nom VARCHAR(40) NOT NULL,
    sexe CHAR(1),
    date_naissance DATETIME NOT NULL,
    commentaires TEXT
)
ENGINE=InnoDB;
```

Ou bien, on ajoute la clé à la suite des colonnes.

```
CREATE TABLE [IF NOT EXISTS] Nom_table (
    colonne1 description_colonne1 [,
    colonne2 description_colonne2,
    colonne3 description_colonne3,
    ...],
    [CONSTRAINT [symbole_contrainte]] PRIMARY KEY (colonne_pk1 [, colonne_pk2, ...])
-- comme pour les index UNIQUE, CONSTRAINT est facultatif
)
[ENGINE=moteur];
```

**Exemple** : création d'Eleves.

```
CREATE TABLE Eleves (
    id SMALLINT AUTO_INCREMENT,
    nom VARCHAR(40) NOT NULL,
    sexe CHAR(1),
    date_naissance DATETIME NOT NULL,
    commentaires TEXT,
    PRIMARY KEY (id)
)
ENGINE=InnoDB;
```

**Après création de la table**

On peut toujours utiliser `ALTER TABLE`. Par contre, `CREATE INDEX` n'est pas utilisable pour les clés primaires.

```
ALTER TABLE nom_table
ADD [CONSTRAINT [symbole_contrainte]] PRIMARY KEY (colonne_pk1 [, colonne_pk2, ...]);
```

**Suppression de la clé primaire**

```
ALTER TABLE nom_table
DROP PRIMARY KEY
```

## Clés étrangères

Les clés étrangères ont pour fonction principale la vérification de l'intégrité de votre base. Elles permettent de s'assurer que vous n'insérez pas de bêtises...

Reprenons l'exemple dans lequel on a une table *Client* et une table *Commande*. Dans la table *Commande*, on a une colonne qui contient une référence au client. Ici, le client numéro 3, M. Nicolas Jacques, a donc passé une commande de trois tubes de colle, tandis que Mme Marie Malherbe (cliente numéro 2) a passé deux commandes, pour du papier et des ciseaux.

C'est bien joli, mais que se passe-t-il si M. Hadrien Piroux passe une commande de 15 tubes de colle, et qu'à l'insertion dans la table *Commande*, votre doigt dérape et met 45 comme numéro de client ? Vous avez dans votre base de données une commande passée par un client inexistant, et vous passez votre après-midi du lendemain à vérifier tous vos bons de commande de la veille pour retrouver qui a commandé ces 15 tubes de colle.

Ce serait quand même simple si, à l'insertion d'une ligne dans la table *Commande*, un programme allait vérifier que le numéro de client indiqué correspond bien à quelque chose dans la table *Client*.
Ce programme s'appelle "clé étrangère".

Par conséquent, si vous créez une clé étrangère sur la colonne client de la table *Commande* en lui donnant comme référence la colonne numero de la table *Client*, MySQL ne vous laissera plus jamais insérer un numéro de client inexistant dans la table *Commande*. Il s'agit bien d'une contrainte !

Quelques points **importants**.

- Comme pour les index et les clés primaires, il est possible de créer des **clés étrangères composites**.
- Lorsque vous créez une clé étrangère sur une colonne (ou un groupe de colonnes) – la colonne *client* de *Commande* dans notre exemple –, un index est **automatiquement ajouté** sur celle-ci (ou sur le groupe).
- Par contre, la colonne (le groupe de colonnes) qui sert de référence - la colonne *numero* de *Client* - **doit** déjà posséder un index (ou être clé primaire, bien sûr).
- La colonne (ou le groupe de colonnes) sur laquelle (lequel) la clé est créée doit être **exactement** du même type que la colonne (le groupe de colonnes) qu'elle (il) référence. Cela implique qu'en cas de clé composite, il y ait le même nombre de colonnes dans la clé et la référence. Donc, si *numero* (dans *Client*) est un `INT UNSIGNED`, client (dans *Commande*) doit être de type `INT UNSIGNED` aussi.
- Tous les moteurs de table ne permettent pas l'utilisation des clés étrangères. Par exemple, MyISAM ne le permet pas, contrairement à InnoDB.

**Création**

Une clé étrangère est un peu plus complexe à créer qu'un index ou une clé primaire, puisqu'il faut deux éléments :

- la ou les colonnes sur lesquelles on crée la clé - on utilise `FOREIGN KEY` ;
- la ou les colonnes qui vont servir de référence - on utilise `REFERENCES`.

**Lors de la création de la table**

Du fait de la présence de deux paramètres, une clé étrangère ne peut que s'ajouter à la suite des colonnes, et pas directement dans la description d'une colonne. Par ailleurs, je vous conseille ici de créer explicitement une contrainte (grâce au mot-clé `CONSTRAINT`) et de lui donner un symbole. En effet, pour les index, on pouvait utiliser leur nom pour les identifier ; pour les clés primaires, le nom de la table suffisait puisqu'il n'y en a qu'une par table. Par contre, pour différencier facilement les clés étrangères d'une table, il est utile de leur donner un nom, à travers la contrainte associée. À nouveau, je respecte certaines conventions de nommage : mes clés étrangères ont des noms commençant par fk (pour `FOREIGN KEY`), suivi du nom de la colonne dans la table, puis (si elle s'appelle différemment) du nom de la colonne de référence, le tout séparé par des `_` (*fk_client_numero, par exemple*).

