

# Administration des BD Libres

## Partie 1 : MySQL et les bases du langage SQL

Vous avez de nombreuses données à traiter et vous voulez les organiser correctement, avec un outil adapté ?

Les bases de données ont été créées pour vous !

**MySQL**, qui est un Système de Gestion de Bases de Données Relationnelles (abrégé SGBDR). C'est-à-dire un logiciel qui permet de gérer des bases de données, et donc de gérer de grosses QL peut donc s'utiliser seul, mais est la plupart du temps combiné à un autre langage de programmation : PHP



MySQL avec l'interface PHPMyAdmin



MySQL avec une console windows

*Différentes façons d'utiliser MySQL*

## Quelques exemples d'applications

Vous gérez une boîte de location de matériel audiovisuel, et afin de toujours savoir où vous en êtes dans votre stock, vous voudriez un système informatique vous permettant de gérer les entrées et sorties de matériel, mais aussi éventuellement les données de vos clients. MySQL est une des solutions possibles pour gérer tout ça.

Vous voulez créer un site web dynamique en HTML/CSS/PHP avec un espace membre, un forum, un système de news ou même un simple livre d'or. Une base de données vous sera presque indispensable.

Vous créez un super logiciel en Java qui va vous permettre de gérer vos dépenses afin de ne plus jamais être à découvert, ou devoir vous affamer pendant trois semaines pour pouvoir payer le cadeau d'anniversaire du petit frère. Vous pouvez utiliser une base de données pour stocker les dépenses déjà effectuées, les dépenses à venir, les rentrées régulières, ...

Votre tantine éleveuse d'animaux voudrait un logiciel simple pour gérer ses bestioles, vous savez programmer en python et lui proposez vos services dans l'espoir d'avoir un top cadeau à Noël. Une base de données vous aidera à retenir que Poupouche le Caniche est né le 13 décembre 2007, que Sami le Persan a des poils blancs et que Igor la tortue est le dernier représentant d'une race super rare !

La conception et l'utilisation de bases de données est un vaste sujet, il a fallu faire des choix sur les thèmes à aborder. Voici les compétences que ce tutoriel vise à vous faire acquérir :

- Création d'une base de données et des tables nécessaires à la gestion des données
- Gestion des relations entre les différentes tables d'une base
- Sélection des données selon de nombreux critères
- Manipulation des données (modification, suppression, calculs divers)
- Utilisation des triggers et des procédures stockées pour automatiser certaines actions
- Utilisation des vues et des tables temporaires
- Gestion des utilisateurs de la base de données Et plus encore...

# Partie 1 : MySQL et les bases du langage SQL

Dans cette partie, vous commencerez par apprendre quelques définitions indispensables, pour ensuite installer MySQL sur votre ordinateur.

Les commandes de base de MySQL seront alors expliquées (création de tables, insertion, sélection et modification de données, etc.)

## Introduction

### Concepts de base

#### Base de données

Une base de données informatique est un **ensemble de données** qui ont été stockées sur un support informatique, et **organisées et structurées** de manière à pouvoir facilement consulter et modifier leur contenu.

Prenons l'exemple d'un site web avec un système de news et de membres. On va utiliser une base de données MySQL pour stocker toutes les données du site : les news (avec la date de publication, le titre, le contenu, éventuellement l'auteur,...) et les membres (leurs noms, leurs emails,...).

Tout ceci va constituer notre base de données pour le site. Mais il ne suffit pas que la base de données existe. Il faut aussi pouvoir **la gérer, interagir avec cette base**. Il faut pouvoir envoyer des messages à MySQL (messages qu'on appellera

"**requêtes**"), afin de pouvoir ajouter des news, modifier des membres, supprimer, et tout simplement afficher des éléments de la base.

Une base de données seule ne suffit donc pas, il est nécessaire d'avoir également :

- un **système permettant de gérer cette base** ;
- un **langage pour transmettre des instructions** à la base de données (par l'intermédiaire du système de gestion).

#### SGBD

Un Système de Gestion de Base de Données (SGBD) est un **logiciel** (ou un ensemble de logiciels) permettant de **manipuler les données d'une base de données**. Manipuler, c'est-à-dire sélectionner et afficher des informations tirées de cette base, modifier des données, en ajouter ou en supprimer (ce groupe de quatre opérations étant souvent appelé "CRUD", pour Create, Read, Update, Delete).

MySQL est un système de gestion de bases de données.

### Le paradigme client - serveur

La plupart des SGBD sont basés sur un **modèle Client - Serveur**. C'est-à-dire que la base de données se trouve sur un serveur qui ne sert qu'à ça, et pour interagir avec cette base de données, il faut utiliser un logiciel "client" qui va interroger le serveur et transmettre la réponse que le serveur lui aura donnée. Le serveur peut être installé sur une machine différente du client ; c'est souvent le cas lorsque les bases de données sont importantes. Ce n'est cependant pas obligatoire, ne sautez pas sur votre petit frère pour lui emprunter son ordinateur. Dans ce tutoriel, nous installerons les logiciels serveur et client sur un seul et même ordinateur.

## Partie 1 : MySQL et les bases du langage SQL

Par conséquent, lorsque vous installez un SGBD basé sur ce modèle (c'est le cas de MySQL), vous installez en réalité deux choses (au moins) : le serveur, et le client. Chaque requête (insertion/modification/lecture de données) est faite par l'intermédiaire du client. Jamais vous ne discuterez directement avec le serveur (d'ailleurs, il ne comprendrait rien à ce que vous diriez). Vous avez donc besoin d'un langage pour discuter avec le client, pour lui donner les requêtes que vous souhaitez effectuer. Dans le cas de MySQL, ce langage est le SQL.

## SGBDR

Le R de SGBDR signifie "**relationnel**". Un SGBDR est un SGBD qui implémente la théorie relationnelle. MySQL implémente la théorie relationnelle ; c'est donc un SGBDR.

La théorie relationnelle dépasse le cadre de ce tutoriel, mais ne vous inquiétez pas, il n'est pas nécessaire de la maîtriser pour être capable d'utiliser convenablement un SGBDR. Il vous suffit de savoir que dans un SGBDR, les données sont contenues dans ce qu'on appelle des **relations**, qui sont représentées sous forme de **tables**. Une relation est composée de deux parties, l'**en-tête** et le **corps**. L'en-tête est lui-même composé de plusieurs attributs. Par exemple, pour la relation "Client", on peut avoir l'en-tête suivant: **Numéro Nom Prénom Email**

Quant au corps, il s'agit d'un ensemble de **lignes** (ou n-uplets) composées d'autant d'éléments qu'il y a d'attributs dans le corps. Voici donc quatre lignes pour la relation "Client" :

Numéro	Nom	Prénom	Email
1	Jean	Dupont	jdupont@email.com
2	Marie	Malherbe	mama@email.com
3	Nicolas	Jacques	Jacques.nicolas@email.com
4	Hadrien	Piroux	happi@email.com

Différentes opérations peuvent alors être appliquées à ces **relations**, ce qui permet d'en tirer des informations. Parmi les opérations les plus utilisées, on peut citer (soient **A** et **B** deux relations) :

La sélection (ou restriction) : obtenir les lignes de **A** répondant à certains critères ; la projection : obtenir une partie des attributs des lignes de **A** ;

L'union  $A \cup B$  : obtenir tout ce qui se trouve dans la relation A ou dans la relation B ;

L'intersection  $A \cap B$  : obtenir tout ce qui se trouve à la fois dans la relation A et dans la relation B

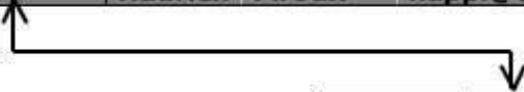
La différence  $A - B$  : obtenir ce qui se trouve dans la relation A mais pas dans la relation B ;

La jointure  $A \bowtie B$  : obtenir l'ensemble des lignes provenant de la liaison de la relation A et de

la relation B à l'aide d'une information commune.

Un petit exemple pour illustrer la jointure : si l'on veut stocker des informations sur les clients d'une société, ainsi que les commandes passées par ces clients, on utilisera deux relations : client et commande, la relation commande étant liée à la relation client par une référence au client ayant passé commande. Un petit schéma clarifiera tout ça !

Numéro	Nom	Prénom	Email
1	Jean	Dupont	jdupont@email.com
2	Marie	Malherbe	mama@email.com
3	Nicolas	Jacques	Jacques.nicolas@email.com
4	Hadrien	Piroux	happi@email.com



Numéro	Client	Produit	Quantité
1	3	Tube de colle	3
2	2	Rame de papier A4	6
3	2	Ciseaux	2

Le client numéro 3, M. Nicolas Jacques, a donc passé une commande de trois tubes de colle, tandis que M<sup>me</sup> Marie Malherbe (cliente numéro 2) a passé deux commandes, pour du papier et des ciseaux.

## Le langage SQL

Le SQL (*Structured Query Language*) est un **langage informatique** qui permet d'**interagir avec des bases de données relationnelles**. C'est le langage pour base de données le plus répandu, et c'est bien sûr celui utilisé par MySQL. C'est donc le langage que nous allons utiliser pour dire au client MySQL d'effectuer des opérations sur la base de données stockée sur le serveur MySQL.

Il a été créé dans les années 1970 et c'est devenu standard en 1986 (pour la norme ANSI - 1987 en ce qui concerne la norme ISO).

Il est encore régulièrement amélioré.

## Présentation succincte de MySQL...



MySQL est donc un Système de Gestion de Bases de Données Relationnelles, qui utilise le langage SQL. C'est un des SGBDR les plus utilisés. Sa popularité est due en grande partie au fait qu'il s'agit d'un logiciel Open Source, ce qui signifie que son code source est librement disponible et que quiconque qui en ressent l'envie et/ou le besoin peut modifier MySQL pour l'améliorer ou l'adapter à ses besoins. Une version gratuite de MySQL est par conséquent disponible. À

noter qu'une version commerciale payante existe également.

Le logo de MySQL est un dauphin, nommé Sakila suite au concours *Name the dolphin* ("Nommez le dauphin").

## Un peu d'histoire

Le développement de MySQL commence en 1994 par David Axmark et Michael Widenius. EN 1995, la société MySQL AB est fondée par ces deux développeurs, et Allan Larsson. C'est la même année que sort la première version officielle de MySQL.

En 2008, MySQL AB est rachetée par la société Sun Microsystems, qui est elle-même rachetée par Oracle Corporation en 2010.

On craint alors la fin de la gratuité de MySQL, étant donné qu'Oracle Corporation édite un des grands concurrents de MySQL : Oracle Database, qui est payant (et très cher). Oracle a cependant promis de continuer à développer MySQL et de conserver la double licence GPL (libre) et commerciale jusqu'en 2015 au moins.



### Mise en garde

MySQL est très utilisé, surtout par les débutants. Vous pourrez faire de nombreuses choses avec ce logiciel, et il convient tout à fait pour découvrir la gestion de bases de données. Sachez cependant que MySQL est loin d'être parfait. En effet, il ne suit pas toujours la norme officielle. Certaines syntaxes peuvent donc être propres à MySQL et ne pas fonctionner sous d'autres SGBDR. J'essayerai de le signaler lorsque le cas se présentera, mais soyez conscients de ce problème. Par ailleurs, il n'implémente pas certaines fonctionnalités avancées, qui pourraient vous être utiles pour un projet un tant soit peu ambitieux. Enfin, il est très permissif, et acceptera donc des requêtes qui génèreraient une erreur sous d'autres SGBDR.

### ... et de ses concurrents

Il existe des dizaines de SGBDR, chacun ayant ses avantages et ses inconvénients. Je présente ici succinctement quatre d'entre eux, parmi les plus connus. Je m'excuse tout de suite auprès des fans (et même simples utilisateurs) des nombreux SGBDR que j'ai omis.

### Oracle Database



Oracle, édité par Oracle Corporation (qui, je rappelle, édite également MySQL) est un SGBDR payant.

Son coût élevé fait qu'il est principalement utilisé par des entreprises.

Oracle gère très bien de grands volumes de données. Il est inutile d'acheter une licence oracle pour un projet de petite taille, car les performances ne seront pas bien différentes de celles de MySQL ou d'un autre SGBDR. Par contre, pour des projets conséquents (plusieurs centaines de Go de données), Oracle sera bien plus performant. Par ailleurs, Oracle dispose d'un langage procédural très puissant (du moins plus puissant que le langage procédural de MySQL) : le PL/SQL.

### PostgreSQL



Comme MySQL, PostgreSQL est un logiciel Open Source. Il est cependant moins utilisé, notamment par les débutants, car moins connu. La raison de cette méconnaissance réside sans doute en partie dans le fait que PostgreSQL a longtemps été disponible uniquement sous Unix. La première version Windows n'est apparue qu'à la sortie de la version 8.0 du logiciel, en 2005.

PostgreSQL a longtemps été plus performant que MySQL, mais ces différences tendent à diminuer. MySQL semble être aujourd'hui équivalent à PostgreSQL en terme de performances sauf pour quelques opérations telles que l'insertion de données et la création d'index. Le langage procédural utilisé par PostgreSQL s'appelle le PL/pgSQL.

### MS Access



MS Access ou Microsoft Access est un logiciel édité par Microsoft (comme son nom l'indique...) Par conséquent, c'est un logiciel payant qui ne fonctionne que sous Windows. Il n'est pas du tout adapté pour gérer un grand volume de données et a beaucoup moins de fonctionnalités que les autres SGBDR. Son avantage principal est l'interface graphique intuitive qui vient avec le logiciel.

### SQLite





La particularité de SQLite est de ne pas utiliser le schéma client-serveur utilisé par la majorité des SGBDR. SQLite stocke toutes les données dans de simples fichiers. Par conséquent, il ne faut pas installer de serveur de base de données, ce qui n'est pas toujours possible (certains hébergeurs web ne le permettent pas).

Pour de très petits volumes de données, SQLite est très performant. Cependant, le fait que les informations soient simplement stockées dans des fichiers rend le système difficile à sécuriser (autant au niveau des accès, qu'au niveau de la gestion de plusieurs utilisateurs utilisant la base simultanément).

## Organisation d'une base de données

Bon, vous savez qu'une base de données sert à gérer les données. Très bien. Mais comment ?? Facile ! Comment organisez-vous vos données dans la "vie réelle" ?? Vos papiers par exemple ? Chacun son organisation bien sûr, mais je suppose que vous les classez d'une manière ou d'une autre.

Toutes les factures ensemble, tous les contrats ensemble, etc. Ensuite on subdivise : les factures d'électricité, les factures pour la voiture. Ou bien dans l'autre sens : tous les papiers concernant la voiture ensemble, puis subdivision en taxes, communication avec l'assureur, avec le garagiste, ...

Une base de données, c'est pareil ! On classe les informations. MySQL étant un SGBDR, je ne parlerai que de l'organisation des bases de données relationnelles.

Comme je vous l'ai dit précédemment, on représente les données sous forme de **tables**. Une base va donc contenir plusieurs tables (elle peut n'en contenir qu'une bien sûr, mais c'est rarement le cas). Si je reprends mon exemple précédent, on a donc une table représentant des clients (donc des personnes).

Chaque table définit un certain nombre de **colonnes**, qui sont les caractéristiques de l'objet représenté par la table (les attributs de l'en-tête dans la théorie relationnelle). On a donc ici une colonne "Nom", une colonne "Prénom", une colonne "Email" et une colonne "Numéro" qui nous permettent d'identifier les clients individuellement (les noms et prénoms ne suffisent pas toujours).

Numéro	Nom	Prénom	Email
1	Jean	Dupont	jdupont@email.com
2	Marie	Malherbe	mama@email.com
3	Nicolas	Jacques	Jacques.nicolas@email.com
4	Hadrien	Piroux	happi@email.com

Si je récapitule, dans une base nous avons donc des **tables**, et dans ces **tables**, on a des **colonnes**. Dans ces tables, vous introduisez vos données. Chaque donnée introduite le sera sous forme de **ligne** dans une **table**, définissant la valeur de chaque **colonne** pour cette donnée.

### En résumé

- MySQL est un **Système de Gestion de Bases de Données Relationnelles** (SGBDR) basé sur le modèle **client-serveur**.
- Le **langage SQL** est utilisé pour communiquer entre le client et le serveur.
- Dans une base de données relationnelle, les données sont représentées sous forme de **tables**.

## Installation de MySQL

Maintenant qu'on sait à peu près de quoi on parle, il est temps d'installer MySQL sur l'ordinateur, et de commencer à l'utiliser. Au programme de ce chapitre :

- Installation de MySQL
- Connexion et déconnexion au client MySQL
- Création d'un utilisateur
- Bases de la syntaxe du langage SQL
- Introduction aux jeux de caractères et aux interclassements

### Avant-propos

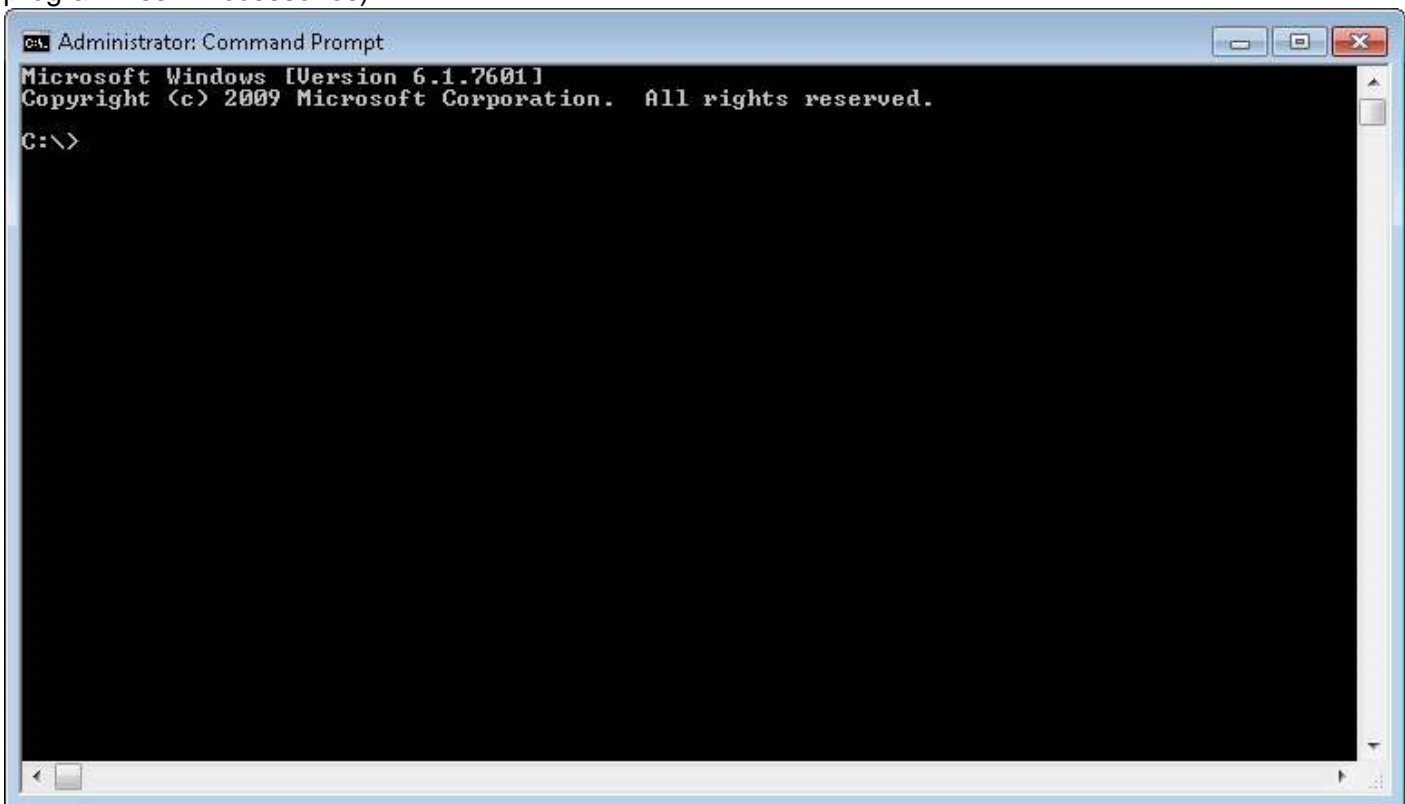
Il existe plusieurs manières d'utiliser MySQL. La première, que je vais utiliser tout au long du tutoriel, est l'utilisation en ligne de commande.

### Ligne de commande

#### Mais qu'est-ce donc ?

Eh bien il s'agit d'une fenêtre toute simple, dans laquelle toutes les instructions sont tapées à la main. Pas de bouton, pas de zone de saisie. Juste votre clavier.

Les utilisateurs de Linux connaissent très certainement. Pour Mac, il faut utiliser l'application "Terminal" que vous trouverez dans Applications > Utilitaires. Quant aux utilisateurs de Windows, c'est le "Command Prompt" que vous devez trouver (Démarrer > Tous les programmes > Accessoires).



### Interface graphique

Si l'on ne veut pas utiliser la ligne de commande (il faut bien avouer que ce n'est pas très sympathique cette fenêtre monochrome), on peut utiliser une interface graphique, qui permet d'exécuter pas mal de choses simples de manière intuitive sur une base de données.



## Partie 1 : MySQL et les bases du langage SQL

Comme interface graphique pour MySQL, on peut citer MySQL Workbench, PhpMyAdmin (souvent utilisé pour créer un site web en combinant MySQL et PHP) ou MySQL Front par exemple.

### Pourquoi utiliser la ligne de commande ?

C'est vrai ça, pourquoi ? Si c'est plus simple et plus convivial avec une interface graphique ?

Deux raisons :

- Primo, parce que je veux que vous maîtrisiez vraiment les commandes. En effet, les interfaces graphiques permettent de faire pas mal de choses, mais une fois que vous serez bien lancés, vous vous mettrez à faire des choses subtiles et compliquées, et il ne serait pas étonnant qu'il vous soit obligatoire d'écrire vous-mêmes vos requêtes ;
- Ensuite, parce qu'il est fort probable que vous désiriez utiliser MySQL en combinaison avec un autre langage de programmation (si ce n'est pas votre but immédiat, ça viendra probablement un jour). Or, dans du code PHP (ou Java, ou Python, etc.), on ne va pas écrire "Ouvre PhpMyAdmin et clique sur le bon bouton pour que je puisse insérer une donnée dans la base". On va devoir écrire en dur les requêtes. Il faut donc que vous sachiez comment faire.

Bien sûr, vous pouvez utiliser une interface graphique. Mais je vous encourage vivement à commencer par utiliser la ligne de commande, ou au minimum à faire l'effort de décortiquer les requêtes que vous laisserez l'interface graphique construire pour vous. Ceci afin de pouvoir les écrire vous-mêmes le jour où vous en aurez.

## Installation du logiciel

Pour télécharger MySQL, vous pouvez vous rendre sur le site suivant :

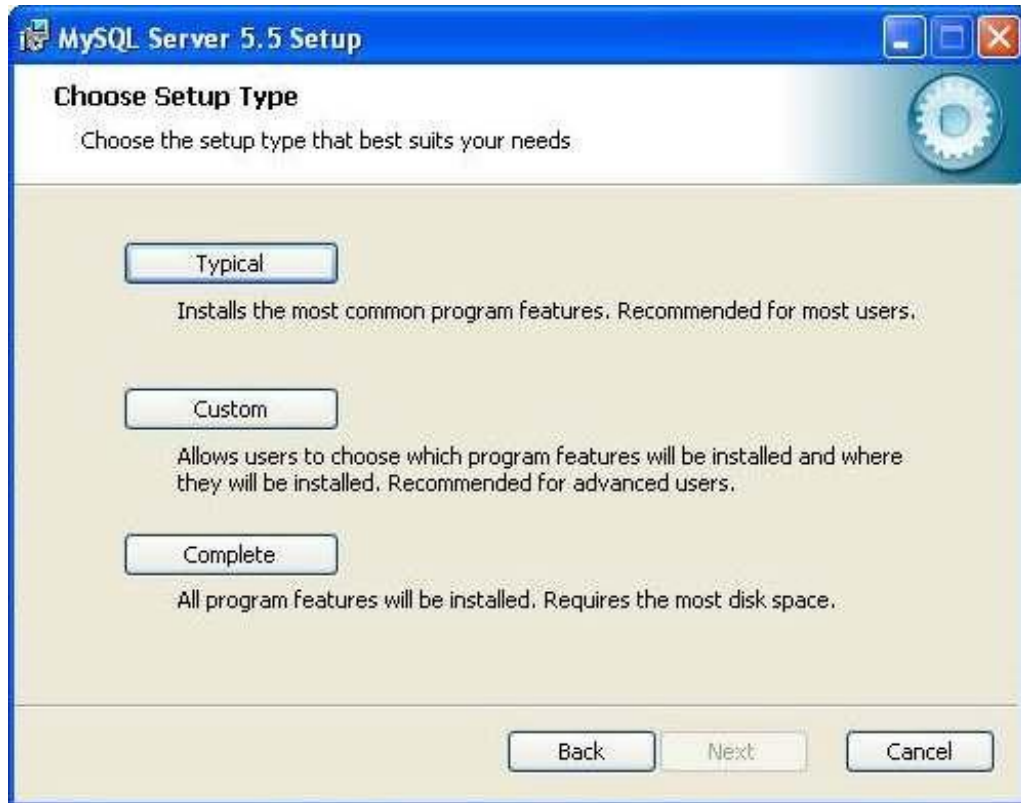
<http://dev.mysql.com/downloads/mysql/#downloads>

Sélectionnez l'OS sur lequel vous travaillez (Windows, Mac OS ou Linux).

### Windows

Téléchargez MySQL avec l'installateur (MSI Installer), puis exécutez le fichier téléchargé. L'installateur démarre et vous guide lors de l'installation.

Lorsqu'il vous demande de choisir entre trois types d'installation, choisissez "Typical". Cela installera tout ce dont nous pourrions avoir besoin.



L'installation se lance. Une fois qu'elle est terminée, cliquez sur "Terminer" **après vous être assurés** que la case "lancer l'outil de configuration MySQL" est cochée.



Dans cet outil de configuration, choisissez la configuration standard, et à l'étape suivante, cochez l'option "Include Bin Directory in Windows PATH"



On vous propose alors de définir un nouveau mot de passe pour l'utilisateur "root". Choisissez un mot de passe et confirmez-le. Ne cochez aucune autre option à cette étape. Cliquez ensuite sur "Execute" pour lancer la configuration.

## Mac OS

Téléchargez l'archive DMG qui vous convient (32 ou 64 bits), double-cliquez ensuite sur ce .dmg pour ouvrir l'image disque.

Vous devriez y trouver 4 fichiers dont deux .pkg. Celui qui nous intéresse s'appelle mysql-5.5.9-osx10.6-x86\_64.pkg (les chiffres peuvent changer selon la version de MySQL téléchargée et votre ordinateur). Ouvrez ce fichier qui est en fait l'installateur de MySQL, et suivez les instructions.

Une fois le programme installé, vous pouvez ouvrir votre terminal (pour rappel, il se trouve dans Applications -> Utilitaires).

Tapez les commandes et exécutez les instructions suivantes :

### Code : Console

```
cd /usr/local/mysql
sudo ./bin/mysqld_safe
```

- Entrez votre mot de passe si nécessaire
- Tapez Ctrl + Z

### Code : Console

## Partie 1 : MySQL et les bases du langage SQL

```
bg
```

- Tapez Ctrl + D
- Quittez le terminal

MySQL est prêt à être utilisé !

### Configuration

Par défaut, aucun mot de passe n'est demandé pour se connecter, même avec l'utilisateur root (qui a tous les droits). Je vous propose donc de définir un mot de passe pour cet utilisateur :

#### Code : Console

```
/usr/local/mysql/bin/mysqladmin -u root password <votre_mot_de_passe>
```

Ensuite, pour pouvoir accéder directement au logiciel client depuis la console, sans devoir aller dans le dossier où est installé le client, il vous faut ajouter ce dossier à votre variable d'environnement PATH. Pour cela, tapez la commande suivante dans le terminal :

#### Code : Console

```
echo 'export PATH=/usr/local/mysql/bin:$PATH' >> ~/.profile
```

/usr/local/mysql/bin est donc le dossier dans lequel se trouve le logiciel client (plusieurs logiciels clients en fait). Redémarrez votre terminal pour que le changement prenne effet.

## Linux

### Sous Debian ou Ubuntu

Exécuter la commande suivante pour installer MySQL :

#### Code : Console

```
sudo apt-get install mysql-server mysql-client
```

Une fois votre mot de passe introduit, MySQL va être installé.

### Sous RedHat

Exécuter la commande suivante pour installer MySQL :

#### Code : Console

```
sudo yum install mysql mysql-server
```

Une fois votre mot de passe introduit, MySQL va être installé.

## Partie 1 : MySQL et les bases du langage SQL

### Dans tous les cas, après installation

Pensez ensuite à modifier le mot de passe de l'utilisateur *root* (administrateur ayant tous les droits)

avec la commande suivante : **Code : Console**

```
sudo mysqladmin -u root -h localhost password '<votre mot de passe>'
```

## Connexion à MySQL

Je vous ai dit que MySQL était basé sur un modèle client - serveur, comme la plupart des SGBD. Cela implique donc que votre base de données se trouve sur un serveur auquel vous n'avez pas accès directement, il faut passer par un client qui fera la liaison entre vous et le serveur. Lorsque vous installez MySQL, plusieurs choses sont donc installées sur votre ordinateur :

- Un serveur de base de données MySQL ;
- Plusieurs logiciels clients qui permettent d'interagir avec le serveur.

## Connexion au client

Parmi ces clients, celui dont nous allons parler à présent est *mysql* (original comme nom). C'est celui que vous utiliserez tout au long de ce cours pour vous connecter à votre base de données et y insérer, consulter et modifier des données. La commande pour lancer le client est tout simplement son nom :

**Code : Console**

```
mysql
```

Cependant cela ne suffit pas. Il vous faut également préciser un certain nombre de paramètres. Le client *mysql* a besoin d'au minimum trois paramètres :

- l'hôte : c'est-à-dire l'endroit où est
- localisé le serveur ; le nom
- d'utilisateur ; et le mot de passe de l'utilisateur.

L'hôte et l'utilisateur ont des valeurs par défaut, et ne sont donc pas toujours indispensables. La valeur par défaut de l'hôte est "localhost", ce qui signifie que le serveur est sur le même ordinateur que le client. C'est bien notre cas, donc nous n'aurons pas à préciser ce paramètre. Pour le nom d'utilisateur, la valeur par défaut dépend de votre système. Sous Windows, l'utilisateur courant est "ODBC", tandis que pour les systèmes Unix (Mac et Linux), il s'agit de votre nom d'utilisateur (le nom qui apparaît dans l'invite de commande).

Pour votre première connexion à MySQL, il faudra vous connecter avec l'utilisateur "root", pour lequel vous avez normalement défini un mot de passe (si vous ne l'avez pas fait, inutile d'utiliser ce paramètre, mais ce n'est pas très sécurisé). Par la suite, nous créerons un nouvel utilisateur.

Pour chacun des trois paramètres, deux syntaxes sont possibles :

## Partie 1 : MySQL et les bases du langage SQL

### Code : Console

```
#####  
# Hôte #  
#####  
  
--hote=nom_hote  
  
# ou  
  
-h nom_hote  
  
#####  
# User #  
#####  
  
--user=nom_utilisateur  
  
# ou  
  
-u nom_utilisateur  
  
#####  
# Mot de passe #  
#####  
  
--password=password  
  
# ou  
  
-ppassword
```

Remarquez l'absence d'espace entre **-p** et le mot de passe. C'est voulu (mais uniquement pour ce paramètre-là), et souvent source d'erreurs.

La commande complète pour se connecter est donc :

### Code : Console

```
mysql -h localhost -u root -pmotdepasstopsecret  
  
# ou  
  
mysql --host=localhost --user=root --password=motdepasstopsecret  
  
# ou un mélange des paramètres courts et longs si ça vous amuse  
  
mysql -h localhost --user=root -pmotdepasstopsecret
```

J'utiliserai uniquement les paramètres courts à partir de maintenant. Choisissez ce qui vous convient le mieux.

Notez que pour le mot de passe, il est possible (et c'est même très conseillé) de préciser uniquement que vous utilisez le paramètre, sans lui donner de valeur :

### Code : Console

```
mysql -h localhost -u root -p
```

Apparaissent alors dans la console les mots suivants :

### Code : Console

Enter password:

Tapez donc votre mot de passe, et là, vous pouvez constater que les lettres que vous tapez ne s'affichent pas. C'est normal, cessez donc de martyriser votre clavier, il n'y peut rien le pauvre . Cela permet simplement de cacher votre mot de passe à d'éventuels curieux qui regarderaient par-dessus votre épaule.

Donc pour résumer, pour me connecter à mysql, je tape la commande suivante :

### Code : Console

```
mysql -u root -p
```

J'ai omis l'hôte, puisque mon serveur est sur mon ordinateur. Je n'ai plus qu'à taper mon mot de passe et je suis connecté.

## Déconnexion

Pour se déconnecter du client, il suffit d'utiliser la commande `quit` ou `exit`.

## Syntaxe SQL et premières commandes

Maintenant que vous savez vous connecter, vous allez enfin pouvoir discuter avec le serveur MySQL (en langage SQL évidemment). Donc, reconnectez-vous si vous êtes déconnectés.

Vous pouvez constater que vous êtes connectés grâce au joli (quoiqu'un peu formel) message de bienvenue, ainsi qu'au changement de l'invite de commande. On voit maintenant `mysql>`.

## "Hello World !"

Traditionnellement, lorsque l'on apprend un langage informatique, la première chose que l'on fait, c'est afficher le célèbre message "Hello World !". Pour ne pas déroger à la règle, je vous propose de taper la commande suivante (sans oublier le ; à la fin) :

### Code : SQL

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

**SELECT** est la commande qui permet la sélection de données, mais aussi l'affichage. Vous devriez donc voir s'afficher "Hello World !"

Hello World !
Hello World !

Comme vous le voyez, "Hello World !" s'affiche en réalité deux fois. C'est parce que MySQL représente les données sous forme de table. Il affiche donc une table avec une colonne, qu'il appelle "Hello World !" faute de meilleure information. Et dans cette table nous avons une ligne de données, le "Hello World !" que nous avons demandé.



### Syntaxe

Avant d'aller plus loin, voici quelques règles générales à retenir concernant le SQL qui, comme tout langage informatique, obéit à des règles syntaxiques très strictes.

#### Fin d'une instruction

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.

Par exemple, la commande suivante devrait afficher 100. Mais tant que MySQL ne recevra pas de ;, il attendra simplement la suite.

#### Code : SQL

```
SELECT 100
```

En appuyant sur la touche Entrée vous passez à la ligne suivante, mais la commande ne s'effectue pas. Remarquez au passage le changement dans l'invite de commande. `mysql>` signifie que vous allez entrer une commande, tandis que `>` signifie que vous allez entrer la suite d'une commande commencée précédemment.

Tapez maintenant ; puis appuyer sur Entrée. Ca y est, la commande est envoyée, l'affichage se fait !

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

Les commentaires sont des parties de code qui ne sont pas interprétées. Ils servent principalement à vous repérer dans votre code. 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

Afin de suivre au maximum la norme SQL, ce sont les -- qui seront utilisés tout au

long de ce tutoriel. **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).



MySQL permet également l'utilisation des guillemets doubles, mais ce n'est pas le cas de la plupart des SGBDR. Histoire de ne pas prendre de mauvaises habitudes, je vous conseille donc de n'utiliser que les guillemets simples pour délimiter vos chaînes de caractères.

**Exemple** : la commande suivante sert à afficher "Bonjour petit Zéro !"

## Partie 1 : MySQL et les bases du langage SQL

### Code : SQL

```
SELECT 'Bonjour petit Zéro !';
```

Par ailleurs, 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 : **Code : SQL**

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

Quelques autres caractères spéciaux :

\n	retour à la ligne
\t	tabulation
\	antislash (eh oui, il faut échapper le caractère d'échappement...)
%	pourcent (vous verrez pourquoi plus tard)
_	souligné (vous verrez pourquoi plus tard aussi)



*Cette manière d'échapper les caractères spéciaux (avec \) est propre à MySQL. D'autres SGBDR demanderont qu'on leur précise quel caractère sert à l'échappement ; d'autres encore demanderont de doubler le caractère spécial pour l'échapper. Soyez donc prudent et renseignez-vous si vous n'utilisez pas MySQL.*

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.

### Code : SQL

```
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 est également doué en calcul :

### Code : SQL

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

Pas de guillemets cette fois puisqu'il s'agit de nombres. MySQL calcule pour nous et nous affiche :

## Partie 1 : MySQL et les bases du langage SQL

$(5+3)*2$
16

MySQL est sensible à la priorité des opérations, comme vous pourrez le constater en

tapant cette commande : **Code : SQL**

```
SELECT (5+3) * 2, 5+3*2;
```

Résultat :

$(5+3)*2$	$5+3*2$
16	11

### Utilisateur

Il n'est pas très conseillé de travailler en tant que "root" dans MySQL, à moins d'en avoir spécifiquement besoin. En effet, "root" a tous les droits. Ce qui signifie que vous pouvez faire n'importe quelle bêtise dans n'importe quelle base de données pendant que j'ai le dos tourné. Pour éviter ça, nous allons créer un nouvel utilisateur, qui aura des droits très restreints. Je l'appellerai "sdz", mais libre à vous de lui donner le nom que vous préférez. Pour ceux qui sont sous Unix, notez que si vous créez un utilisateur du même nom que votre utilisateur Unix, vous pourrez dès lors omettre ce paramètre lors de votre connexion à mysql.

Je vous demande ici de me suivre aveuglément, car je ne vous donnerai que très peu d'explications. En effet, la gestion des droits et des utilisateurs fera l'objet d'un chapitre entier dans une prochaine partie du cours. Tapez donc cette commande dans mysql, en remplaçant sdz par le nom d'utilisateur que vous avez choisi, et mot\_de\_passe par le mot de passe que vous voulez lui attribuer :

**Code : SQL**

```
GRANT ALL PRIVILEGES ON elevage.* TO 'sdz'@'localhost' IDENTIFIED BY 'mot_de_passe';
```

Je décortique donc rapidement :

- **GRANT ALL PRIVILEGES** : Cette commande permet d'attribuer tous les droits (c'est-à-dire insertions de données, sélections, modifications, suppressions...)
- **ON elevage.\*** : définit les bases de données et les tables sur lesquelles ces droits sont acquis. Donc ici, on donne les droits sur la base "elevage" (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 'sdz'** : définit l'utilisateur auquel on accorde ces droits. Si l'utilisateur n'existe pas, il est créé.
- **@'localhost'** : définit à partir d'où l'utilisateur peut exercer ces droits. 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.

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

**Code : Console**

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

### En résumé

- MySQL peut s'utiliser en ligne de commande ou avec une interface graphique.
- Pour se connecter à MySQL en ligne de commande, on utilise : `mysql -u utilisateur [-h hôte] -p`.
- Pour terminer une instruction SQL, on utilise le caractère **;(point-virgule)**.
- En SQL, les chaînes de caractères doivent être entourées de guillemets simples '.
- Lorsque l'on se connecte à MySQL, il faut définir l'encodage utilisé, soit directement dans la connexion avec l'option `-default-character-set`, soit avec la commande **SET NAMES**.

## Les types de données

Nous avons vu dans l'introduction qu'une base de données contenait des **tables** qui, elles-mêmes sont 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 valeur 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, vous en conviendrez. Réfléchissez donc bien aux types de vos champs.

### L'attribut UNSIGNED

Vous pouvez également préciser que vos colonnes sont UNSIGNED, c'est-à-dire qu'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). Notez bien que cela ne change pas les capacités de stockage dans la colonne. Si vous déclarez un INT(2), vous pourrez toujours y stocker 45282 par exemple. Simplement, si vous stockez un nombre avec un nombre de chiffres inférieur au nombre défini, le caractère par défaut sera ajouté à gauche du chiffre, pour qu'il prenne la bonne taille. Sans précision, le caractère par défaut est l'espace.



Soyez prudents cependant. Si vous stockez des nombres dépassant la taille d'affichage définie, il est possible que vous ayez des problèmes lors de l'utilisation de ces nombres, notamment pour des jointures (nous le verrons dans la deuxième partie).

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

#### Code : SQL