```
CREATE TABLE [IF NOT EXISTS] Nom_table (
    colonne1 description_colonne1,
    [colonne2 description_colonne2,
    colonne3 description_colonne3,
```

```
    ...,]
    [ [CONSTRAINT [symbole_contrainte]]  FOREIGN KEY (colonne(s)_clé_étrangère)
REFERENCES table_référence (colonne(s)_référence)]
)
[ENGINE=moteur];
```

Donc si l'on imagine les tables *Client* et *Commande*, pour créer la table *Commande* avec une clé étrangère ayant pour référence la colonne numero de la table *Client*, on utilisera :

```
CREATE TABLE Commande (
    numero INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    client INT UNSIGNED NOT NULL,
    produit VARCHAR(40),
    quantite SMALLINT DEFAULT 1,
    CONSTRAINT fk_client_numero          -- On donne un nom à notre clé
        FOREIGN KEY (client)             -- Colonne sur laquelle on crée la clé
        REFERENCES Client(numero)        -- Colonne de référence
)
ENGINE=InnoDB;                          -- MyISAM interdit, je le rappelle encore une
fois !
```

#### Après création de la table

Tout comme pour les clés primaires, pour créer une clé étrangère après création de la table, il faut utiliser `ALTER TABLE`.

```
ALTER TABLE Commande
ADD CONSTRAINT fk_client_numero FOREIGN KEY (client) REFERENCES Client(numero);
```

#### Suppression d'une clé étrangère

Il peut y avoir plusieurs clés étrangères par table. Par conséquent, lors d'une suppression, il faut identifier la clé à détruire. Cela se fait grâce au symbole de la contrainte.

```
ALTER TABLE nom_table
DROP FOREIGN KEY symbole_contrainte
```

---

## Gestion des utilisateurs

### Voir les bases de données

```
SHOW DATABASES;
```

### Création, modification et suppression des utilisateurs

#### Syntaxe pour la création et la suppression

Voici les requêtes à utiliser pour créer et supprimer un utilisateur :