```
INT(4) ZEROFILL
```

Vous aurez l'affichage suivant :

Nombre stock Nombre affiché	
45	0045
4156	4156
785164	785164

### Nombres décimaux

Cinq mots-clés permettent de stocker des nombres décimaux dans une colonne : DECIMAL, NUMERIC, FLOAT, REAL et 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 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).

## Partie 1 : MySQL et les bases du langage SQL



En SQL pur, on ne peut pas stocker dans un champ `DECIMAL(5,3)` un nombre supérieur à 99.999, puisque le nombre ne peut avoir que deux chiffres avant la virgule (5 chiffres en tout, dont 3 après la virgule,  $5-3 = 2$  avant). Cependant, MySQL permet en réalité de stocker des nombres allant jusqu'à 999.999. En effet, dans le cas de nombres positifs, MySQL utilise l'octet qui sert à stocker le signe - pour stocker un chiffre supplémentaire.

Comme pour les nombres entiers, si l'on entre un nombre qui n'est pas dans l'intervalle supporté par la colonne, MySQL le remplacera par le plus proche supporté. Donc si la colonne est définie comme un `DECIMAL(5,3)` et que le nombre est trop loin dans les positifs (1012,43 par exemple), 999.999 sera stocké, et -99.999 si le nombre est trop loin dans les négatifs. S'il y a trop de chiffres après la virgule, MySQL arrondira à l'échelle définie.

### FLOAT, DOUBLE et REAL

Le mot-clé `FLOAT` peut s'utiliser sans paramètre, auquel cas quatre 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. Je vous conseille donc d'utiliser `DOUBLE` pour éviter les surprises en cas de changement de SGBDR. *Valeurs exactes vs. valeurs approchées*

Les nombres stockés en tant que `NUMERIC` ou `DECIMAL` sont stockés sous forme de chaînes de caractères. Par conséquent, c'est la valeur exacte qui est stockée. Par contre, les types `FLOAT`, `DOUBLE` et `REAL` sont stockés sous forme de nombres, et c'est une valeur approchée qui est stockée.

Cela signifie que si vous stockez par exemple 56,6789 dans une colonne de type `FLOAT`, en réalité, MySQL stockera une valeur qui se rapproche de 56,6789 (par exemple, 56,67890000000000000001). Cela peut poser problème pour des comparaisons notamment (56,67890000000000000001 n'étant pas égal à 56,6789). S'il est nécessaire de conserver la précision exacte de vos données (l'exemple type est celui des données bancaires), il est donc conseillé d'utiliser un type numérique à valeur exacte (`NUMERIC` ou `DECIMAL` donc).



La documentation **anglaise** de MySQL donne des exemples de problèmes rencontrés avec les valeurs approchées. N'hésitez pas à y faire un tour si vous pensez pouvoir être concernés par ce problème, ou si vous êtes simplement curieux.

## Types alphanumériques

### Chaînes de type texte

#### CHAR et VARCHAR

Pour stocker un texte relativement court (moins de 255 octets), vous pouvez utiliser les types `CHAR` et `VARCHAR`. Ces deux types s'utilisent avec un paramètre qui précise la taille que peut prendre votre texte (entre 1 et 255). La différence entre `CHAR` et `VARCHAR` est la manière dont ils sont stockés en mémoire. Un `CHAR(x)` stockera toujours x octets, en remplissant si nécessaire le texte avec des espaces vides pour le compléter, tandis qu'un `VARCHAR(x)` stockera jusqu'à x octets (entre 0 et x), et stockera en plus en mémoire la taille du texte stocké.

Si vous entrez un texte plus long que la taille maximale définie pour le champ, celui-ci sera tronqué.



Je parle ici en **octets**, et non en caractères pour ce qui est de la taille des champs. C'est important : si la plupart des caractères sont stockés en mémoire sur un seul octet, ce n'est pas toujours le cas. Par exemple, lorsque l'on utilise l'encodage UTF-8, les caractères accentués (é, è, ...) sont codés sur deux octets.

## Partie 1 : MySQL et les bases du langage SQL

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

Text 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 octets (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)**.

### TEXT



Et si je veux pouvoir stocker des textes de plus de 255 octets ?

Il suffit alors d'utiliser le type **TEXT**, ou un de ses dérivés **TINYTEXT**, **MEDIUMTEXT** ou **LONGTEXT**. La différence entre ceux-ci étant la place qu'ils permettent d'occuper en mémoire. Petit tableau habituel :

Type	Longueur maximale	Mémoire occupée
TINYTEXT	2 <sup>8</sup> octets	Longueur de la chaîne + 1 octet
<b>TEXT</b>	2 <sup>16</sup> octets	Longueur de la chaîne + 2 octets
MEDIUMTEXT	2 <sup>24</sup> octets	Longueur de la chaîne + 3 octets
LONGTEXT	2 <sup>32</sup> 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 maximum (avec une gestion de la



## Partie 1 : MySQL et les bases du langage SQL

mémoire identique à **VARCHAR**(x) et **CHAR**(x)). Pour les chaînes plus longues, il existe les types **TINYBLOB**, **BLOB**, **MEDIUMBLOB** et **LONGBLOB**, également avec les mêmes limites de stockage que les types **TEXT**.

## SET et ENUM

### ENUM

Une colonne de type **ENUM** est une colonne pour laquelle on définit un certain nombre de valeurs autorisées, de type "chaîne de caractère". Par exemple, si l'on définit une colonne *espece* (pour une espèce animale) de la manière suivante : **Code : SQL**

```
espece ENUM('chat', 'chien', 'tortue')
```

La colonne *espece* pourra alors contenir les chaînes "chat", "chien" ou "tortue", mais pas les chaînes "lapin" ou "cheval".

En plus de "chat", "chien" et "tortue", la colonne *espece* pourrait prendre deux autres valeurs :

- si vous essayez d'introduire une chaîne non-autorisée, MySQL stockera une chaîne vide "" dans le champ ;
- si vous autorisez le champ à ne pas contenir de valeur (vous verrez comment faire ça dans le chapitre sur la création des tables), le champ contiendra **NULL**, qui correspond à "pas de valeur" en SQL (et dans beaucoup de langages informatiques).

Pour remplir un champ de type **ENUM**, deux possibilités s'offrent à vous :

- soit remplir directement avec la valeur choisie ("chat", "chien" ou "tortue" dans notre exemple) ; soit utiliser l'index de la valeur, c'est-à-dire le nombre associé par MySQL à la valeur. Ce nombre est compris entre 1 et le nombre de valeurs définies. L'index est attribué selon l'ordre dans lequel les valeurs ont été données lors de la création du champ. De plus, la chaîne vide (stockée en cas de valeur non-autorisée) correspond à l'index 0. Le tableau suivant reprend les valeurs d'index pour notre exemple précédent : le champ *espece*.

Valeur	Index
<b>NULL</b>	<b>NULL</b>
"	0
'chat'	1
'chien'	2
'tortue'	3

Afin que tout soit bien clair : si vous voulez stocker "chien" dans votre champ, vous pouvez donc y insérer "chien" ou insérer 2 (sans guillemets, il s'agit d'un nombre, pas d'un caractère).



**Un ENUM peut avoir maximum 65535 valeurs possibles**

### SET

**SET** est fort semblable à **ENUM**. Une colonne **SET** est en effet une colonne qui permet de stocker une chaîne de caractères dont les valeurs possibles sont prédéfinies par l'utilisateur. La différence avec **ENUM**, c'est qu'on peut stocker dans la colonne entre 0 et x valeur(s), x étant le nombre de valeurs autorisées.

## Partie 1 : MySQL et les bases du langage SQL

Donc, si l'on définit une colonne de type **SET** de la manière suivante :

Code : SQL

```
espece SET('chat', 'chien', 'tortue')
```

On pourra stocker dans cette colonne :

- '' (chaîne vide) ;
- 'chat' ;
- 'chat,tortue' ;
- 'chat,chien,tortue' ;
- 'chien,tortue' ;
- ...

Vous remarquerez que lorsqu'on stocke plusieurs valeurs, il faut les séparer par une virgule, sans espace et entourer la totalité des valeurs par des guillemets (non pas chaque valeur séparément). Par conséquent, les valeurs autorisées d'une colonne **SET** ne peuvent pas contenir de virgule elles-mêmes.



**On ne peut pas stocker la même valeur plusieurs fois dans un SET. "chien, chien" par exemple, n'est donc pas valable.**

Les colonnes **SET** utilisent également un système d'index, quoiqu'un peu plus complexe que pour le type ENUM. **SET** utilise en effet un système d'index binaire. Concrètement, la présence/absence des valeurs autorisées va être enregistrée sous forme de bits, mis à 1 si la valeur correspondante est présente, à 0 si la valeur correspondante est absente. Si l'on reprend notre exemple, on a donc :

Code : SQL

```
espece SET('chat', 'chien', 'tortue')
```

Trois valeurs sont autorisées. Il nous faut donc trois bits pour savoir quelles valeurs sont stockées dans le champ. Le premier, à droite, correspondra à "chat", le second (au milieu) à "chien" et le dernier (à gauche) à "tortue".

- 000 signifie qu'aucune valeur n'est présente.
- 001 signifie que 'chat' est présent.
- 100 signifie que 'tortue' est présent.
- 110 signifie que 'chien' et 'tortue' sont présents.
- ...

Par ailleurs, ces suites de bits représentent des nombres en binaire convertibles en décimal. Ainsi 000 en binaire correspond à 0 en nombre décimal, 001 correspond à 1, 010 correspond à 2, 011 à 3...

Puisque j'aime bien les tableaux, je vous en fais un, ce sera peut-être plus clair.

## Partie 1 : MySQL et les bases du langage SQL

Valeur	Binaire	Décimal
'chat'	001	1
'chien'	010	2
'tortue'	100	4

Pour stocker 'chat' et 'tortue' dans un champ, on peut donc utiliser 'chat,tortue' ou 101 (addition des nombres binaires correspondants) ou 5 (addition des nombres décimaux correspondants).

Notez que cette utilisation des binaires a pour conséquence que l'ordre dans lequel vous rentrez vos valeurs n'a pas d'importance. Que vous écriviez 'chat,tortue' ou 'tortue,chat' ne fait aucune différence. Lorsque vous récupérerez votre champ, vous aurez 'chat,tortue' (dans le même ordre que lors de la définition du champ).



**Un champ de type SET peut avoir au plus 64 valeurs définies**

### Avertissement

**SET** et ENUM sont des types **propres à MySQL**. Ils sont donc à utiliser avec une grande prudence !



**Pourquoi avoir inventé ces types propres à MySQL ?**

La plupart des SGBD implémentent ce qu'on appelle des contraintes d'assertions, qui permettent de définir les valeurs que peuvent prendre une colonne (par exemple, on pourrait définir une contrainte pour une colonne contenant un âge, devant être compris entre 0 et 130).

MySQL n'implémente pas ce type de contrainte et a par conséquent créé deux types de données spécifiques (**SET** et ENUM), pour pallier en partie ce manque.



**Dans quelles situations faut-il utiliser ENUM ou SET ?**

La meilleure réponse à cette question est : **jamais** ! Je déconseille fortement l'utilisation des **SET** et des ENUM. Je vous ai présenté ces deux types par souci d'exhaustivité, mais il faut toujours éviter autant que possible les fonctionnalités propres à un seul SGBD. Ceci afin d'éviter les problèmes si un jour vous voulez en utiliser un autre.

Mais ce n'est pas la seule raison. Imaginez que vous vouliez utiliser un ENUM ou un **SET** pour un système de catégories. Vous avez donc des éléments qui peuvent appartenir à une catégorie (dans ce cas, vous utilisez une colonne ENUM pour la catégorie) ou appartenir à plusieurs catégories (et vous utilisez **SET**).

**Code : SQL**

```
categorie ENUM('Soupes', 'Viandes', 'Tarte', 'Dessert')
categorie SET('Soupes', 'Viandes', 'Tarte', 'Dessert')
```

Tout se passe plutôt bien tant que vos éléments appartiennent aux catégories que vous avez définies au départ. Et puis tout à coup, vous vous retrouvez avec un élément qui ne correspond à aucune de vos catégories, mais qui devrait plutôt se trouver dans la catégorie "Entrées". Avec **SET** ou ENUM, il vous faut modifier la colonne *categorie* pour ajouter "Entrées" aux valeurs possibles. Or, une des règles de base à respecter lorsque l'on conçoit une base de données, est que **la**

## Partie 1 : MySQL et les bases du langage SQL

**structure de la base (donc les tables, les colonnes) ne doit pas changer lorsque l'on ajoute des données.** Par conséquent, tout ce qui est susceptible de changer doit être une donnée, et non faire partie de la structure de la base.

Il existe deux solutions pour éviter les ENUM, et une solution pour éviter les **SET**.

### Pour éviter ENUM

- Vous pouvez faire de la colonne *categorie* une simple colonne **VARCHAR(100)**. Le désavantage est que vous ne pouvez pas limiter les valeurs entrées dans cette colonne. Cette vérification pourra éventuellement se faire à un autre niveau (par exemple au niveau du PHP si vous faites un site web avec PHP et MySQL).
- Vous pouvez aussi ajouter une table *Categorie* qui reprendra toutes les catégories possibles. Dans la table des éléments, il suffira alors de stocker une référence vers la catégorie de l'élément.

### Pour éviter **SET**

La solution consiste en la création de deux tables : une table *Categorie*, qui reprend les catégories possibles, et une table qui lie les éléments aux catégories auxquels ils appartiennent.

## Types temporels

Pour les données temporelles, MySQL dispose de cinq types qui permettent, lorsqu'ils sont bien utilisés, de faire énormément de choses.

Avant d'entrer dans le vif du sujet, une petite remarque importante : lorsque vous stockez une date dans MySQL, certaines vérifications sont faites sur la validité de la date entrée. Cependant, ce sont des vérifications de base : le jour doit être compris entre 1 et 31 et le mois entre 1 et 12. Il vous est tout à fait possible d'entrer une date telle que le 31 février 2011. Soyez donc prudents avec les dates que vous entrez et récupérez.

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, l'ordre des données est la seule contrainte. 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. S'il s'agit d'une chaîne de caractères, n'importe quelle ponctuation peut être utilisée pour délimiter les parties (ou aucune). 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 :

## Partie 1 : MySQL et les bases du langage SQL

si l'année donnée est entre 00 et 69, on utilisera le 21<sup>e</sup> siècle, on ira donc de 2000 à 2069 ; par contre, si l'année est comprise entre 70 et 99, on utilisera le 20<sup>e</sup> 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. Pour entrer un DATETIME, c'est le même principe que pour **DATE** : pour la date, année-mois-jour, et pour l'heure, il faut donner d'abord l'heure, ensuite les minutes, puis les secondes. Si 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

Si vous n'avez besoin de retenir que l'année, **YEAR** est un type intéressant 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. Ceci dit, ça 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, 0h0min0s (TUC) et la date en question.

Les timestamps étant stockés sur 4 octets, il existe une limite supérieure : le 19 janvier 2038 à 3h14min7s. Par conséquent, vérifiez bien que vous êtes dans l'intervalle de validité avant d'utiliser un timestamp.

Le type **TIMESTAMP** de MySQL est cependant un peu particulier. Prenons par exemple le 4 octobre 2011, à 21h05min51s. Entre cette date et le 1er janvier 1970, 0h0min0s, il s'est écoulé exactement 1317755151 secondes. Le nombre 1317755151 est donc, par définition, le timestamp de cette date du 4 octobre 2011, 21h05min51s.

Pourtant, pour stocker cette date dans un **TIMESTAMP** SQL, ce n'est pas 1317755151 qu'on utilisera, mais 20111004210551. C'est-à-dire l'équivalent, au format numérique, du DATETIME '2011-10-04 21:05:51'.

Le **TIMESTAMP** SQL n'a donc de timestamp que le nom. Il ne sert pas à stocker un nombre de secondes, mais bien une date sous format numérique AAAAMMJJHHMMSS (alors qu'un DATETIME est donc stocké sous forme de chaîne de caractères).

Il n'est donc pas possible de stocker un "vrai" timestamp dans une colonne de type **TIMESTAMP**. C'est évidemment contre intuitif, et source d'erreur.

Notez que malgré cela, le **TIMESTAMP** SQL a les mêmes limites qu'un vrai timestamp : il n'acceptera que des date entre le 1e janvier 1970 à 00h00min00s et le 19 janvier 2038 à 3h14min7s.

### 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	0000000000000000

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).

#### En résumé

- MySQL définit plusieurs types de données : des numériques entiers, des numériques décimaux, des textes alphanumériques, des chaînes binaires alphanumériques et des données temporelles.
- Il est important de toujours utiliser le type de données adapté à la situation.

**SET** et **ENUM** sont des types de données qui n'existent que chez MySQL. Il vaut donc mieux éviter de les utiliser.

## **SQL DATE\_FORMAT()**

La fonction DATE\_FORMAT(), dans le langage SQL et plus particulièrement avec MySQL, permet de formater une donnée DATE dans le format indiqué. Il s'agit de la fonction idéal si l'ont souhaite définir le formatage de la date directement à partir du langage SQL et pas uniquement dans la partie applicative.

### **Syntaxe**

La fonction DATE\_FORMAT() s'utilise via la syntaxe suivante :

**SELECT DATE\_FORMAT(date, format)**

Le premier paramètre correspond à une donnée de type DATE (ou DATETIME), tandis que le deuxième paramètre est une chaîne de caractère contenant le choix de formatage :

- %a : nom du jour de la semaine abrégé (Sun, Mon, ... Sat)
- %b : nom du mois abrégé (Jan, Feb, ... Dec)
- %c : numéro du mois (0, 1, 2, ... 12) (numérique)
- %D : numéro du jour, avec le suffixe anglais (0th, 1st, 2nd, 3rd, ...)
- %d : numéro du jour du mois, sous 2 décimales (00..31) (numérique)
- %e : numéro du jour du mois (0..31) (numérique)
- %f : microsecondes (000000..999999)
- %H : heure (00..23)
- %h : heure (01..12)
- %I : heure (01..12)
- %i : minutes (00..59) (numérique)
- %j : jour de l'année (001..366)
- %k : heure (0..23)
- %l : heure (1..12)
- %M : nom du mois (January..December)
- %m : mois (00..12) (numérique)
- %p : AM ou PM
- %r : heure au format 12-heures (hh:mm:ss suivi par AM ou PM)
- %S : secondes (00..59)
- %s : secondes (00..59)
- %T : heure au format 24 heures (hh:mm:ss)
- %U : Semaine (00..53), pour lequel le dimanche est le premier jour de la semaine; WEEK() mode 0
- %u : Semaine (00..53), pour lequel le lundi est le premier jour de la semaine; WEEK() mode 1
- %V : Semaine (01..53), pour lequel le dimanche est le premier jour de la semaine; WEEK() mode 2; utilisé avec %X
- %v : Semaine (01..53), pour lequel le lundi est le premier jour de la semaine; WEEK() mode 3; utilisé avec %x
- %W : nom du jour de la semaine (Sunday, Monday, ... Saturday)
- %w : numéro du jour de la semaine (0=dimanche, 1=lundi, ... 6=samedi)
- %X : numéro de la semaine de l'année, pour lequel le dimanche est le premier jour de la semaine, numérique, 4 digits; utilisé avec %V
- %x : numéro de la semaine de l'année, pour lequel le lundi est le premier jour de la semaine, numérique, 4 digits; utilisé avec %v
- %Y : année, numérique (avec 4 digits)
- %y : année, numérique (avec 2 digits)
- %% : un caractère %
- %x x, pour n'importe lequel "x" non listé ci-dessus



### Exemple

Il est possible de combiner plusieurs valeurs ci-dessous afin d'obtenir le formatage d'une date dans le format de son choix. Voici une liste d'exemples d'usages de cette fonction DATE\_FORMAT() dans des requêtes SQL :

```
SELECT DATE_FORMAT("2018-09-24", "%D %b %Y");
```

-- résultat : "24th Sep 2018"

```
SELECT DATE_FORMAT("2018-09-24", "%M %d %Y");
```

-- résultat : "September 24 2018"

```
SELECT DATE_FORMAT("2018-09-24 22:21:20", "%W %M %e %Y");
```

-- résultat : "Monday September 24 2018"

```
SELECT DATE_FORMAT("2018-09-24", "%d/%m/%Y");
```

-- résultat : "24/09/2018"

```
SELECT DATE_FORMAT("2018-09-24", "Message du : %d/%m/%Y");
```

-- résultat : "Message du : 24/09/2018"

```
SELECT DATE_FORMAT("2018-09-24 22:21:20", "%H:%i:%s");
```

-- résultat : "22:21:20"

Les exemples ci-dessous ne sont que quelques cas d'usages, la richesse de la fonction DATE\_FORMAT() réside dans le fait qu'il est possible de combiner le formatage de la façon qui vous intéresse. La fonction est idéale pour s'adapter au formatage de date et heure dans des projets internationaux.

Exemple d'usage dans une requête SQL réelle :

```
SELECT *, DATE_FORMAT(date_inscription, "%d/%m/%Y")  
FROM utilisateur;
```

La requête ci-dessous permet d'obtenir la liste des données d'une table contenant des utilisateurs, ainsi que la date d'inscription avec un formatage jour/mois/année prêt à être exploité affiché dans un format facile à lire pour les utilisateurs.

# Création d'une base de données

Ça y est, le temps est venu d'écrire vos premières lignes de commande. Dans ce chapitre plutôt court, je vous donnerai pour commencer quelques conseils indispensables. Ensuite, je vous présenterai la problématique sur laquelle nous allons travailler tout au long de ce tutoriel : la base de données d'un élevage d'animaux. Pour finir, nous verrons comment créer et supprimer une base de données. La partie purement théorique est donc bientôt finie. Gardez la tête et les mains à l'intérieur du véhicule.

## Avant-propos : conseils et conventions

### Noms de tables et de colonnes

N'utilisez jamais, au grand 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". Avouez que ça reste lisible, et ça vous évitera pas mal d'ennuis.

Évitez également d'utiliser des mots réservés comme nom de colonnes/tables/bases. Par "mot réservé", j'entends un mot-clé SQL, donc un mot qui sert à définir quelque chose dans le langage SQL. Vous trouverez une liste exhaustive des mots réservés dans la [documentation officielle](#). Parmi les plus fréquents : **date**, **text**, **type**. Ajoutez donc une précision à vos noms dans ces cas-là (date\_naissance, text\_article ou type\_personnage par exemple).

Notez que MySQL permet l'utilisation de mots-clés comme noms de tables ou de colonnes, à condition que ce nom soit entouré de ` (accent grave/backquote). Cependant, ceci est propre à MySQL et ne devrait pas être utilisé. **Soyez cohérents**

Vous vous y retrouverez bien mieux si vous restez cohérents dans votre base. Par exemple, mettez tous vos noms de tables au singulier, ou au contraire au pluriel. Choisissez, mais tenez-vous-y. Même chose pour les noms de colonnes. Et lorsqu'un nom de table ou de colonne nécessite plusieurs mots, séparez les toujours avec '\_' (ex : date\_naissance) ou bien toujours avec une majuscule (ex : dateNaissance).

Ce ne sont que quelques exemples de situations dans lesquelles vous devez décider d'une marche à suivre, et la garder tout au long de votre projet (voire pour tous vos projets futurs). Vous gagnerez énormément de temps en prenant de telles habitudes.

## Conventions

### Mots-clés

Une convention largement répandue veut que les commandes et mots-clés SQL soient écrits complètement en majuscules. Je respecterai cette convention et vous encourage à le faire également. Il est plus facile de relire une commande de 5 lignes lorsqu'on peut différencier au premier coup d'œil les commandes SQL des noms de tables et de colonnes. **Noms de bases, de tables et de colonnes**

Je viens de vous dire que les mots-clés SQL seront écrits en majuscule pour les différencier du reste, donc évidemment, les noms de bases, tables et colonnes seront écrits en minuscule.

Toutefois, par habitude et parce que je trouve cela plus clair, je mettrai une majuscule à la première lettre de mes noms de tables (et uniquement pour les tables : ni pour la base de données ni pour les colonnes). Notez que MySQL n'est pas nécessairement sensible à la casse en ce qui concerne les noms de tables et de colonnes. En fait, il est très probable que si vous travaillez sous Windows, MySQL ne soit pas sensible à la casse pour les noms de tables et de colonnes. Sous Mac et Linux

## Partie 1 : MySQL et les bases du langage SQL

par contre, c'est le contraire qui est le plus probable. Quoi qu'il en soit, j'utiliserai des majuscules pour la première lettre de mes noms de tables. Libre à vous de me suivre ou non. **Options facultatives.**

Lorsque je commencerai à vous montrer les commandes SQL à utiliser pour interagir avec votre base de données, vous verrez que certaines commandes ont des options facultatives. Dans ces cas-là, j'utiliserai des crochets [ ] pour indiquer ce qui est facultatif. La même convention est utilisée dans la documentation officielle MySQL (et dans beaucoup d'autres documentations d'ailleurs). La requête suivante signifie donc que vous pouvez commander votre glace vanille toute seule, ou avec du chocolat, ou avec de la chantilly, ou avec du chocolat ET de la chantilly.

**Code : Autre**

```
COMMANDE glace vanille [avec chocolat] [avec chantilly]
```

## Mise en situation

Histoire que nous soyons sur la même longueur d'onde, je vous propose de baser le cours sur une problématique bien précise. Nous allons créer une base de données qui permettra de gérer un élevage d'animaux. Pourquoi un élevage ? Tout simplement parce que j'ai dû moi-même créer une telle base pour le laboratoire de biologie pour lequel je travaillais. Par conséquent, j'ai une assez bonne idée des problèmes qu'on peut rencontrer avec ce type de bases, et je pourrai donc appuyer mes explications sur des problèmes réalistes, plutôt que d'essayer d'en inventer.

Nous nous occupons donc d'un élevage d'animaux. On travaille avec plusieurs espèces : chats, chiens, tortues entre autres (tiens, ça me rappelle quelque chose). Dans la suite de cette partie, nous nous contenterons de créer une table *Animal* qui contiendra les caractéristiques principales des animaux présents dans l'élevage, mais dès le début de la deuxième partie, d'autres tables seront créées afin de pouvoir gérer un grand nombre de données complexes.

## Création et suppression d'une base de données

Nous allons donc créer notre base de données, que nous appellerons *elevage*. Rappelez-vous, lors de la création de votre utilisateur MySQL, vous lui avez donné tous les droits sur la base *elevage*, qui n'existait pas encore. Si vous choisissez un autre nom de base, vous n'aurez aucun droit dessus.

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

**Code : SQL**

```
CREATE DATABASE nom_base;
```

Avouez que je ne vous surmène pas le cerveau pour commencer...

Cependant, attendez avant de créer votre base de données *elevage*. 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 :

**Code : SQL**

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

## Partie 1 : MySQL et les bases du langage SQL

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

### Suppression

Si vous avez envie d'essayer cette commande, faites-le maintenant, tant qu'il n'y a rien dans votre base de données. Soyez très prudents, car vous effacez tous les fichiers créés par MySQL qui servent à stocker les informations de votre base.

#### Code : SQL

```
DROP DATABASE elevage;
```

Si vous essayez cette commande alors que la base de données *elevage* n'existe pas, MySQL

vous affichera une erreur : **Code : Console**

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

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

#### Code : SQL

```
DROP DATABASE IF EXISTS elevage;
```

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

#### Code : Console

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

Si elle n'existe pas, vous aurez :

#### Code : Console

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

Pour afficher les warnings de MySQL, il faut utiliser la commande **Code : SQL**

```
SHOW WARNINGS;
```

Cette commande affiche un tableau :

Level	Code	Message
Note	1008	Can't drop database 'elevage'; database doesn't exist

### Utilisation d'une base de données

Vous avez maintenant créé une base de données (si vous l'avez effacée avec **DROP DATABASE**, recréez-la). Mais pour pouvoir agir sur cette base, vous devez encore avertir MySQL que c'est bien sûr cette base-là que vous voulez travailler. Une fois de plus, la commande est très simple :

#### Code : SQL

```
USE elevage
```

C'est tout ! À partir de maintenant, toutes les actions effectuées le seront sur la base de données *elevage* (création et modification de tables par exemple).

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 :

#### Code : Console

```
mysql -u sdz -p elevage
```

### En résumé

- Pour créer une base de données, on utilise la commande **CREATE DATABASE** nom\_base.
- Pour supprimer une base de données : **DROP DATABASE** nom\_base.
- À chaque connexion à MySQL, il faut préciser avec quelle base on va travailler, avec **USE** nom\_base.

### Création de tables

Dans ce chapitre, nous allons créer, étape par étape, une table *Animal*, qui servira à stocker les animaux présents dans notre élevage.

Soyez gentils avec cette table, car c'est elle qui vous accompagnera tout au long de la première partie (on apprendra à jongler avec plusieurs tables dans la deuxième partie).

Pour commencer, il faudra définir de quelles colonnes (et leur type) la table sera composée. Ne négligez pas cette étape, c'est la plus importante. Une base de données mal conçue est un cauchemar à utiliser.

Ensuite, petit passage obligé par de la théorie : vous apprendrez ce qu'est une clé primaire et à quoi ça sert, et découvrirez cette fonctionnalité exclusive de MySQL que sont les moteurs de table.

Enfin, la table *Animal* sera créée, et la requête de création des tables décortiquée. Et dans la foulée, nous verrons également comment supprimer une table.

## 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 *Animal*. Qu'est-ce qui caractérise un animal ? Son espèce, son sexe, sa date de naissance. Quoi d'autre ? Une éventuelle colonne *commentaires* qui peut servir de fourre-tout. Dans le cas d'un élevage sentimental, on peut avoir donné un nom à nos bestioles. Disons que c'est tout pour le moment. Examinons donc les colonnes afin d'en choisir le type au mieux.

- **Espèce** : on a des chats, des chiens et des tortues pour l'instant. On peut donc caractériser l'espèce par un ou plusieurs mots. Ce sera donc un champ de type alphanumérique.

Les noms d'espèces sont relativement courts, mais n'ont pas tous la même longueur. On choisira donc un **VARCHAR**. Mais quelle longueur lui donner ? Beaucoup de noms d'espèces ne contiennent qu'un mot, mais "harfang des neiges", par exemple, en contient trois, et 18 caractères. Histoire de ne prendre aucun risque, autant autoriser jusqu'à 40 caractères pour l'espèce.

- **Sexe** : ici, deux choix possibles (mâle ou femelle). Le risque de voir un troisième sexe apparaître est extrêmement faible. Par conséquent, il serait possible d'utiliser un ENUM. Cependant, ENUM reste un type non standard. Pour cette raison, nous utiliserons plutôt une colonne **CHAR(1)**, contenant soit 'M' (mâle), soit 'F' (femelle).
- **Date de naissance** : pas besoin de réfléchir beaucoup ici. Il s'agit d'une date, donc soit un DATETIME, soit une **DATE**. L'heure de la naissance est-elle importante ? Disons que oui, du moins pour les soins lors des premiers jours. DATETIME donc !
- **Commentaires** : de nouveau un type alphanumérique évidemment, mais on a ici aucune idée de la longueur. Ce sera sans doute succinct mais il faut prévoir un minimum de place quand même. Ce sera donc un champ **TEXT**.

**Nom** : plutôt facile à déterminer. On prendra simplement un **VARCHAR(30)**. On ne pourra pas appeler nos tortues "Petite maison dans la prairie verdoyante", mais c'est amplement suffisant pour "Rox" ou "Roucky".

### NULL or NOT NULL ?

Il faut maintenant déterminer si l'on autorise les colonnes à ne pas stocker de valeur (ce qui est donc représenté par **NULL**).

## Partie 1 : MySQL et les bases du langage SQL

- **Espèce** : un éleveur digne de ce nom connaît l'espèce des animaux qu'il élève. On n'autorisera donc pas la colonne *espece* à être **NULL**.
- **Sexe** : le sexe de certains animaux est très difficile à déterminer à la naissance. Il n'est donc pas impossible qu'on doive attendre plusieurs semaines pour savoir si "Rox" est en réalité "Roxa". Par conséquent, la colonne *sexe* peut contenir **NULL**.
- **Date de naissance** : pour garantir la pureté des races, on ne travaille qu'avec des individus dont on connaît la provenance (en cas d'apport extérieur), les parents, la date de naissance. Cette colonne ne peut donc pas être **NULL**.
- **Commentaires** : ce champ peut très bien ne rien contenir, si la bestiole concernée ne présente absolument aucune particularité.
- **Nom** : en cas de panne d'inspiration (ça a l'air facile comme ça mais, une chatte pouvant avoir entre 1 et 8 petits d'un coup, il est parfois difficile d'inventer 8 noms originaux comme ça !), il vaut mieux autoriser cette colonne à être **NULL**.

### Récapitulatif

Comme d'habitude, un petit tableau pour récapituler tout ça :

Caractéristique	Nom de la colonne	Type	NULL?
Espèce	<i>espece</i>	VARCHAR(40)	Non
Sexe	<i>sexe</i>	CHAR(1)	Oui
Date de naissance	<i>date_naissance</i>	DATETIME	Non
Commentaires	<i>commentaires</i>	TEXT	Oui
Nom	<i>nom</i>	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

On va donc définir cinq colonnes : *espece*, *sexe*, *date\_naissance*, *commentaires* et *nom*. Ces colonnes permettront de caractériser nos animaux. Mais que se passe-t-il si deux animaux sont de la même espèce, du même sexe, sont nés exactement le même jour, et ont exactement les mêmes commentaires et le même nom ? Comment les différencier ? Évidemment, on pourrait s'arranger pour que deux animaux n'aient jamais le même nom. Mais imaginez la situation suivante : une chatte vient de donner naissance à sept petits. On ne peut pas encore définir leur sexe, on n'a pas encore trouvé de nom pour certains d'entre eux et il n'y a encore aucun commentaire à faire à leur propos. Ils auront donc exactement les mêmes caractéristiques. Pourtant, ce ne sont pas les mêmes individus. Il faut donc les différencier. Pour cela, on va ajouter une colonne à notre table.

### Identité

Imaginez que quelqu'un ait le même nom de famille que vous, le même prénom, soit né dans la même ville et ait la même taille. En dehors de la photo et de la signature, quelle sera la différence entre vos deux cartes d'identité ? Son numéro !

Suivant le même principe, on va donner à chaque animal un **numéro d'identité**. La colonne qu'on ajoutera s'appellera donc *id*, et il s'agira d'un **INT**, toujours positif donc **UNSIGNED**. Selon la taille de l'élevage (la taille actuelle mais aussi la taille qu'on imagine qu'il pourrait avoir dans le futur !), il peut être plus intéressant d'utiliser un **SMALLINT**, voire un **MEDIUMINT**. Comme il est peu probable que l'on dépasse les 65000 animaux, on utilisera **SMALLINT**. Attention, il faut bien considérer tous



les animaux qui entreront un jour dans la base, pas uniquement le nombre d'animaux présents en même temps dans l'élevage. En effet, si l'on supprime pour une raison ou une autre un animal de la base, il n'est pas question de réutiliser son numéro d'identité.

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 *Animal*, 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.

Par exemple, dans le cas où vous essayez d'insérer un *id* qui existe déjà, vous obtiendrez l'erreur suivante :

#### Code : Console

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

Je n'en dirai pas plus pour l'instant sur les clés primaires mais j'y reviendrai de manière détaillée dans la seconde partie de ce cours.

### Auto-incrémentation

Il faut donc, pour chaque animal, décider d'une valeur pour *id*. Le plus simple, et le plus logique, est de donner le numéro 1 au premier individu enregistré, puis le numéro 2 au second, etc.

Mais si vous ne vous souvenez pas quel numéro vous avez utilisé en dernier, pour insérer un nouvel animal il faudra récupérer cette information dans la base, ensuite seulement vous pourrez ajouter une ligne en lui donnant comme *id* le dernier *id* utilisé + 1. C'est bien sûr faisable, mais c'est fastidieux... Heureusement, il est possible de demander à MySQL de faire tout ça pour nous ! Comment ? En utilisant l'auto-incrémentation des colonnes. Incrémenter veut dire "ajouter une valeur fixée". 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 mettre dedans lors de la prochaine insertion. MySQL va chercher ça tout seul comme un grand en prenant la dernière valeur insérée et en l'incrémentant de 1.

### Les moteurs de tables

Les moteurs de tables sont une spécificité de MySQL. Ce sont des moteurs de stockage. Cela permet de gérer différemment les tables selon l'utilité qu'on en a. Je ne vais pas vous détailler tous les moteurs de tables existant. Si vous voulez plus d'informations, je vous renvoie à la [documentation officielle](#).

Les deux moteurs les plus connus sont **MyISAM** et **InnoDB**.

### MyISAM

C'est le moteur par défaut. Les commandes d'insertion et sélection de données sont particulièrement rapides sur les tables utilisant ce moteur. Cependant, il ne gère pas certaines fonctionnalités importantes comme les clés étrangères, qui permettent de vérifier l'intégrité d'une référence d'une table à une autre table (voir la deuxième partie du cours) ou les transactions, qui permettent de réaliser des séries de modifications "en bloc" ou au contraire d'annuler ces modifications (voir la cinquième partie du cours).

### InnoDB

Plus lent et plus gourmand en ressources que MyISAM, ce moteur gère les clés étrangères et les transactions. Étant donné que nous nous servirons des clés étrangères dès la deuxième partie, c'est celui-là que nous allons utiliser. De plus, en cas de crash du serveur, il possède un système de récupération automatique des données.

## Préciser un moteur lors de la création de la table

Pour qu'une table utilise le moteur de notre choix, il suffit d'ajouter ceci à la fin de la commande de création :

#### Code : SQL

```
ENGINE = moteur;
```

En remplaçant bien sûr "moteur" par le nom du moteur que nous voulons utiliser, ici InnoDB :

#### Code : SQL

```
ENGINE = INNODB;
```

## Syntaxe de CREATE TABLE

Avant de voir la syntaxe permettant de créer une table, résumons un peu. Nous voulons donc créer une table *Animal* avec six colonnes telles que décrites dans le tableau suivant.

Caractéristique	Nom du champ	Type	NULL?	Divers
Numéro d'identité	<i>id</i>	SMALLINT	Non	Clé primaire + auto-incrément + UNSIGNED
Espèce	<i>espece</i>	VARCHAR(40)	Non	-
Sexe	<i>sexe</i>	CHAR(1)	Oui	-
Date de naissance	<i>date_naissance</i>	DATETIME	Non	-
Commentaires	<i>commentaires</i>	TEXT	Oui	-
Nom	<i>nom</i>	VARCHAR(30)	Oui	-

## Syntaxe

Par souci de clarté, je vais diviser l'explication de la syntaxe de **CREATE TABLE** 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

Code : SQL

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

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.

Ce n'est pas non plus une erreur de ne pas préciser la clé primaire directement à la création de la table. Il est tout à fait possible de l'ajouter par la suite. Nous verrons comment un peu plus tard. [Définition des colonnes](#)

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

Code : SQL

```
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 :

Code : SQL

```
espece VARCHAR(40) NOT NULL,  
date_naissance DATETIME NOT NULL
```

L'auto-incrémentation se définit également à cet endroit. Notez qu'il est également possible de définir une colonne comme étant la clé primaire dans sa description. Il ne faut alors plus l'indiquer après la définition de toutes les colonnes. Je vous conseille néanmoins de ne pas l'indiquer à cet endroit, nous verrons plus tard pourquoi.

Code : SQL

```
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. Notez que si une colonne est autorisée à contenir **NULL** et qu'on ne précise pas de valeur par défaut, alors **NULL** est implicitement considéré comme valeur par défaut.

## Partie 1 : MySQL et les bases du langage SQL

Exemple :

Code : SQL

```
espece VARCHAR(40) NOT NULL DEFAULT 'chien'
```



Une valeur par défaut DOIT être une constante. Ce ne peut pas être une fonction (comme par exemple la fonction `NOW()` qui renvoie la date et l'heure courante).

### Application : création de *Animal*

Si l'on met tout cela ensemble pour créer la table *Animal* (je rappelle que nous utiliserons le moteur

InnoDB), on a donc : **Code : SQL**

```
CREATE TABLE Animal
(
  id SMALLINT UNSIGNED NOT NULL
    PRIMARY KEY,
  sexe CHAR(1),
  date_naissance DATE NOT NULL,
  nom VARCHAR(40),
  type TEXT,
  PRIMARY KEY
)
ENGINE=INNODB
```



Je n'ai pas gardé la valeur par défaut pour `type` car je trouve que ça n'a pas beaucoup de sens en contexte. C'était juste un exemple pour vous montrer la syntaxe.

### Vérifications

Au cas où vous ne me croiriez pas (et aussi un peu parce que cela pourrait vous être utile un jour), voici deux commandes vous permettant de vérifier que vous avez bien créé une jolie table *Animal* avec les six colonnes que vous vouliez.

Code : SQL

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

DESCRIBE Animal; -- 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. Elle est, bien sûr, à utiliser avec prudence, car irréversible.

Code : SQL

```
DROP TABLE Animal;
```

## Partie 1 : MySQL et les bases du langage SQL

### En résumé

- Avant de créer une table, il faut **définir ses colonnes**. Pour cela, il faut donc déterminer le type de chacune des colonnes et décider si elles peuvent ou non contenir **NULL** (c'est-à-dire ne contenir aucune donnée).
- Chaque table créée doit définir une **clé primaire**, donc une colonne qui permettra d'identifier chaque ligne de manière unique.

Le **moteur d'une table** définit la manière dont elle est gérée. Nous utiliserons le moteur **InnoDB**, qui permet notamment de définir des relations entre plusieurs tables.

## Modification d'une table

La création et la suppression de tables étant acquises, parlons maintenant des requêtes permettant de modifier une table. Plus précisément, ce chapitre portera sur la modification des colonnes d'une table (ajout d'une colonne, modification, suppression de colonnes).

Il est possible de modifier d'autres éléments (des contraintes, ou des index par exemple), mais cela nécessite des notions que vous ne possédez pas encore, aussi n'en parlerai-je pas ici.

Notez qu'idéalement, il faut penser à l'avance à la structure de votre base et créer toutes vos tables directement et proprement, de manière à ne les modifier qu'exceptionnellement.

### 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 :

#### Code : SQL

```
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
```

### Créons une table pour faire joujou

Dans la seconde partie de ce tutoriel, nous devons faire quelques modifications sur notre table *Animal*, mais en attendant, je vous propose d'utiliser la table suivante, si vous avez envie de tester les différentes possibilités d'**ALTER TABLE** :

#### Code : SQL

```
CREATAB
id IN NO NUL,
VARCH(1) NO NUL,
PRIMAKE(1)
);
```

### Ajout et suppression d'une colonne

#### Ajout

On utilise la syntaxe suivante :

#### Code : SQL

```
ALTERTABLE 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. Il contient le type de donnée et éventuellement **NULL** ou **NOT NULL**, etc.