```
-- Création
CREATE USER 'login'@'hote' [IDENTIFIED BY 'mot_de_passe'];
```

```
-- Suppression
DROP USER 'login'@'hote';
```

**Utilisateur**

L'utilisateur est donc défini par deux éléments :

- `son login ;`
- `l'hôte à partir duquel il se connecte.`

**Login**

Le login est un simple identifiant. Vous pouvez le choisir comme vous voulez. Il n'est pas obligatoire de l'entourer de guillemets, sauf s'il contient des caractères spéciaux comme `-` ou `@`. C'est cependant conseillé.

**Hôte**

L'hôte est `l'adresse à partir de laquelle l'utilisateur va se connecter`. Si l'utilisateur se connecte à partir de la machine sur laquelle le serveur MySQL se trouve, on peut utiliser 'localhost'. Sinon, on utilise en général une adresse IP ou un nom de domaine.

**Exemples**

```
CREATE USER 'max'@'localhost' IDENTIFIED BY 'maxisthebest';
CREATE USER 'elodie'@'194.28.12.4' IDENTIFIED BY 'ginko1';
CREATE USER 'gabriel'@'arb.brab.net' IDENTIFIED BY 'chinypower';
```

Il est également possible de permettre à un utilisateur de se connecter à partir de plusieurs hôtes différents (sans devoir créer un utilisateur par hôte) : en utilisant le joker %, on peut préciser des noms d'hôtes partiels ou permettre à l'utilisateur de se connecter à partir de n'importe quel hôte.

**Exemples**

```
-- thibault peut se connecter à partir de n'importe quel hôte dont l'adresse IP
commence par 194.28.12.
CREATE USER 'thibault'@'194.28.12.%' IDENTIFIED BY 'basketball8';

-- joelle peut se connecter à partir de n'importe quel hôte du domaine brab.net
CREATE USER 'joelle'@'%.brab.net' IDENTIFIED BY 'singingisfun';

-- hannah peut se connecter à partir de n'importe quel hôte
CREATE USER 'hannah'@'%' IDENTIFIED BY 'looking4sun';
```

Comme pour le `login`, les guillemets ne sont pas obligatoires, sauf si un caractère spécial est utilisé (comme le `joker %`, par exemple). Notez que, si vous ne précisez pas d'hôte, c'est comme si vous autorisiez tous les hôtes. `'hannah'@'%'` est donc équivalent à `'hannah'`.

**Renommer l'utilisateur**

Pour modifier l'identifiant d'un utilisateur (login et/ou hôte), on peut utiliser `RENAME USER ancien_utilisateur TO nouvel_utilisateur`.

**Exemple** : on renomme max en maxime, en gardant le même hôte.

```
RENAME USER 'max'@'localhost' TO 'maxime'@'localhost';
```

#### Mot de passe

Le mot de passe de l'utilisateur est donné par la clause `IDENTIFIED BY`. Cette clause n'est pas obligatoire, auquel cas l'utilisateur peut se connecter sans donner de mot de passe. Ce n'est évidemment pas une bonne idée du point de vue de la sécurité. **Évitez au maximum les utilisateurs sans mot de passe**.

Lorsqu'un mot de passe est précisé, il n'est pas stocké tel quel dans la table mysql.user. Il est d'abord hashé, et c'est cette valeur hashée qui est stockée.

Modifier le mot de passe Pour modifier le mot de passe d'un utilisateur, on peut utiliser la commande `SET PASSWORD` (à condition d'avoir les privilèges nécessaires, ou d'être connecté avec l'utilisateur dont on veut changer le mot de passe). Cependant, cette commande ne hashe pas le mot de passe automatiquement. Il faut donc utiliser la fonction `PASSWORD()`.

#### Exemple

```
SET PASSWORD FOR 'thibault'@'194.28.12.%' = PASSWORD('basket8');
```

### Ajout et révocation de privilèges

#### Ajout de privilèges

Pour pouvoir ajouter un privilège à un utilisateur, il faut posséder le privilège `GRANT OPTION`. Pour l'instant, seul l'utilisateur "root" le possède. Étant donné qu'il s'agit d'un privilège un peu particulier, nous n'en parlerons pas tout de suite. Connectez-vous donc avec "root" pour exécuter les commandes de cette partie.

#### Syntaxe

La commande pour ajouter des privilèges à un utilisateur est la suivante :