Ajoutons une colonne *date\_insertion* à notre table de test. Il s'agit d'une date, donc une colonne de type **DATE** convient parfaitement. Disons que cette colonne ne peut pas être **NULL** (si c'est dans la table, ça a forcément été inséré). Cela nous donne

:

**Code : SQL**

```
ALTER TABLE Test_tuto
ADD COLUMN date_insertion DATE NOT NULL;
```

Un petit **DESCRIBE** Test\_tuto; vous permettra de vérifier les changements apportés.

### Suppression

La syntaxe de **ALTER TABLE ... DROP ...** est très simple : **Code**  
: **SQL**

```
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.

**Exemple** : nous allons supprimer la colonne *date\_insertion*, que nous remercions pour son passage éclair dans le cours. **Code : SQL**

```
ALTER TABLE Test_tuto
DROP COLUMN date_insertion; -- Suppression de la colonne
date_insertion
```

### Modification de colonne

#### Changement du nom de la colonne

Vous pouvez utiliser la commande suivante pour changer le nom d'une colonne :

**Code : SQL**

```
ALTER TABLE nom_table
CHANGE ancien_nom nouveau_nom description_colonne;
```

Par exemple, pour renommer la colonne *nom* en *prenom*, vous pouvez écrire

**Code : SQL**

```
ALTER TABLE Test_tuto
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**,



## Partie 1 : MySQL et les bases du langage SQL

vous pouvez, comme on vient de le voir, 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 :

### Code : SQL

```
ALTER TABLE nom_table  
CHANGE ancien_nom nouveau_nom nouvelle_description;  
  
ALTER TABLE nom_table  
MODIFY nom_colonne nouvelle_description;
```

Des exemples pour illustrer :

### Code : SQL

```
ALTER TABLE Test_tuto  
CHANGE prenom nom VARCHAR(30) NOT NULL; -- Changement du type + changement du  
nom  
  
ALTER TABLE Test_tuto  
CHANGE id id BIGINT NOT NULL; -- Changement du type sans renommer  
  
ALTER TABLE Test_tuto  
MODIFY id BIGINT NOT NULL AUTO_INCREMENT; -- Ajout de l'autoincrémentation  
  
ALTER TABLE Test_tuto  
MODIFY nom VARCHAR(30) NOT NULL DEFAULT 'Blabla'; -- Changement de la description  
(même type mais ajout d'une valeur par défaut)
```

Il existe pas mal d'autres possibilités et combinaisons pour la commande **ALTER TABLE** mais en faire la liste complète ne rentre pas dans le cadre de ce cours. Si vous ne trouvez pas votre bonheur ici, je vous conseille de le chercher dans la [documentation officielle](#).

### En résumé

- La commande **ALTER TABLE** permet de modifier une table
- Lorsque l'on ajoute ou modifie une colonne, il faut toujours préciser sa (nouvelle) description **complète** (type, valeur par défaut, auto-incrément éventuel)

## Insertion de données

Ce chapitre est consacré à l'insertion de données dans une table. Rien de bien compliqué, mais c'est évidemment crucial. En effet, que serait une base de données sans données ?

Nous verrons entre autres :

- comment insérer une ligne dans une table ; comment insérer plusieurs lignes dans une table ; comment exécuter des requêtes SQL écrites dans un fichier (requêtes d'insertion ou autres) ; comment insérer dans une table des lignes définies dans un fichier de format particulier.

Et pour terminer, nous peuplerons notre table *Animal* d'une soixantaine de petites bestioles sur lesquelles nous pourrions tester toutes sortes de ~~requêtes~~ requêtes dans la suite de ce tutoriel.

## 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

Je rappelle pour les distraits que notre table *Animal* est composée de six colonnes : *id*, *espece*, *sexe*, *date\_naissance*, *nom* et *commentaires*.

Voici donc la syntaxe à utiliser pour insérer une ligne dans *Animal*, 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).

#### Code : SQL

```
INSERT INTO Animal
VALUES (1, 'chien', 'M', '2010-04-05 13:43:00', 'Rox', 'Mordille
beaucoup');
```

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

#### Code : SQL

```
INSERT INTO Animal
VALUES (2, 'chat', NULL, '2010-03-24 02:23:00', 'Roucky', 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 comme un grand quel *id* il faut donner à la ligne (ici : 3).

#### Code : SQL

```
INSERT INTO Animal
VALUES (NULL, 'chat', 'F', '2010-09-13 15:02:00', 'Schtroumpfette',
NULL);
```

Vous avez maintenant trois animaux dans votre table :

Id	Espèce	Sexe	Date de naissance	Nom	Commentaires
1	chien	M	2010-04-05 13:43:00	Rox Mordille	beaucoup
2	chat	NULL	2010-03-24 02:23:00	Roucky	NULL
3	chat	F	2010-09-13 15:02:00	Schtroumpfette	NULL

Pour vérifier, vous pouvez utiliser la requête suivante :

#### Code : SQL

```
SELECT * FROM Animal;
```

Deux choses importantes à retenir ici.

## Partie 1 : MySQL et les bases du langage SQL

- *id* est un nombre, on ne met donc pas de guillemets autour. Par contre, l'espèce, le nom, la date de naissance et le sexe sont donnés sous forme de chaînes de caractères. Les guillemets sont donc indispensables. Quant à **NULL**, il s'agit d'un marqueur SQL qui, je rappelle, signifie "pas de valeur". Pas de guillemets donc.
- Les valeurs des colonnes sont données dans le bon ordre (donc dans l'ordre donné lors de la création de la table). C'est indispensable évidemment. Si vous échangez le nom et l'espèce par exemple, comment MySQL pourrait-il le savoir ?

### 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** lorsqu'on n'a pas de valeur à mettre.

Quelques exemples :

**Code : SQL**

```
INSERT INTO Animal (espece, sexe, date_naissance) VALUES ('tortue',  
'F', '2009-08-03 05:12:00');  
INSERT INTO Animal (nom, commentaires, date_naissance, espece)  
VALUES ('Choupi', 'Né sans oreille gauche', '2010-10-03  
16:44:00', 'chat');  
INSERT INTO Animal (espece, date_naissance, commentaires, nom, sexe)  
VALUES ('tortue', '2009-06-13 08:17:00', 'Carapace bizarre', 'Bobosse', 'F');
```

Ce qui vous donne trois animaux supplémentaires (donc six en tout, il faut suivre !)

### Insertion multiple

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

**Code : SQL**

```
INSERT INTO Animal (espece, sexe, date_naissance, nom)  
VALUES ('chien', 'F', '2008-12-06 05:18:00', 'Caroline'),  
( 'chat', 'M', '2008-09-11 15:38:00', 'Bagherra'),  
( 'tortue', NULL, '2010-08-23 05:18:00', NULL);
```

Bien entendu, vous êtes alors obligés de préciser les mêmes colonnes pour chaque entrée, quitte à mettre **NULL** pour certaines. Mais avouez que ça fait quand même moins à écrire !

### Syntaxe alternative de MySQL

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

**Code : SQL**

```
INSERT INTO Animal
SET nom='Bobo', espece='chien', 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 = 'Bobo') rend la syntaxe plus lisible et plus facile à manipuler. En effet, ici il n'y a que six colonnes, mais imaginez une table avec 20, voire 100 colonnes. Difficile d'être sûrs 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 (mais bon, une requête de plus ou de moins, ce n'est pas non plus énorme...)



Cependant, cette syntaxe alternative présente également des défauts, qui pour moi sont plus importants que les avantages apportés. C'est pourquoi je vous déconseille de l'utiliser. Je vous la montre surtout pour que vous ne soyez pas surpris si vous la rencontrez quelque part.

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

Maintenant que vous savez insérer des données, je vous propose de remplir un peu cette table, histoire qu'on puisse s'amuser par la suite.

Rassurez-vous, je ne vais pas vous demander d'inventer cinquante bestioles et d'écrire une à une les requêtes permettant de les insérer. Je vous ai prémâché le boulot. De plus, ça nous permettra d'avoir, vous et moi, la même chose dans notre base. Ce sera ainsi plus facile de vérifier que vos requêtes font bien ce qu'elles doivent faire.

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

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

Écrire toutes les commandes à la main dans la console, ça peut vite devenir pénible. Quand c'est une petite requête, pas de problème. Mais quand vous avez une longue requête, ou beaucoup de requêtes à faire, ça peut être assez long.

Une solution sympathique est d'écrire les requêtes dans un fichier texte, puis de dire à MySQL d'exécuter les requêtes contenues dans ce fichier. Et pour lui dire ça, c'est facile :

**Code : SQL**

```
SOURCE monFichier.sql;
```

Ou

**Code : SQL**

```
\. monFichier.sql;
```

Ces deux commandes sont équivalentes et vont exécuter le fichier monFichier.sql. Il n'est pas indispensable de lui donner l'extension .sql, mais je préfère le faire pour repérer mes fichiers SQL directement. De plus, si vous utilisez un éditeur de texte un peu plus évolué que le bloc-note (ou textEdit sur Mac), cela colorera votre code SQL, ce qui vous facilitera aussi les choses.

## Partie 1 : MySQL et les bases du langage SQL

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

**Exemple :** on donne le chemin complet vers le fichier

**Code : SQL**

```
SOURCE C:\Document and Settings\dossierX\monFichier.sql;
```

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

Par fichier formaté, j'entends 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 :

**Code : 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**, dont voici la syntaxe :

**Code : SQL**

```
LOAD DATA [LOCAL] INFILE 'nom_fichier'  
INTO TABLE nom_table  
[FIELDS  
  [TERMINATEDBY '\t']  
  [ENCLOSEDBY '']  
  [ESCAPEDBY '\\']  
]  
[LINES  
  [STARTINGBY '']  
  [TERMINATEDBY '\n']  
]  
[IGNORE nombre LINES]  
[(nom_colonne, ...)];
```

Le mot-clé **LOCAL** sert à spécifier si le fichier se trouve côté client (dans ce cas, on utilise **LOCAL**) ou côté serveur (auquel cas, on ne met pas **LOCAL** dans la commande). Si le fichier se trouve du côté 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, et qui contient les fichiers dans lesquels sont stockées les données de la base. Pour ma part, j'utiliserai toujours **LOCAL**, afin de pouvoir mettre simplement mes fichiers dans mon dossier de travail.

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. Les valeurs que j'ai mises ci-dessus sont les valeurs par défaut.

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,

## Partie 1 : MySQL et les bases du langage SQL

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.

Si je reprends mon exemple, en imaginant que nous ayons une table *Personne* contenant les colonnes *id* (clé primaire autoincrémentée), *nom*, *prenom*, *date\_naissance* et *adresse* (qui peut être **NULL**).

### Code : CSV

```
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* :

### Code : SQL

```
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 que je viens de vous montrer pour remplir un peu notre base. N'oubliez pas de modifier les commandes données pour ajouter le chemin vers vos fichiers,

## Exécution de commandes SQL

Voici donc le code que je vous demande de copier-coller dans votre éditeur de texte préféré, puis de le sauver sous le nom *remplissageAnimal.sql* (ou un autre nom de votre choix).

Secret (cliquez pour afficher)

Code : SQL

```
INSERT INTO Animal (espece, sexe, date_naissance, nom, commentaires) VALUES
('chien', 'F', '2008-02-20 15:45:00', 'Canaille', NULL),
('chien', 'F', '2009-05-26 08:54:00', 'Cali', NULL),
('chien', 'F', '2007-04-24 12:54:00', 'Rouquine', NULL),
('chien', 'F', '2009-05-26 08:56:00', 'Fila', NULL),
('chien', 'F', '2008-02-20 15:47:00', 'Any', NULL),
('chien', 'F', '2009-05-26 08:50:00', 'Louya', NULL),
('chien', 'F', '2008-03-10 13:45:00', 'Welva', NULL),
('chien', 'F', '2007-04-24 12:59:00', 'Zira', NULL),
('chien', 'F', '2009-05-26 09:02:00', 'Java', NULL),
('chien', 'M', '2007-04-24 12:45:00', 'Balou', NULL),
('chien', 'M', '2008-03-10 13:43:00', 'Pataud', NULL),
('chien', 'M', '2007-04-24 12:42:00', 'Bouli', NULL),
('chien', 'M', '2009-03-05 13:54:00', 'Zoulou', NULL),
('chien', 'M', '2007-04-12 05:23:00', 'Cartouche', NULL),
('chien', 'M', '2006-05-14 15:50:00', 'Zambo', NULL),
('chien', 'M', '2006-05-14 15:48:00', 'Samba', NULL),
('chien', 'M', '2008-03-10 13:40:00', 'Moka', NULL),
('chien', 'M', '2006-05-14 15:40:00', 'Pilou', NULL),
('chat', 'M', '2009-05-14 06:30:00', 'Fiero', NULL),
('chat', 'M', '2007-03-12 12:05:00', 'Zonko', NULL),
('chat', 'M', '2008-02-20 15:45:00', 'Filou', NULL),
('chat', 'M', '2007-03-12 12:07:00', 'Farceur', NULL),
('chat', 'M', '2006-05-19 16:17:00', 'Caribou', NULL),
('chat', 'M', '2008-04-20 03:22:00', 'Capou', NULL), ('chat', 'M', '2006-05-19 16:56:00',
'Raccou', 'Pas de queue depuis la naissance');
```

Vous n'avez alors qu'à taper :

Code : SQL

```
SOURCE remplissageAnimal.sql;
```

## LOAD DATA INFILE

À nouveau, copiez-collez le texte ci-dessous dans votre éditeur de texte, et enregistrez le fichier. Cette fois, sous le nom animal.csv.

Secret (cliquez pour afficher)

Code : CSV

```
"chat","M","2009-05-14 06:42:00","Boucan";NULL
"chat","F","2006-05-19 16:06:00","Callune";NULL
"chat","F","2009-05-14 06:45:00","Boule";NULL
"chat","F","2008-04-20 03:26:00","Zara";NULL
"chat","F","2007-03-12 12:00:00","Milla";NULL
"chat","F","2006-05-19 15:59:00","Feta";NULL
"chat","F","2008-04-
20 03:20:00","Bilba","Sourde de l'oreille droite à 80%"
"chat","F","2007-03-12 11:54:00","Cracotte";NULL
"chat","F","2006-05-19 16:16:00","Cawette";NULL
```



```
"tortue";"F";"2007-04-01 18:17:00";"Nikki";NULL
"tortue";"F";"2009-03-24 08:23:00";"Tortilla";NULL
"tortue";"F";"2009-03-26 01:24:00";"Scroupy";NULL
"tortue";"F";"2006-03-15 14:56:00";"Lulla";NULL
"tortue";"F";"2008-03-15 12:02:00";"Dana";NULL
"tortue";"F";"2009-05-25 19:57:00";"Cheli";NULL
"tortue";"F";"2007-04-01 03:54:00";"Chicaca";NULL
"tortue";"F";"2006-03-15 14:26:00";"Redbul";"Insomniaque"
"tortue";"M";"2007-04-02 01:45:00";"Spoutnik";NULL
"tortue";"M";"2008-03-16 08:20:00";"Bubulle";NULL
"tortue";"M";"2008-03-15 18:45:00";"Relou";"Surpoids"
"tortue";"M";"2009-05-25 18:54:00";"Bulbizard";NULL
"perroquet";"M";"2007-03-04 19:36:00";"Safran";NULL
"perroquet";"M";"2008-02-20 02:50:00";"Gingko";NULL
"perroquet";"M";"2009-03-26 08:28:00";"Bavard";NULL
"perroquet";"F";"2009-03-26 07:55:00";"Parlotte";NULL
```



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

Exécutez ensuite la commande suivante :

#### Code : SQL

```
LOAD DATA LOCAL INFILE 'animal.csv'
INTO TABLE Animal
FIELDS TERMINATED BY ';' ENCLOSED BY '"'
LINES TERMINATED BY '\n' -- ou '\r\n' selon l'ordinateur et le programme utilisés pour créer le
fichier
(espece, sexe, date_naissance, nom, commentaires);
```

Et hop ! Vous avez plus d'une cinquantaine d'animaux dans votre table.

Si vous voulez vérifier, je rappelle que vous pouvez utiliser la commande suivante, qui vous affichera toutes les données contenues dans la table *Animal*.

#### Code : SQL

```
SELECT * FROM Animal;
```

Nous pouvons maintenant passer au chapitre suivant !

### En résumé

- Pour insérer des lignes dans une table, on utilise la commande

#### Code : SQL

```
INSERT INTO nom_table [(colonne1, colonne2, ...)] VALUES
(valeur1, valeur2, ...);
```

- Si l'on ne précise pas à quelles colonnes on donne une valeur, il faut donner une valeur à toutes les colonnes, et dans le bon ordre.
- Il est possible d'insérer plusieurs lignes en une fois, en séparant les listes de valeurs par une virgule.
- Si l'on a un fichier texte contenant des requêtes SQL, on peut l'exécuter en utilisant **SOURCE** nom\_fichier; ou \. nom\_fichier;.
- La commande **LOAD DATA [LOCAL] INFILE** permet de charger des données dans une table à partir d'un fichier formaté (.csv par exemple).

## Sélection de données

Comme son nom l'indique, ce chapitre traitera de la sélection et de l'affichage de données.

Au menu :

- syntaxe de la requête **SELECT** (que vous avez déjà croisée il y a quelque temps) ; sélection
- de données répondant à certaines conditions ; tri des données ; élimination des données en
- double ; récupération de seulement une partie des données (uniquement les 10 premières
- lignes, par exemple).
- 

### Syntaxe de SELECT

La requête qui permet de sélectionner et afficher des données s'appelle **SELECT**. Nous l'avons déjà un peu utilisée dans le chapitre d'installation, ainsi que pour afficher tout le contenu de la table *Animal*.

**SELECT** permet donc d'afficher des données directement. Des chaînes de caractères, des résultats de calculs, etc.

Exemple

Code : SQL

```
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.

Code : SQL

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

Par exemple, si l'on veut sélectionner l'espèce, le nom et le sexe des animaux présents dans la table *Animal*, on utilisera :

Code : SQL

```
SELECT espece, nom, sexe  
FROM Animal;
```

### Sélectionner toutes les colonnes

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

Code : SQL

```
SELECT *  
FROM Animal;
```

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

## Partie 2 : Les fonctions et procédures stockées

Le désavantage est bien sûr que vous avez plus à écrire, mais le jeu en vaut la chandelle.

Comme vous avez pu le constater, les requêtes **SELECT** faites jusqu'à présent sélectionnent toutes les lignes de la table. Or, bien souvent, on ne veut qu'une partie des données. Dans la suite de ce chapitre, nous allons voir ce que nous pouvons ajouter à cette requête **SELECT** pour faire des sélections à l'aide de critères.

### La clause WHERE

La clause **WHERE** ("où" en anglais) permet de restreindre les résultats selon des critères de recherche. On peut par exemple vouloir ne sélectionner que les chiens :

Code : SQL

```
SELECT *  
FROM Animal  
WHERE espece='chien';
```



Comme 'chien' est une chaîne de caractères, je dois bien sûr l'entourer de guillemets.

### 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 <b>NULL</b> aussi)

Exemples :

Code : SQL

```
SELECT *  
FROM Animal  
WHERE date_naissance< '2008-01-01'; -- Animaux nés avant 2008  
  
SELECT *  
FROM Animal  
WHERE espece <> 'chat'; -- Tous les animaux sauf les chats
```

### Combinaisons de critères

Tout ça c'est bien beau, mais comment faire si on veut les chats et les chiens par exemple ? Faut-il faire deux requêtes ?

Non bien sûr, il suffit de combiner les critères. Pour cela, il faut des opérateurs logiques, qui sont au nombre de quatre :

## Partie 2 : Les fonctions et procédures stockées

Opérateur	Symbole	Signification
AND	&&	ET
OR		OU
XOR		OU exclusif
NOT	!	NON

Voici quelques exemples, sûrement plus efficaces qu'un long discours.

### AND

Je veux sélectionner toutes les chattes. Je veux donc sélectionner les animaux qui sont à la fois des chats ET des femelles. J'utilise l'opérateur **AND** :

Code : SQL

```
SELECT *
FROM Animal
WHERE espece='chat'
      AND sexe='F';
-- OU
SELECT *
FROM Animal
WHERE espece='chat'
      && sexe='F';
```

### OR

Sélection des tortues et des perroquets. Je désire donc obtenir les animaux qui sont des tortues OU des perroquets : Code : SQL

```
SELECT *
FROM Animal
WHERE espece='tortue'
      OR espece='perroquet';
-- OU
SELECT *
FROM Animal
WHERE espece='tortue'
      || espece='perroquet';
```



Je vous conseille 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 animaux femelles sauf les chiennes.

Code : SQL

## Partie 2 : Les fonctions et procédures stockées

```
SELECT *
FROM Animal
WHERE sexe='F'
      AND NOT espece='chien';
-- OU
SELECT *
FROM Animal
WHERE sexe='F'
      AND ! espece='chien';
```

### XOR

Sélection des animaux qui sont soit des mâles, soit des perroquets (mais pas les deux) :

#### Code : SQL

```
SELECT *
FROM Animal
WHERE sexe='M'
      XOR espece='perroquet';
```

Et voilà pour les opérateurs logiques. Rien de bien compliqué, et pourtant, c'est souvent source d'erreur. Pourquoi ? Tout simplement parce que tant que vous n'utilisez qu'un seul opérateur logique, tout va très bien. Mais on a souvent besoin de combiner plus de deux critères, et c'est là que ça se corse.

## 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. En effet, cela n'a pas de sens de mettre plusieurs opérateurs logiques différents sur un même niveau.

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 qu'on désire sélectionner !

### Exercice/Exemple

Alors, imaginons une requête bien tordue...

Je voudrais les animaux qui sont, soit nés après 2009, soit des chats mâles ou femelles, mais dans le cas des femelles, elles doivent être nées avant juin 2007.

Je vous conseille d'essayer d'écrire cette requête tout seuls. Si vous n'y arrivez pas, voici une petite aide : l'astuce, c'est de penser en niveaux. Je vais donc découper ma requête.

## **Partie 2 : Les fonctions et procédures stockées**

Je cherche :

- Les animaux nés après 2009 ;
- Les chats mâles et femelles (uniquement nées avant juin 2007 pour les femelles).

C'est mon premier niveau. L'opérateur logique sera **OR** puisqu'il faut que les animaux 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. Je cherche :

- Les animaux nés après 2009 ;
- Les chats : mâles ; et femelles nées avant juin 2007.

# Les fonctions et procédures stockées

## Triggers

Les triggers (ou déclencheurs) sont des objets de la base de données. **Attachés à une table**, ils vont **déclencher l'exécution d'une instruction**, ou d'un bloc d'instructions, **lorsqu'une, ou plusieurs lignes sont insérées, supprimées ou modifiées** dans la table à laquelle ils sont attachés.

Dans ce chapitre, nous allons voir comment ils fonctionnent exactement, comment on peut les créer et les supprimer, et surtout, comment on peut s'en servir et quelles sont leurs restrictions.

- Principe et usage
- Création des triggers
- Suppression des triggers
- Exemples
- Restrictions

### Principe et usage

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

Tout comme les procédures stockées, les triggers servent à exécuter une ou plusieurs instructions. Mais à la différence des procédures, il n'est pas possible d'appeler un trigger : un trigger doit être déclenché par un événement.

Un trigger est **attaché à une table**, et peut **être déclenché** par :

- une insertion dans la table (requête INSERT) ;
- la suppression d'une partie des données de la table (requête DELETE) ;
- la modification d'une partie des données de la table (requête UPDATE).

Par ailleurs, une fois le trigger déclenché, ses instructions peuvent être exécutées soit juste avant l'exécution de l'événement déclencheur, soit juste après.

#### Que fait un trigger ?

Un trigger exécute un **traitement pour chaque ligne insérée, modifiée ou supprimée** par l'événement déclencheur. Donc si l'on insère cinq lignes, les instructions du trigger seront exécutées cinq fois, chaque itération permettant de traiter les données d'une des lignes insérées.



Les instructions d'un trigger suivent les mêmes principes que les instructions d'une procédure stockée. S'il y a plus d'une instruction, il faut les mettre à l'intérieur d'un bloc d'instructions. Les structures que nous avons vues dans les deux chapitres précédents sont bien sûr utilisables (structures conditionnelles, boucles, gestionnaires d'erreur, etc.), avec toutefois quelques restrictions que nous verrons en fin de chapitre.

Un trigger peut modifier et/ou insérer des données dans n'importe quelle table sauf les tables utilisées dans la requête qui l'a déclenché. En ce qui concerne la table à laquelle le trigger est attaché (qui est forcément utilisée par l'événement déclencheur), le trigger peut lire et modifier uniquement la ligne insérée, modifiée ou supprimée qu'il est en train de traiter.

### À quoi sert un trigger ?

On peut faire de nombreuses choses avec un trigger. Voici quelques exemples d'usage fréquent de ces objets. Nous verrons plus loin certains de ces exemples appliqués à notre élevage d'animaux.

### Contraintes et vérifications de données

Comme cela a déjà été mentionné dans le chapitre sur les types de données, MySQL n'implémente pas de contraintes d'assertion, qui sont des contraintes permettant de limiter les valeurs acceptées par une colonne (limiter une colonne TINYINT à TRUE (1) ou FALSE (0) par exemple). Avec des triggers se déclenchant avant l'INSERT et avant l'UPDATE, on peut vérifier les valeurs d'une colonne lors de l'insertion ou de la modification, et les corriger si elles ne font pas partie des valeurs acceptables, ou bien faire échouer la requête. On peut ainsi pallier l'absence de contraintes d'assertion.

### Intégrité des données

Les triggers sont parfois utilisés pour remplacer les options des clés étrangères ON UPDATE RESTRICT|CASCADE|SET NULL et ON DELETE RESTRICT|CASCADE|SET NULL. Notamment pour des tables MyISAM, qui sont non-transactionnelles et ne supportent pas les clés étrangères. Cela peut aussi être utilisé avec des tables transactionnelles, dans les cas où le traitement à appliquer pour garder des données cohérentes est plus complexe que ce qui est permis par les options de clés étrangères.

Par exemple, dans certains systèmes, on veut pouvoir appliquer deux systèmes de suppression :

- une vraie suppression pure et dure, avec effacement des données, donc une requête DELETE ;
- un archivage, qui masquera les données dans l'application mais les conservera dans la base de données.

Dans ce cas, une solution possible est d'ajouter aux tables contenant des données archivables une colonne *archive*, pouvant contenir 0 (la ligne n'est pas archivée) ou 1 (la ligne est archivée). Pour une vraie suppression, on peut utiliser simplement un `ON DELETE RESTRICT|CASCADE|SET NULL`, qui se répercutera sur les tables référençant les données supprimées. Par contre, dans le cas d'un archivage, on utilisera plutôt un trigger pour traiter les lignes qui référencent les données archivées, par exemple en les archivant également.

### Historisation des actions

On veut parfois garder une trace des actions effectuées sur la base de données, c'est-à-dire par exemple, savoir qui a modifié telle ligne, et quand. Avec les triggers, rien de plus simple, il suffit de mettre à jour des données d'historisation à chaque insertion, modification ou suppression. Soit directement dans la table concernée, soit dans une table utilisée spécialement et exclusivement pour garder un historique des actions.

### Mise à jour d'informations qui dépendent d'autres données

Comme pour les procédures stockées, une partie de la logique "business" de l'application peut être codée directement dans la base de données, grâce aux triggers, plutôt que du côté applicatif (en PHP, Java ou quel que soit le langage de programmation utilisé). À nouveau, cela peut permettre d'harmoniser un traitement à travers plusieurs applications utilisant la même base de données.

Par ailleurs, lorsque certaines informations dépendent de la valeur de certaines données, on peut en général les retrouver en faisant une requête `SELECT`. Dans ce cas, il n'est pas indispensable de stocker ces informations. Cependant, utiliser les triggers pour stocker ces informations peut faciliter la vie de l'utilisateur, et peut aussi faire gagner en performance. Par exemple, si l'on a très souvent besoin de cette information, ou si la requête à faire pour trouver cette information est longue à exécuter. C'est typiquement cet usage qui est fait des triggers dans ce qu'on appelle les "vues matérialisées", auxquelles un chapitre est consacré dans la partie 6.

Création des triggers

### Syntaxe

Pour créer un trigger, on utilise la commande suivante :

```
1 CREATE TRIGGER nom_trigger moment_trigger evenement_trigger
2 ON nom_table FOR EACH ROW
3 corps_trigger
```

## Partie 2 : Les fonctions et procédures stockées

- **CREATE TRIGGER nom\_trigger** : les triggers ont donc un nom.
- **moment\_trigger evenement\_trigger** : servent à définir quand et comment le trigger est déclenché.
- **ON nom\_table** : c'est là qu'on définit à quelle table le trigger est attaché.
- **FOR EACH ROW** : signifie littéralement "pour chaque ligne", sous-entendu "pour chaque ligne insérée/supprimée/modifiée" selon ce qui a déclenché le trigger.
- **corps\_trigger** : c'est le contenu du trigger. Comme pour les procédures stockées, il peut s'agir soit d'une seule instruction, soit d'un bloc d'instructions.

### Événement déclencheur

Trois événements différents peuvent déclencher l'exécution des instructions d'un trigger.

- L'insertion de lignes (**INSERT**) dans la table attachée au trigger.
- La modification de lignes (**UPDATE**) de cette table.
- La suppression de lignes (**DELETE**) de la table.

Un trigger est soit déclenché par **INSERT**, soit par **UPDATE**, soit par **DELETE**. Il ne peut pas être déclenché par deux événements différents. On peut par contre créer plusieurs triggers par table pour couvrir chaque événement.

### Avant ou après

Lorsqu'un trigger est déclenché, ses instructions peuvent être exécutées à deux moments différents. Soit juste avant que l'événement déclencheur n'ait lieu (**BEFORE**), soit juste après (**AFTER**).

Donc, si vous avez un trigger **BEFORE UPDATE** sur la table *A*, l'exécution d'une requête **UPDATE** sur cette table va d'abord déclencher l'exécution des instructions du trigger, ensuite seulement les lignes de la table seront modifiées.

### Exemple

Pour créer un trigger sur la table *Animal*, déclenché par une insertion, et s'exécutant après ladite insertion, on utilisera la syntaxe suivante :

```
1  CREATE TRIGGER after_insert_animal AFTER INSERT
2  ON Animal FOR EACH ROW
3  corps_trigger;
```

### Règle et convention

Il ne peut exister qu'un seul trigger par combinaison moment\_trigger/événement\_trigger par table. Donc un seul trigger BEFORE UPDATE par table, un seul AFTER DELETE, etc. Étant donné qu'il existe deux possibilités pour le moment d'exécution, et trois pour l'événement déclencheur, on a donc un **maximum de six triggers par table**.

Cette règle étant établie, il existe une convention quant à la manière de nommer ses triggers, que je vous encourage à suivre : *nom\_trigger = moment\_evenement\_table*. Donc le trigger BEFORE UPDATE ON Animal aura pour nom : *before\_update\_animal*.

### OLD et NEW

Dans le corps du trigger, MySQL met à disposition deux mots-clés : OLD et NEW.

- OLD : représente les valeurs des colonnes de la ligne traitée **avant qu'elle ne soit modifiée** par l'événement déclencheur. Ces valeurs peuvent être **lues, mais pas modifiées**.
- NEW : représente les valeurs des colonnes de la ligne traitée **après qu'elle a été modifiée** par l'événement déclencheur. Ces valeurs peuvent être **lues et modifiées**.

Il n'y a que dans le cas d'un trigger UPDATE que OLD et NEW coexistent. Lors d'une insertion, OLD n'existe pas, puisque la ligne n'existe pas avant l'événement déclencheur ; dans le cas d'une suppression, c'est NEW qui n'existe pas, puisque la ligne n'existera plus après l'événement déclencheur.

**Premier exemple** : l'insertion d'une ligne.

Exécutons la commande suivante :

```
1  INSERT INTO Adoption (client_id, animal_id, date_reservation, prix, paye)
2  VALUES (12, 15, NOW(), 200.00, FALSE);
```

Pendant le traitement de cette ligne par le trigger correspondant,

- NEW.client\_id vaudra 12 ;
- NEW.animal\_id vaudra 15 ;
- NEW.date\_reservation vaudra NOW() ;
- NEW.date\_adoption vaudra NULL ;
- NEW.prix vaudra 200.00 ;
- NEW.payé vaudra FALSE (0).

## Partie 2 : Les fonctions et procédures stockées

Les valeurs de OLD ne seront pas définies. Dans le cas d'une suppression, on aura exactement l'inverse.

**Second exemple** : la modification d'une ligne. On modifie la ligne que l'on vient d'insérer en exécutant la commande suivante :

```
1  UPDATE Adoption
2  SET paye = TRUE
3  WHERE client_id = 12 AND animal_id = 15;
```

Pendant le traitement de cette ligne par le trigger correspondant,

- NEW.paye vaudra TRUE, tandis que OLD.paye vaudra FALSE.
- Par contre les valeurs respectives de NEW.animal\_id, NEW.client\_id, NEW.date\_reservation, NEW.date\_adoption et NEW.prix seront les mêmes que OLD.animal\_id, OLD.client\_id, OLD.date\_reservation, OLD.date\_adoption et OLD.prix, puisque ces colonnes ne sont pas modifiées par la requête.

Dans le cas d'une insertion ou d'une modification, si un trigger peut potentiellement changer la valeur de NEW.colonne, il doit être exécuté avant l'événement (BEFORE). Sinon, la ligne aura déjà été insérée ou modifiée, et la modification de NEW.colonne n'aura plus aucune influence sur celle-ci.

### Erreur déclenchée pendant un trigger

- Si un trigger BEFORE génère une erreur (non interceptée par un gestionnaire d'erreur), la requête ayant déclenché le trigger ne sera pas exécutée. Si l'événement devait également déclencher un trigger AFTER, il ne sera bien sûr pas non plus exécuté.
- Si un trigger AFTER génère une erreur, la requête ayant déclenché le trigger échouera.
- Dans le cas d'une table transactionnelle, si une erreur est déclenchée, un ROLLBACK sera fait. Dans le cas d'une table non-transactionnelle, tous les changements qui auraient été faits par le (ou les) trigger(s) avant le déclenchement de l'erreur persisteront.

### Suppression des triggers

Encore une fois, la commande DROP permet de supprimer un trigger.

```
1  DROP TRIGGER nom_trigger;
```

## Partie 2 : Les fonctions et procédures stockées

Tout comme pour les procédures stockées, il n'est pas possible de modifier un trigger. Il faut le supprimer puis le recréer différemment.

Par ailleurs, si l'on supprime une table, on supprime également tous les triggers qui y sont attachés.

### Exemples

#### Contraintes et vérification des données

##### Vérification du sexe des animaux

Dans notre table *Animal* se trouve la colonne *sexe*. Cette colonne accepte tout caractère, ou NULL. Or, seuls les caractères "M" et "F" ont du sens. Nous allons donc créer deux triggers, un pour l'insertion, l'autre pour la modification, qui vont empêcher qu'on donne un autre caractère que "M" ou "F" pour *sexe*.

Ces deux triggers devront se déclencher avant l'insertion et la modification. On aura donc :

```
1  -- Trigger déclenché par l'insertion
2  DELIMITER |
3  CREATE TRIGGER before_insert_animal BEFORE INSERT
4  ON Animal FOR EACH ROW
5  BEGIN
6      -- Instructions
7  END |
8
9  -- Trigger déclenché par la modification
10 CREATE TRIGGER before_update_animal BEFORE UPDATE
11 ON Animal FOR EACH ROW
12 BEGIN
13     -- Instructions
14 END |
15 DELIMITER ;
```

## Partie 2 : Les fonctions et procédures stockées

Il ne reste plus qu'à écrire le code du trigger, qui sera similaire pour les deux triggers. Et comme ce corps contiendra des instructions, il ne faut pas oublier de changer le délimiteur.

Le corps consistera en une simple structure conditionnelle, et définira un comportement à adopter si le sexe donné ne vaut ni "M", ni "F", ni NULL.

Quel comportement adopter en cas de valeur erronée ?

Deux possibilités :

- on modifie la valeur du sexe, en le mettant à NULL par exemple ;
- on provoque une erreur, ce qui empêchera l'insertion/la modification.

Commençons par le plus simple : mettre le sexe à NULL.

```
1  DELIMITER |
2  CREATE TRIGGER before_update_animal BEFORE UPDATE
3  ON Animal FOR EACH ROW
4  BEGIN
5      IF NEW.sexe IS NOT NULL -- le sexe n'est ni NULL
6      AND NEW.sexe != 'M'    -- ni "M"
7      AND NEW.sexe != 'F'    -- ni "F"
8      THEN
9          SET NEW.sexe = NULL;
10     END IF;
11 END |
12 DELIMITER ;
```

Test :

```
1  UPDATE Animal
2  SET sexe = 'A'
3  WHERE id = 20; -- l'animal 20 est Balou, un mâle
4
```



```
5  SELECT id, sexe, date_naissance, nom
6  FROM Animal
7  WHERE id = 20;
```

id	sexe	date_naissance	nom
20	NULL	2007-04-24 12:45:00	Balou

Le sexe est bien NULL, le trigger a fonctionné.

Pour le second trigger, déclenché par l'insertion de lignes, on va implémenter le second comportement : on va déclencher une erreur, ce qui empêchera l'insertion, et affichera l'erreur.

Mais comment déclencher une erreur ?

Contrairement à certains SGBD, MySQL ne dispose pas d'une commande permettant de déclencher une erreur personnalisée. La seule solution est donc de faire une requête dont on sait qu'elle va générer une erreur.

**Exemple :**

```
1  SELECT 1, 2 INTO @a;

1  ERROR 1222 (21000): The used SELECT statements have a different n
    umber of columns
```

Cependant, il serait quand même intéressant d'avoir un message d'erreur qui soit un peu explicite. Voici une manière d'obtenir un tel message : on crée une table *Erreur*, ayant deux colonnes, *id* et *erreur*. La colonne *id* est clé primaire, et *erreur* contient un texte court décrivant l'erreur. Un index UNIQUE est ajouté sur cette dernière colonne. On insère ensuite une ligne correspondant à l'erreur qu'on veut utiliser dans le trigger. Ensuite dans le corps du trigger, en cas de valeur erronée, on refait la même insertion. Cela déclenche une erreur de contrainte d'unicité, laquelle affiche le texte qu'on a essayé d'insérer dans *Erreur*.

```
1  -- Création de la table Erreur
2  CREATE TABLE Erreur (
3      id TINYINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
4      erreur VARCHAR(255) UNIQUE);
5
6  -- Insertion de l'erreur qui nous intéresse
7  INSERT INTO Erreur (erreur) VALUES ('Erreur : sexe doit valoir "M",
8  "F" ou NULL.');
```

```
9
10 -- Création du trigger
11 DELIMITER |
12 CREATE TRIGGER before_insert_animal BEFORE INSERT
13 ON Animal FOR EACH ROW
14 BEGIN
15     IF NEW.sexe IS NOT NULL -- le sexe n'est ni NULL
16     AND NEW.sexe != 'M'      -- ni "M"
17     AND NEW.sexe != 'F'      -- ni "F"
18     THEN
19         INSERT INTO Erreur (erreur) VALUES ('Erreur : sexe doit valoir
20 "M", "F" ou NULL.');
```

```
21     END IF;
22 END |
23 DELIMITER ;
```

Test :

```
1  INSERT INTO Animal (nom, sexe, date_naissance, espece_id)
2  VALUES ('Babar', 'A', '2011-08-04 12:34', 3);
```

```
1 ERROR 1062 (23000): Duplicate entry 'Erreur : sexe doit valoir "M", "F" ou NULL.' for key 'erreur'
```

Et voilà, ce n'est pas parfait, mais au moins le message d'erreur permet de cerner d'où vient le problème. Et Babar n'a pas été inséré.

### Vérification du booléen dans *Adoption*

Il est important de savoir si un client a payé ou non pour les animaux qu'il veut adopter. Il faut donc vérifier la valeur de ce qu'on insère dans la colonne *paye*, et refuser toute insertion/modification donnant une valeur différente de TRUE (1) ou FALSE (0). Les deux triggers à créer sont très similaires à ce que l'on a fait pour la colonne *sexe* d'*Animal*. Essayez donc de construire les requêtes vous-mêmes.