```
GRANT privilege [(liste_colonnes)] [, privilege [(liste_colonnes)], ...]
ON [type_objet] niveau_privilege
TO utilisateur [IDENTIFIED BY mot_de_passe];
```

- `privilege` : le privilège à accorder à l'utilisateur ( `SELECT`, `CREATE VIEW`, `EXECUTE`...) ;
- `(liste_colonnes)` : facultatif - liste des colonnes auxquelles le privilège s'applique ;
- `niveau_privilege` : niveau auquel le privilège s'applique (\.*, nom_bdd.nom_table...*) ;
- `type_objet` : en cas de noms ambigus, il est possible de préciser à quoi se rapporte le niveau, `TABLE` ou `PROCEDURE`.

#### Révocation de privilèges

Pour retirer un ou plusieurs privilèges à un utilisateur, on utilise la commande `REVOKE`.

```
REVOKE privilege [, privilege, ...]
ON niveau_privilege
FROM utilisateur;
```

### Privilèges particuliers

#### ALL

Le privilège `ALL` (ou `ALL PRIVILEGES`), comme son nom l'indique, représente tous les privilèges. Accorder le privilège `ALL` revient donc à accorder tous les droits à l'utilisateur. Il faut évidemment préciser le niveau auquel tous les droits sont accordés (on octroie tous les privilèges possibles sur une table, ou sur une base de données, etc.).

> *Un privilège fait exception : `GRANT OPTION` n'est pas compris dans les privilèges représentés par `ALL`.*

**Exemple :** on accorde tous les droits sur la table *Client* à `'john'@'localhost'`.

```
GRANT ALL
ON bachelor3.Client
TO 'john'@'localhost';
```

#### USAGE

À l'inverse de `ALL`, le privilège `USAGE` signifie "aucun privilège". À première vue, utiliser la commande `GRANT` pour n'accorder aucun privilège peut sembler un peu ridicule. En réalité, c'est extrêmement utile : `USAGE` permet en fait de modifier les caractéristiques d'un compte avec la commande `GRANT`, sans modifier les privilèges du compte. Cet usage de la commande `GRANT` est cependant déconseillé. `USAGE` est toujours utilisé comme un privilège global (donc `ON *.*`).

#### GRANT OPTION

Nous voici donc au fameux privilège `GRANT OPTION`. Un utilisateur ayant ce privilège est autorisé à utiliser la commande `GRANT`, pour accorder des privilèges à d'autres utilisateurs. Ce privilège n'est pas compris dans le privilège `ALL`. Par ailleurs, un utilisateur ne peut accorder que les privilèges qu'il possède lui-même.

On peut accorder `GRANT OPTION` de deux manières :

- comme un privilège normal, après le mot `GRANT` ;
- à la fin de la commande `GRANT`, avec la clause `WITH GRANT OPTION`.

Exemple : on accorde les privilèges `SELECT`, `UPDATE`, `INSERT`, `DELETE` et `GRANT OPTION` sur la base de données *bachelor3* à `'joseph'@'localhost'`.

```
GRANT SELECT, UPDATE, INSERT, DELETE, GRANT OPTION
ON bachelor3.*
TO 'joseph'@'localhost' IDENTIFIED BY 'ploc4';

-- OU

GRANT SELECT, UPDATE, INSERT, DELETE
ON bachelor3.*
TO 'joseph'@'localhost' IDENTIFIED BY 'ploc4'
WITH GRANT OPTION;
```

---

## Informations sur la base de données et les requêtes

### Commandes de description

#### Description d'objets

```
SHOW objets;
```

Cette commande permet d'afficher une liste des objets, ainsi que certaines caractéristiques de ces objets.

**Exemple** : liste des tables et des vues