```
1 INSERT INTO Erreur (erreur) VALUES ('Erreur : paye doit valoir TRUE (1)
2 ou FALSE (0).');
3
4 DELIMITER |
5 CREATE TRIGGER before_insert_adoption BEFORE INSERT
6 ON Adoption FOR EACH ROW
7 BEGIN
8     IF NEW.paye != TRUE    -- ni TRUE
9     AND NEW.paye != FALSE -- ni FALSE
10    THEN
11        INSERT INTO Erreur (erreur) VALUES ('Erreur : paye doit valoir TRUE
12 (1) ou FALSE (0).');
13    END IF;
14 END |
15
16 CREATE TRIGGER before_update_adoption BEFORE UPDATE
```

```
16  ON Adoption FOR EACH ROW
17  BEGIN
18      IF NEW.paye != TRUE    -- ni TRUE
19      AND NEW.paye != FALSE  -- ni FALSE
20      THEN
21          INSERT INTO Erreur (erreur) VALUES ('Erreur : paye doit valoir TRUE
22      (1) ou FALSE (0).');
23      END IF;
    END |
    DELIMITER ;
```

Test :

```
1  UPDATE Adoption
2  SET paye = 3
3  WHERE client_id = 9;
```

```
1  ERROR 1062 (23000): Duplicate entry 'Erreur : paye doit valoir TRUE (1) ou FALS
    E (0)' for key 'erreur'
```

### Vérification de la date d'adoption

Il reste une petite chose à vérifier, et ce sera tout pour les vérifications de données : la date d'adoption ! En effet, celle-ci doit être postérieure ou égale à la date de réservation. Un client ne peut pas emporter chez lui un animal avant même d'avoir prévenu qu'il voulait l'adopter. À nouveau, essayez de faire le trigger vous-mêmes. Pour rappel, il ne peut exister qu'un seul trigger BEFORE UPDATE et un seul BEFORE INSERT pour chaque table.

```
1  INSERT INTO Erreur (erreur) VALUES ('Erreur : date_adoption doit
2  être >= à date_reservation.');
```

3

```
4  DELIMITER |
5  DROP TRIGGER before_insert_adoption|
6  CREATE TRIGGER before_insert_adoption BEFORE INSERT
7  ON Adoption FOR EACH ROW
8  BEGIN
9      IF NEW.payé != TRUE                -- On remet la vérification
sur payé
10     AND NEW.payé != FALSE
11     THEN
12         INSERT INTO Erreur (erreur) VALUES ('Erreur : payé doit valoir
13         TRUE (1) ou FALSE (0).');
```

14

```
15     ELSEIF NEW.date_adoption < NEW.date_reservation THEN    --
16     Adoption avant réservation
17         INSERT INTO Erreur (erreur) VALUES ('Erreur : date_adoption
18         doit être >= à date_reservation.');
```

19

```
20     END IF;
21 END |
22
23 DROP TRIGGER before_update_adoption|
24 CREATE TRIGGER before_update_adoption BEFORE UPDATE
25 ON Adoption FOR EACH ROW
26 BEGIN
27     IF NEW.payé != TRUE                -- On remet la vérification
sur payé
28     AND NEW.payé != FALSE
```

```
28      THEN
29          INSERT INTO Erreur (erreur) VALUES ('Erreur : paye doit valoir
30      TRUE (1) ou FALSE (0).');
31
      ELSEIF NEW.date_adoption < NEW.date_reservation THEN    --
      Adoption avant réservation
          INSERT INTO Erreur (erreur) VALUES ('Erreur : date_adoption
      doit être >= à date_reservation.');
```

END IF;

END |

DELIMITER ;

On aurait pu faire un second IF au lieu d'un ELSEIF, mais de toute façon, le trigger ne pourra déclencher qu'une erreur à la fois.

**Test :**

```
1  INSERT INTO Adoption (animal_id, client_id, date_reservation,
2  date_adoption, prix, paye)
3  VALUES (10, 10, NOW(), NOW() - INTERVAL 2 DAY, 200.00, 0);
4
5  INSERT INTO Adoption (animal_id, client_id, date_reservation,
6  date_adoption, prix, paye)
7  VALUES (10, 10, NOW(), NOW(), 200.00, 4);
```

```
1  ERROR 1062 (23000): Duplicate entry 'Erreur : date_adoption doit être >=
2  à date_reservation.' for key 'erreur'
3
4  ERROR 1062 (23000): Duplicate entry 'Erreur : paye doit valoir TRUE (1) ou
5  FALSE (0).' for key 'erreur'
```

Les deux vérifications fonctionnent !

### Mise à jour d'informations dépendant d'autres données

Pour l'instant, lorsque l'on a besoin de savoir quels animaux restent disponibles pour l'adoption, il faut faire une requête avec sous-requête.

```
1  SELECT id, nom, sexe, date_naissance, commentaires
2  FROM Animal
3  WHERE NOT EXISTS (
4      SELECT *
5      FROM Adoption
6      WHERE Animal.id = Adoption.animal_id
7  );
```

Mais une telle requête n'est pas particulièrement performante, et elle est relativement peu facile à lire. Les triggers peuvent nous permettre de stocker automatiquement une donnée permettant de savoir immédiatement si un animal est disponible ou non.

Pour cela, il suffit d'ajouter une colonne *disponible* à la table *Animal*, qui vaudra FALSE ou TRUE, et qui sera mise à jour grâce à trois triggers sur la table *Adoption*.

- À l'insertion d'une nouvelle adoption, il faut retirer l'animal adopté des animaux disponibles ;
- en cas de suppression, il faut faire le contraire ;
- en cas de modification d'une adoption, si l'animal adopté change, il faut remettre l'ancien parmi les animaux disponibles et retirer le nouveau.

```
1  -- Ajout de la colonne disponible
2  ALTER TABLE Animal ADD COLUMN disponible BOOLEAN DEFAULT TRUE;
3  -- À l'insertion, un animal est forcément disponible
4
```

```
5  -- Remplissage de la colonne
6  UPDATE Animal
7  SET disponible = FALSE
8  WHERE EXISTS (
9      SELECT *
10     FROM Adoption
11     WHERE Animal.id = Adoption.animal_id
12 );
13
14 -- Création des trois triggers
15 DELIMITER |
16 CREATE TRIGGER after_insert_adoption AFTER INSERT
17 ON Adoption FOR EACH ROW
18 BEGIN
19     UPDATE Animal
20     SET disponible = FALSE
21     WHERE id = NEW.animal_id;
22 END |
23
24 CREATE TRIGGER after_delete_adoption AFTER DELETE
25 ON Adoption FOR EACH ROW
26 BEGIN
27     UPDATE Animal
28     SET disponible = TRUE
29     WHERE id = OLD.animal_id;
30 END |
31
```



```
32 CREATE TRIGGER after_update_adoption AFTER UPDATE
33 ON Adoption FOR EACH ROW
34 BEGIN
35     IF OLD.animal_id <> NEW.animal_id THEN
36         UPDATE Animal
37         SET disponible = TRUE
38         WHERE id = OLD.animal_id;
39
40         UPDATE Animal
41         SET disponible = FALSE
42         WHERE id = NEW.animal_id;
43     END IF;
44 END |
DELIMITER ;
```

Test :

```
1 SELECT animal_id, nom, sexe, disponible, client_id
2 FROM Animal
3 INNER JOIN Adoption ON Adoption.animal_id = Animal.id
4 WHERE client_id = 9;
```

animal_id	nom	sexe	disponible	client_id
33	Caribou	M	0	9
54	Bubulle	M	0	9
55	Relou	M	0	9

animal_id	nom	sexe	disponible	client_id
1	DELETE FROM Adoption			-- 54
2	doit redevenir disponible			
3	WHERE animal_id = 54;			
4				
5	UPDATE Adoption			
6	SET animal_id = 38, prix = 985.00			--
7	38 doit devenir indisponible			
8	WHERE animal_id = 33;			-- et
9	33 redevenir disponible			
10				
11	INSERT INTO Adoption (client_id, animal_id,			
12	date_reservation, prix, paye)			
13	VALUES (9, 59, NOW(), 700.00, FALSE);			-
14	- 59 doit devenir indisponible			
	SELECT Animal.id AS animal_id, nom, sexe, disponible,			
	client_id			
	FROM Animal			
	LEFT JOIN Adoption ON Animal.id = Adoption.animal_id			
	WHERE Animal.id IN (33, 54, 55, 38, 59);			

animal_id	nom	sexe	disponible	client_id
33	Caribou	M	1	NULL
38	Boule	F	0	9
54	Bubulle	M	1	NULL
55	Relou	M	0	9
59	Bavard	M	0	9

## Partie 2 : Les fonctions et procédures stockées

Désormais, pour savoir quels animaux sont disponibles, il suffira de faire la requête suivante :

```
1  SELECT *
2  FROM Animal
3  WHERE disponible = TRUE;
4
5  -- Ou même
6
7  SELECT *
8  FROM Animal
9  WHERE disponible;
```

### Historisation

Voici deux exemples de systèmes d'historisation :

- l'un très basique, gardant simplement trace de l'insertion (date et utilisateur) et de la dernière modification (date et utilisateur), et se faisant directement dans la table concernée ;
- l'autre plus complet, qui garde une copie de chaque version antérieure des lignes dans une table dédiée, ainsi qu'une copie de la dernière version en cas de suppression.

### Historisation basique

On va utiliser cette historisation pour la table *Race*. Libre à vous d'adapter ou de créer les triggers d'autres tables pour les historiser également de cette manière.

On ajoute donc quatre colonnes à la table. Ces colonnes seront toujours remplies automatiquement par les triggers.

```
1  -- On modifie la table Race
2  ALTER TABLE Race ADD COLUMN date_insertion DATETIME,      --
3  date d'insertion
4
```

```
5      ADD COLUMN utilisateur_insertion VARCHAR(20),      --
6      utilisateur ayant inséré la ligne
7      ADD COLUMN date_modification DATETIME,      -- date de
8      dernière modification
9      ADD COLUMN utilisateur_modification VARCHAR(20);      --
10     utilisateur ayant fait la dernière modification
11
12     -- On remplit les colonnes
13
14     UPDATE Race
15     SET date_insertion = NOW() - INTERVAL 1 DAY,
16         utilisateur_insertion = 'Test',
17         date_modification = NOW() - INTERVAL 1 DAY,
18         utilisateur_modification = 'Test';
```

J'ai mis artificiellement les dates d'insertion et de dernière modification à la veille d'aujourd'hui, et les utilisateurs pour l'insertion et la modification à "Test", afin d'avoir des données intéressantes lors des tests. Idéalement, ce type d'historisation doit bien sûr être mis en place dès la création de la table.

Occupons-nous maintenant des triggers. Il en faut sur l'insertion et sur la modification.

```
1  DELIMITER |
2  CREATE TRIGGER before_insert_race BEFORE INSERT
3  ON Race FOR EACH ROW
4  BEGIN
5      SET NEW.date_insertion = NOW();
6      SET NEW.utilisateur_insertion = CURRENT_USER();
7      SET NEW.date_modification = NOW();
8      SET NEW.utilisateur_modification = CURRENT_USER();
9  END |
```

```
10
11 CREATE TRIGGER before_update_race BEFORE UPDATE
12 ON Race FOR EACH ROW
13 BEGIN
14     SET NEW.date_modification = NOW();
15     SET NEW.utilisateur_modification = CURRENT_USER();
16 END |
17 DELIMITER ;
```

Les triggers sont très simples : ils mettent simplement à jour les colonnes d'historisation nécessaires ; ils doivent donc nécessairement être BEFORE.

**Test :**

```
1 INSERT INTO Race (nom, description, espece_id, prix)
2 VALUES ('Yorkshire terrier', 'Chien de petite taille au pelage long et
3 soyeux de couleur bleu et feu.', 1, 700.00);
4
5 UPDATE Race
6 SET prix = 630.00
7 WHERE nom = 'Rottweiler' AND espece_id = 1;
8
9 SELECT     nom,      DATE(date_insertion)    AS    date_ins,
10 utilisateur_insertion AS utilisateur_ins, DATE(date_modification) AS
date_mod, utilisateur_modification AS utilisateur_mod
FROM Race
WHERE espece_id = 1;
```

nom	date_ins	utilisateur_ins	date_mod	utilisateur_mod
Berger allemand	2012-05-02	Test	2012-05-02	Test
Berger blanc suisse	2012-05-02	Test	2012-05-02	Test
Rottweiller	2012-05-02	Test	2012-05-03	sdz@localhost
Yorkshire terrier	2012-05-03	sdz@localhost	2012-05-03	sdz@localhost

### Historisation complète

Nous allons mettre en place un système d'historisation complet pour la table *Animal*. Celle-ci ne change pas et contiendra la dernière version des données. Par contre, on va ajouter une table *Animal\_histo*, qui contiendra les versions antérieures (quand il y en a) des données d'*Animal*.

```
1  CREATE TABLE Animal_histo (  
2      id SMALLINT(6) UNSIGNED NOT NULL,          -- Colonnes historisées  
3      sexe CHAR(1),  
4      date_naissance DATETIME NOT NULL,  
5      nom VARCHAR(30),  
6      commentaires TEXT,  
7      espece_id SMALLINT(6) UNSIGNED NOT NULL,  
8      race_id SMALLINT(6) UNSIGNED DEFAULT NULL,  
9      mere_id SMALLINT(6) UNSIGNED DEFAULT NULL,  
10     pere_id SMALLINT(6) UNSIGNED DEFAULT NULL,  
11     disponible BOOLEAN DEFAULT TRUE,  
12  
13     date_histo DATETIME NOT NULL,                -- Colonnes techniques  
14     utilisateur_histo VARCHAR(20) NOT NULL,  
15     evenement_histo CHAR(6) NOT NULL,  
16     PRIMARY KEY (id, date_histo)  
17 ) ENGINE=InnoDB;
```

## Partie 2 : Les fonctions et procédures stockées

Les colonnes *date\_histo* et *utilisateur\_histo* contiendront bien sûr la date à laquelle la ligne a été historisée, et l'utilisateur qui a provoqué cette historisation. Quant à la colonne *evenement\_histo*, elle contiendra l'événement qui a déclenché le trigger (soit "DELETE", soit "UPDATE"). La clé primaire de cette table est le couple (*id*, *date\_histo*).

Voici les triggers nécessaires. Cette fois, ils pourraient être soit BEFORE, soit AFTER. Cependant, aucun traitement ne concerne les nouvelles valeurs de la ligne modifiée (ni, a fortiori, de la ligne supprimée). Par conséquent, autant utiliser AFTER, cela évitera d'exécuter les instructions du trigger en cas d'erreur lors de la requête déclenchant celui-ci.

```
1  DELIMITER |
2  CREATE TRIGGER after_update_animal AFTER UPDATE
3  ON Animal FOR EACH ROW
4  BEGIN
5      INSERT INTO Animal_histo (
6          id,
7          sexe,
8          date_naissance,
9          nom,
10         commentaires,
11         espece_id,
12         race_id,
13         mere_id,
14         pere_id,
15         disponible,
16
17         date_histo,
18         utilisateur_histo,
19         evenement_histo)
20     VALUES (
21         OLD.id,
```

```
22      OLD.sexe,  
23      OLD.date_naissance,  
24      OLD.nom,  
25      OLD.commentaires,  
26      OLD.espece_id,  
27      OLD.race_id,  
28      OLD.mere_id,  
29      OLD.pere_id,  
30      OLD.disponible,  
31  
32      NOW(),  
33      CURRENT_USER(),  
34      'UPDATE');  
35 END |  
36  
37 CREATE TRIGGER after_delete_animal AFTER DELETE  
38 ON Animal FOR EACH ROW  
39 BEGIN  
40     INSERT INTO Animal_histo (  
41         id,  
42         sexe,  
43         date_naissance,  
44         nom,  
45         commentaires,  
46         espece_id,  
47         race_id,  
48         mere_id,
```



```
49     pere_id,  
50     disponible,  
51  
52     date_histo,  
53     utilisateur_histo,  
54     evenement_histo)  
55     VALUES (  
56         OLD.id,  
57         OLD.sexe,  
58         OLD.date_naissance,  
59         OLD.nom,  
60         OLD.commentaires,  
61         OLD.espece_id,  
62         OLD.race_id,  
63         OLD.mere_id,  
64         OLD.pere_id,  
65         OLD.disponible,  
66  
67         NOW(),  
68         CURRENT_USER(),  
69         'DELETE');  
70     END |  
71     DELIMITER ;
```

Cette fois, ce sont les valeurs avant modification/suppression qui nous intéressent, d'où l'utilisation de OLD.

**Test :**

```
1  UPDATE Animal
2  SET commentaires = 'Petit pour son âge'
3  WHERE id = 10;
4
5  DELETE FROM Animal
6  WHERE id = 47;
7
8  SELECT id, sexe, date_naissance, nom, commentaires, espece_id
9  FROM Animal
10 WHERE id IN (10, 47);
11
12 SELECT id, nom, date_histo, utilisateur_histo, evenement_histo
13 FROM Animal_histo;
```

id	sexe	date_naissance	nom	commentaires	espece_id
10	M	2010-07-21 15:41:00	Bobo	Petit pour son âge	1
id	nom	date_histo	utilisateur_histo	evenement_histo	
10	Bobo	2012-05-03 21:51:12	sdz@localhost	UPDATE	
47	Scroupy	2012-05-03 21:51:12	sdz@localhost	DELETE	

### Quelques remarques sur l'historisation

Les deux systèmes d'historisation montrés dans ce cours ne sont que deux possibilités parmi des dizaines. Si vous pensez avoir besoin d'un système de ce type, prenez le temps de réfléchir, et de vous renseigner sur les diverses possibilités qui s'offrent à vous. Dans certains systèmes, on combine les deux historisations que j'ai présentées. Parfois, on ne conserve pas les lignes supprimées dans la table d'historisation, mais on utilise plutôt un système d'archive, séparé de l'historisation. Au-delà du modèle d'historisation que vous choisirez, les détails sont également modifiables. Voulez-vous garder toutes les versions des données, ou les garder seulement pour une certaine période de temps ? Voulez-vous enregistrer l'utilisateur SQL ou plutôt des utilisateurs créés pour votre application, découplés des

utilisateurs SQL ? Ne restez pas bloqués sur les exemples montrés dans ce cours (que ce soit pour l'historisation ou le reste), le monde est vaste !

### Restrictions

Les restrictions sur les triggers sont malheureusement trop importantes pour qu'on puisse se permettre de ne pas les mentionner. On peut espérer qu'une partie de ces restrictions soit levée dans une prochaine version de MySQL, mais en attendant, il est nécessaire d'avoir celles-ci en tête. Voici donc les principales.

#### Commandes interdites

**Il est impossible de travailler avec des transactions à l'intérieur d'un trigger.** Cette restriction est nécessaire, puisque la requête ayant provoqué l'exécution du trigger pourrait très bien se trouver elle-même à l'intérieur d'une transaction. Auquel cas, toute commande START TRANSACTION, COMMIT ou ROLLBACK interagirait avec cette transaction, de manière intempestive.

**Les requêtes préparées ne peuvent pas non plus être utilisées.**

Enfin, on ne peut pas appeler n'importe quelle procédure à partir d'un trigger.

- Les procédures appelées par un trigger **ne peuvent pas envoyer d'informations au client MySQL**. Par exemple, elles ne peuvent pas exécuter un simple SELECT, qui produit un affichage dans le client (un SELECT...INTO par contre est permis). Elles peuvent toutefois renvoyer des informations au trigger grâce à des paramètres OUT ou INOUT.
- Les procédures appelées ne peuvent utiliser ni les transactions (START TRANSACTION, COMMIT ou ROLLBACK) ni les requêtes préparées. C'est-à-dire qu'**elles doivent respecter les restrictions des triggers**.

#### Tables utilisées par la requête

Comme mentionné auparavant, il est **impossible de modifier les données d'une table utilisée par la requête ayant déclenché le trigger** à l'intérieur de celui-ci.

Cette restriction est importante, et peut remettre en question l'utilisation de certains triggers.

**Exemple** : le trigger AFTER INSERT ON Adoption modifie les données de la table *Animal*. Si l'on exécute la requête suivante, cela posera problème.