```
SHOW TABLES;
```

> *Pour pouvoir utiliser `SHOW TABLES`, il faut avoir sélectionné une base de données.*

#### Objets listables avec `SHOW`

Les tables et les vues ne sont pas les seuls objets que l'on peut lister avec la commande `SHOW`. Pour une liste exhaustive, je vous renvoie à la documentation officielle, mais voici quelques-uns de ces objets.

| Commande | Description |
|---|---|
| `SHOW CHARACTER SET` | Montre les sets de caractères (encodages) disponibles. |
| `SHOW [FULL] COLUMNS FROM nom_table [FROM nom_bdd]` | Liste les colonnes de la table précisée, ainsi que diverses informations (type, contraintes…). Il est possible de préciser également le nom de la base de données. En ajoutant le mot-clé `FULL`, les informations affichées pour chaque colonne sont plus nombreuses. |
| `SHOW DATABASES` | Montre les bases de données sur lesquelles on possède des privilèges (ou toutes si l'on possède le privilège global `SHOW DATABASES`). |
| `SHOW GRANTS [FOR utilisateur]` | Liste les privilèges de l'utilisateur courant, ou de l'utilisateur précisé par la clause `FOR` optionnelle. |
| `SHOW INDEX FROM nom_table [FROM nom_bdd]` | Liste les index de la table désignée. Il est possible de préciser également le nom de la base de données. |
| `SHOW PRIVILEGES` | Liste les privilèges acceptés par le serveur MySQL (dépend de la version de MySQL). |
| `SHOW PROCEDURE STATUS` | Liste les procédures stockées. |
| `SHOW [FULL] TABLES [FROM nom_bdd]` | Liste les tables de la base de données courante, ou de la base de données désignée par la clause `FROM`. Si `FULL` est utilisé, une colonne apparaîtra en plus, précisant s'il s'agit d'une vraie table ou d'une vue. |

| | |
|---|---|
| `SHOW TRIGGERS [FROM nom_bdd]` | Liste les triggers de la base de données courante, ou de la base de données précisée grâce à la clause `FROM`. |
| `` `SHOW [GLOBAL | SESSION] VARIABLES` `` | |
| `SHOW WARNINGS` | Liste les avertissements générés par la dernière requête effectuée. |

**DESCRIBE**

La commande `DESCRIBE nom_table`, qui affiche les colonnes d'une table ainsi que certaines de leurs caractéristiques, est en fait un raccourci pour `SHOW COLUMNS FROM nom_table`.

#### Requête de création d'un objet

La commande `SHOW` peut également montrer `la requête ayant servi à créer un objet`.

```
SHOW CREATE type_objet nom_objet;
```

### La base de données information_schema

La base de données *information_schema* contient des informations sur les schémas.

> *En MySQL, un schéma est une base de données. Ce sont des synonymes. La base information_schema contient donc des* ***informations sur les bases de données****.*

```
SHOW TABLES FROM information_schema;
```

### Déroulement d'une requête

`EXPLAIN` permet de décortiquer l'exécution d'une requête. Grâce à cette commande, il est possible de savoir quelles tables et quels index sont utilisés, et dans quel ordre.

L'utilisation de cette commande est extrêmement simple : il suffit d'ajouter `EXPLAIN` devant la requête que l'on désire examiner. `EXPLAIN` peut être utilisée pour les requêtes `SELECT`, `UPDATE`, `DELETE`, `INSERT` et `REPLACE`.

**Exemple**

```
EXPLAIN SELECT Eleves.nom, Nom.nom_courant AS nom, Classe.nom AS classe
FROM Eleves
INNER JOIN Nom ON Eleve.nom = Nom.id
LEFT JOIN Nom ON Eleve.classe = Classe.id
WHERE Eleve.id = 37;
```

---

## Annexes

- Cours OpenClassroom
- Documentation Officielle

---

*Réalisé en Markdown avec Dillinger - Par Nicolas Barbarisi & Alexandre Kramer*