```
1  INSERT INTO Adoption (animal_id, client_id, date_reservation, prix, paye)
2  SELECT Animal.id, 4, NOW(), COALESCE(Race.prix, Espece.prix), FALSE
3  FROM Animal
```

```
4  INNER JOIN Espece ON Espece.id = Animal.espece_id
5  LEFT JOIN Race ON Race.id = Animal.race_id
6  WHERE Animal.nom = 'Boucan' AND Animal.espece_id = 2;
```

```
1  ERROR 1442 (HY000): Can't update table 'animal' in stored function/trigger
   because it is already used by statement which invoked this stored func
   on/trigger.
```

Le trigger échoue puisque la table *Animal* est utilisée par la requête INSERT qui le déclenche. L'insertion elle-même est donc finalement annulée.

### Clés étrangères

**Une suppression ou modification de données déclenchée par une clé étrangère ne provoquera pas l'exécution du trigger correspondant.** Par exemple, la colonne *Animal.race\_id* possède une clé étrangère, qui référence la colonne *Race.id*. Cette clé étrangère a été définie avec l'option ON DELETE SET NULL. Donc en cas de suppression d'une race, tous les animaux de cette race seront modifiés, et leur *race\_id* changée en NULL. Il s'agit donc d'une modification de données. Mais comme cette modification a été déclenchée par une contrainte de clé étrangère, les éventuels triggers BEFORE UPDATE et AFTER UPDATE de la table *Animal* ne seront pas déclenchés.

En cas d'utilisation de triggers sur des tables présentant des clés étrangères avec ces options, il vaut donc mieux supprimer celles-ci et déplacer ce comportement dans des triggers. Une autre solution est de ne pas utiliser les triggers sur les tables concernées. Vous pouvez alors remplacer les triggers par l'utilisation de procédures stockées et/ou de transactions.

Qu'avons-nous comme clés étrangères dans nos tables ?

- *Race* : CONSTRAINT fk\_race\_espece\_id FOREIGN KEY (espece\_id) REFERENCES Espece (id) ON DELETE CASCADE;
- *Animal* : CONSTRAINT fk\_race\_id FOREIGN KEY (race\_id) REFERENCES Race (id) ON DELETE SET NULL;
- *Animal* : CONSTRAINT fk\_espece\_id FOREIGN KEY (espece\_id) REFERENCES Espece (id);

## Partie 2 : Les fonctions et procédures stockées

- *Animal* : CONSTRAINT fk\_mere\_id FOREIGN KEY (mere\_id) REFERENCES Animal (id) ON DELETE SET NULL;
- *Animal* : CONSTRAINT fk\_pere\_id FOREIGN KEY (pere\_id) REFERENCES Animal (id) ON DELETE SET NULL;

Quatre d'entre elles pourraient donc poser problème. Quatre, sur cinq ! Ce n'est donc pas anodin comme restriction !

On va donc modifier nos clés étrangères pour qu'elles reprennent leur comportement par défaut. Il faudra ensuite créer (ou recréer) quelques triggers pour reproduire le comportement que l'on avait défini. À ceci près que la restriction sur la modification des données d'une table utilisée par l'événement déclencheur fait qu'on ne pourra pas reproduire certains comportements. On ne pourra pas mettre à NULL les colonnes *pere\_id* et *mere\_id* de la table *Animal* en cas de suppression de l'animal de référence.

Voici les commandes :

```
1  -- On supprime les clés
2  ALTER TABLE Race DROP FOREIGN KEY fk_race_espece_id;
3  ALTER TABLE Animal DROP FOREIGN KEY fk_race_id,
4      DROP FOREIGN KEY fk_mere_id,
5      DROP FOREIGN KEY fk_pere_id;
6
7  -- On les recrée sans option
8  ALTER TABLE Race ADD CONSTRAINT fk_race_espece_id FOREIGN KEY
9  (espece_id) REFERENCES Espece (id);
10 ALTER TABLE Animal ADD CONSTRAINT fk_race_id FOREIGN KEY
11 (race_id) REFERENCES Race (id),
12     ADD CONSTRAINT fk_mere_id FOREIGN KEY (mere_id)
13 REFERENCES Animal (id),
14     ADD CONSTRAINT fk_pere_id FOREIGN KEY (pere_id)
15 REFERENCES Animal (id);
16 -- Trigger sur Race
```

```
17 DELIMITER |
18 CREATE TRIGGER before_delete_race BEFORE DELETE
19 ON Race FOR EACH ROW
20 BEGIN
21     UPDATE Animal
22     SET race_id = NULL
23     WHERE race_id = OLD.id;
24 END |
25
26 -- Trigger sur Espece
27 CREATE TRIGGER before_delete_espece BEFORE DELETE
28 ON Espece FOR EACH ROW
29 BEGIN
30     DELETE FROM Race
31     WHERE espece_id = OLD.id;
32 END |
33 DELIMITER ;
```

---

### En résumé

- Un trigger est un objet **stocké dans la base de données**, à la manière d'une table ou d'une procédure stockée. La seule différence est qu'un trigger est **lié à une table**, donc en cas de suppression d'une table, les triggers liés à celle-ci sont supprimés également
- Un trigger **définit une ou plusieurs instructions**, dont l'exécution est **déclenchée par une insertion, une modification ou une suppression** de données dans la table à laquelle le trigger est lié.
- Les instructions du trigger peuvent être exécutées **avant la requête ayant déclenché celui-ci, ou après**. Ce comportement est à définir à la création du trigger.

- Une table ne peut posséder qu'un seul trigger par combinaison événement/moment (BEFORE UPDATE, AFTER DELETE,...)
- Les triggers sous MySQL sont soumis à d'importantes (et potentiellement très gênantes) **restrictions**.

## LES VUES

Le langage SQL acronyme de Structured Query Language (Langage Structuré de Requêtes), a été conçu pour gérer les données dans un SGBDR. A l'aide des DML (Data Manipulation Language ie les requêtes *SELECT*, *INSERT*, *UPDATE*, *DELETE*) il est possible de manipuler ces données qui sont stockées dans des tables.

SQL nous propose une autre interface pour accéder à cette information: **les vues**. Dans ces pages, nous verrons comment créer et se servir des vues, puis avec quelques exemples pratiques, nous allons voir comment les utiliser le mieux possible.

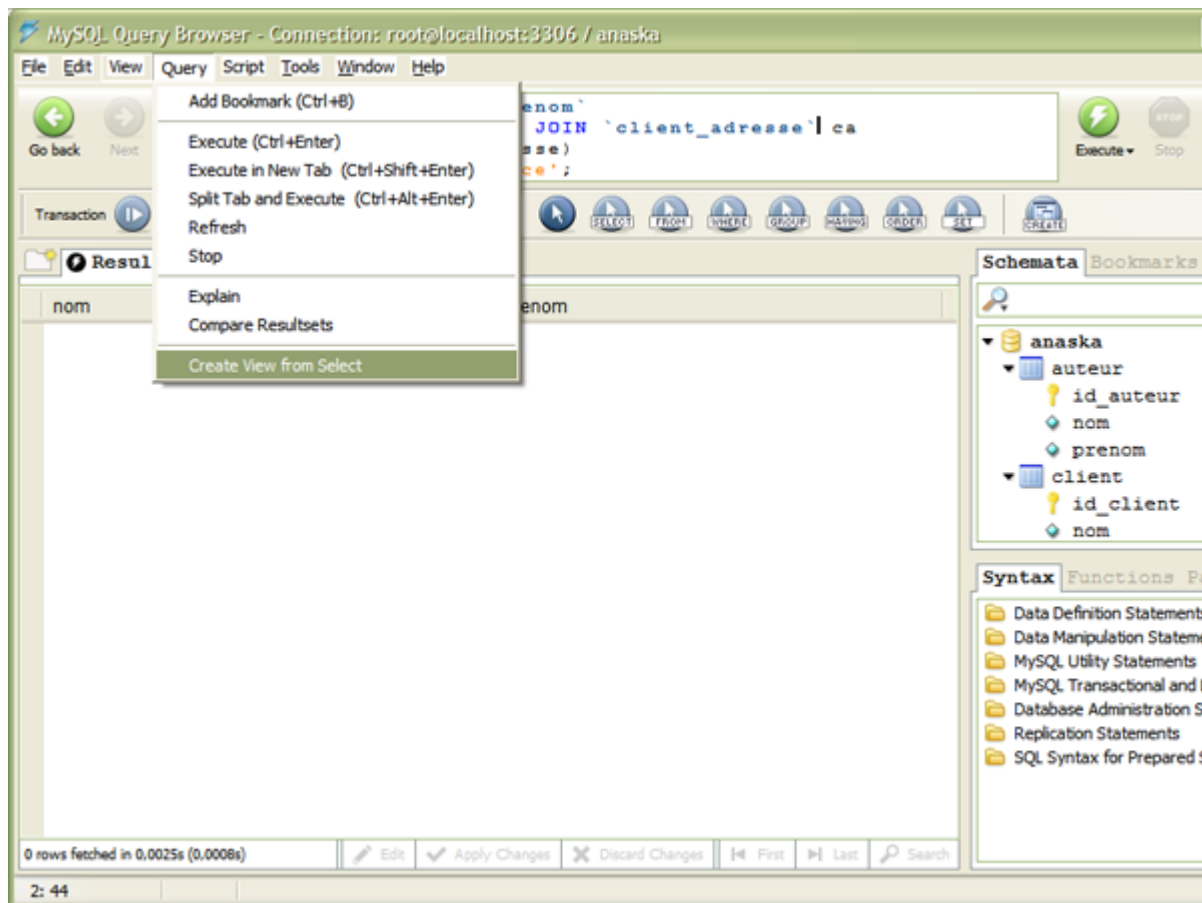
### Qu'est-ce qu'une vue ?

Les vues sont des tables virtuelles issues de l'assemblage d'autres tables en fonction de critères. Techniquement les vues sont créées à l'aide d'une requête *SELECT*. Elles ne stockent pas les données qu'elles contiennent mais conservent juste la requête permettant de les créer.

La requête *SELECT* qui génère la vue référence une ou plusieurs tables. La vue peut donc être, par exemple, une jointure entre différentes tables, l'agrégation ou l'extraction de certaines colonnes d'une table. Elle peut également être créée à partir d'une autre vue.

Les vues sont souvent en lecture seule et ne permettent donc que de lire des données. Cependant MySQL permet la création de vues modifiables sous certaines conditions :

- La requête qui génère la vue doit permettre à MySQL de retrouver la trace de l'enregistrement à modifier dans la ou les tables sous-jacentes ainsi que celle de toutes les valeurs de chaque colonne. La requête *SELECT* créant la vue ne doit donc pas contenir de clause *DISTINCT*, *GROUP BY*, *HAVING*... et autres fonctions d'agrégation. La liste complète est disponible dans la documentation de MySQL.
- L'autre condition est que sa clause *ALGORITHM* ne doit pas être de valeur *TEMPTABLE*. Nous reviendrons sur ce point.



### A quoi servent les vues ?

Les vues peuvent être utilisées pour différentes raisons. Elles permettent de :

- Contrôler l'intégrité en restreignant l'accès aux données pour améliorer la confidentialité.
  - Partitionnement vertical et/ou horizontal pour cacher des champs aux utilisateurs, ce qui permet de personnaliser l'affichage des informations suivant le type d'utilisateur.
- Masquer la complexité du schéma.
  - Indépendance logique des données, utile pour donner aux utilisateurs l'accès à un ensemble de relations représentées sous la forme d'une table. Les données de la vue sont alors des champs de différentes tables regroupées, ou des résultats d'opérations sur ces champs.
- Modifier automatiquement des données sélectionnées (*sum()*, *avg()*, *max()*,...).
  - Manipuler des valeurs calculées à partir d'autres valeurs du schéma.
- Conserver la structure d'une table si elle doit être modifiée.
  - Le schéma peut ainsi être modifié sans qu'il ne soit nécessaire de changer les requêtes du côté applicatif.



### Les droits nécessaires

Pour créer une vue l'utilisateur doit avoir le droit `CREATE VIEW`. Il faut également avoir la permission de sélectionner toutes les colonnes qui apparaissent dans la commande `SELECT` spécifiant ce qu'est la vue.

De plus si la clause `REPLACE` est utilisée, le droit `DROP` est également nécessaire. Le droit `SHOW VIEW` donne la possibilité d'exécuter la commande `SHOW CREATE VIEW`. Cette commande permet d'obtenir les informations de création d'une vue. Une autre façon d'obtenir ces informations est d'interroger la table **view** du schéma **information\_schema**. Cette information sera exhaustive seulement pour les vues que vous avez créé.

### Accorder les droits sur une vue pour un utilisateur

```
GRANT
```

```
SELECT,
```

```
DROP,
```

```
CREATE VIEW,
```

```
SHOW VIEW
```

```
ON `projet`. * TO 'secretaire'@'localhost';
```

### Syntaxe d'une vue

#### **CREATE VIEW**

La commande MySQL pour créer une vue est assez proche de la syntaxe du standard SQL.

### Syntaxe de création d'une vue sous MySQL

```
CREATE VIEW nom_de_la_vue AS requête_select
```

### Création d'une vue pour la relation "etudiant"

```
CREATE TABLE etudiant
```

```
(
```

```
id_etudiant INT UNSIGNED PRIMARY KEY,
```

```
nom CHAR(30),
```

```
prenom CHAR(30),
```

```
age TINYINT UNSIGNED,
```

```
cursus ENUM('Licence', 'Master', 'Doctorat')
```

```
);
```

```
CREATE VIEW v_etudiant_liste AS SELECT nom, prenom FROM etudiant;
```

Après avoir créé la table **etudiant**, on crée la vue **v\_etudiant\_liste** qui contient le nom et le prénom des étudiants.

Il est possible d'ajouter d'autres informations lors de la création de la vue :

### Syntaxe d'une vue MySQL

```
CREATE  
[OR REPLACE]  
[ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]  
[DEFINER = { user | CURRENT_USER }]  
[SQL SECURITY { DEFINER | INVOKER }]  
VIEW nom_de_la_vue [(colonne(s))]  
AS requête_select  
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

Voici dans le détail les différentes clauses.

**OR REPLACE** : si une vue du même nom existe, elle est alors supprimée et remplacée par la nouvelle.

**ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}** : clause non standard, qui prend les valeurs suivantes :

- **UNDEFINED** : C'est la valeur par défaut. MySQL décide lui-même quel algorithme choisir entre **MERGE** et **TEMPTABLE**.
- **MERGE** utilise la requête SQL ayant servi à la création de la vue comme base d'opération. En d'autres termes, faire une requête sur la vue revient à faire la même requête sur la ou les tables sous-jacentes.
- **TEMPTABLE** utilise une table temporaire créée pour stocker (temporairement) les résultats. Un intérêt de cet algorithme est de libérer plus rapidement les verrous sur les tables sous-jacentes. Les autres requêtes sont alors moins pénalisées.

Il faut noter également qu'une vue avec pour valeur **MERGE** sera modifiable alors qu'avec la valeur **TEMPTABLE** elle ne le sera pas.

**DEFINER** = { *user* | **CURRENT\_USER** } : clause non standard qui permet d'assigner un créateur à la vue. Par défaut, le créateur de la vue est *DEFINER = current\_user*, c'est-à-dire, l'utilisateur qui exécute la commande *CREATE VIEW*. Il est cependant possible d'assigner la vue à un autre compte utilisateur, à condition d'avoir le droit *SUPER*.

**SQL SECURITY { DEFINER | INVOKER }** : clause non standard encore qui permet de définir quels seront les droits de l'utilisateur lors de l'exécution de la vue. Deux valeurs sont possibles :

- *DEFINER* qui permet d'exécuter la vue avec les droits du créateur. C'est la valeur par défaut.
- *INVOKER* qui permet d'exécuter la vue avec ses propres droits.

**WITH [CASCADED | LOCAL] CHECK OPTION** : permet de vérifier les contraintes spécifiées dans la clause *WHERE* d'une vue modifiable lorsque l'on y modifie ses données. Deux valeurs sont possibles :

- *CASCADED*, la valeur par défaut. Elle permet de vérifier la contrainte pour la vue ainsi que pour les vues sous-jacentes dont elle dérive.
- *LOCAL* qui permet de vérifier seulement la contrainte de la vue.

### ALTER VIEW

Une fois la vue créée, il est bien évidemment possible de la modifier avec la commande *ALTER VIEW*.

#### Editer une vue à l'ordre ALTER

```
ALTER definer='secretaire'@'localhost'
```

```
VIEW v_etudiant_liste AS SELECT nom, prenom, cursus FROM etudiant;
```

Cette commande modifie la clause *DEFINER* en lui assignant le compte *secretaire@localhost* et modifie la définition de la vue en rajoutant le champ *cursus*.

### DROP VIEW

L'ordre *DROP VIEW* permet d'effacer une vue.

#### Supprimer des vues à partir de l'ordre DROP VIEW

```
DROP VIEW v_etudiant_liste, v_prof_liste;
```

Supprime les vues *v\_etudiant\_liste* et *v\_prof\_liste*. Il est possible d'ajouter la clause *IF EXISTS* qui retourne un avertissement au lieu d'une erreur si la vue à effacer n'existe pas.

### Restrictions

Lors de la création d'une vue, certaines contraintes doivent être prises en compte :

## Partie 2 : Les fonctions et procédures stockées

- Il n'est pas possible de créer un index sur une vue.
- La vue ne peut pas contenir de sous-requêtes dans la clause *FROM* du *SELECT*.
- Il n'est pas possible d'utiliser de variables dans une vue.
- Les objets (tables et vues) nécessaires à la création de la vue doivent exister avant de la créer.
- Si un objet référencé par la vue est effacé, la vue n'est alors plus accessible.
- Une vue ne peut référencer une table temporaire (*TEMPORARY TABLE*).
- Il n'est pas possible de créer des vues temporaires.
- Il n'est pas possible d'associer un **trigger** à une vue.
- La définition d'une vue est "**gelée**" dans une requête préparée.

Par exemple :

```
mysql> CREATE VIEW ma_vue AS SELECT 'première valeur';
```

Query OK, 0 rows affected (0.24 sec)

```
mysql> desc ma_vue;
```

FIELD	Type	NULL	Key	DEFAULT	Extra
première valeur	<b>VARCHAR</b> (15)	NO			

1 row IN SET (0.50 sec)

```
mysql> PREPARE req_prepare FROM 'SELECT * FROM ma_vue';
```

Query OK, 0 rows affected (0.00 sec)

Statement prepared

```
mysql> EXECUTE req_prepare;
```

```
+-----+
```

## Partie 2 : Les fonctions et procédures stockées

```
| première valeur |
```

```
+-----+
```

```
| première valeur |
```

```
+-----+
```

```
1 row IN SET (0.01 sec)
```

```
ALTER VIEW ma_vue AS SELECT 'deuxième valeur';
```

```
Query OK, 0 rows affected (0.05 sec)
```

```
mysql> desc ma_vue;
```

```
+-----+-----+-----+-----+-----+
```

```
| FIELD      | Type      | NULL | Key | DEFAULT | Extra |
```

```
+-----+-----+-----+-----+-----+
```

```
| deuxième valeur | VARCHAR(15) | NO   |     |         |      |
```

```
+-----+-----+-----+-----+-----+
```

```
1 row IN SET (0.00 sec)
```

```
mysql> EXECUTE req_prepare;
```

```
+-----+
```

```
| première valeur |
```

```
+-----+
```

```
| première valeur |
```

```
+-----+
```

```
1 row IN SET (0.00 sec)
```

Il faut en fait recréer la requête préparée :

**Utiliser une vue dans une requête préparée**

```
mysql> DEALLOCATE PREPARE req_prepare;
```

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

```
mysql> PREPARE req_prepare FROM 'SELECT * FROM ma_vue';
```

Query OK, 0 rows affected (0.00 sec)

Statement prepared

```
mysql> EXECUTE req_prepare;
```

```
+-----+
```

```
| deuxième valeur |
```

```
+-----+
```

```
| deuxième valeur |
```

```
+-----+
```

1 row IN SET (0.00 sec)

### Utiliser les vues

Voici quelques exemples pratiques très simples pour illustrer les différents besoins que peuvent combler les vues. On aura ici, une vue administrateur de base de données. Les objets créés ne seront pas utilisés directement par les utilisateurs mais aux travers d'une application.

### ***Contrôler l'intégrité en restreignant l'accès aux données pour améliorer la confidentialité***

La table **employe** de mon application, contient toutes les informations sur les employés.

### **Structure de la table " employe "**

```
CREATE TABLE `employe`
```

```
(
```

```
  `id_employe` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
```

```
  `nom` CHAR(45) NOT NULL,
```

```
  `prenom` CHAR(45) NOT NULL,
```

```
  `tel_perso` CHAR(10) NOT NULL,
```

```
  `tel_bureau` CHAR(10) NOT NULL,
```

```
  `statut` CHAR(45) NOT NULL,
```

```
  `ville` CHAR(45) NOT NULL,
```

```
`salaire` DECIMAL(7,2) NOT NULL,  
PRIMARY KEY (`id_employe`)  
);
```

Toutes les informations présentes dans cette table ne sont pas pertinentes pour les trois types d'utilisateurs suivant : le comptable, la secrétaire pour Paris et la secrétaire pour le reste de la France.

Une solution est donc de créer une vue par type.

Pour le comptable, il faut avoir accès aux champs nom, prénom, téléphone du bureau, statut et salaire de chaque employé. On fait donc un partitionnement vertical de la table **employe**. La vue correspondante est la suivante :

### **Création de la vue pour le comptable**

```
CREATE ALGORITHM=MERGE SQL SECURITY DEFINER VIEW `v_comptable` AS  
SELECT nom, prenom, tel_bureau, statut, salaire FROM employe;
```

Le profil "secrétaire pour Paris", n'a pas besoin de l'identifiant et il ne doit surtout pas avoir accès aux salaires, cette information étant confidentielle. Autre restriction, ce profil ne gère que les employés de la filiale de Paris. Le partitionnement est vertical et horizontal.

### **Création de la vue pour la secrétaire de Paris**

```
CREATE ALGORITHM= MERGE SQL SECURITY DEFINER VIEW `v_secretaire_paris` AS  
SELECT nom, prenom, tel_perso, tel_bureau, statut FROM employe  
WHERE ville = 'Paris';
```

Notre troisième vue est très proche de la deuxième. La seule différence vient du fait que là, on veut les employés qui ne travaillent pas à Paris.

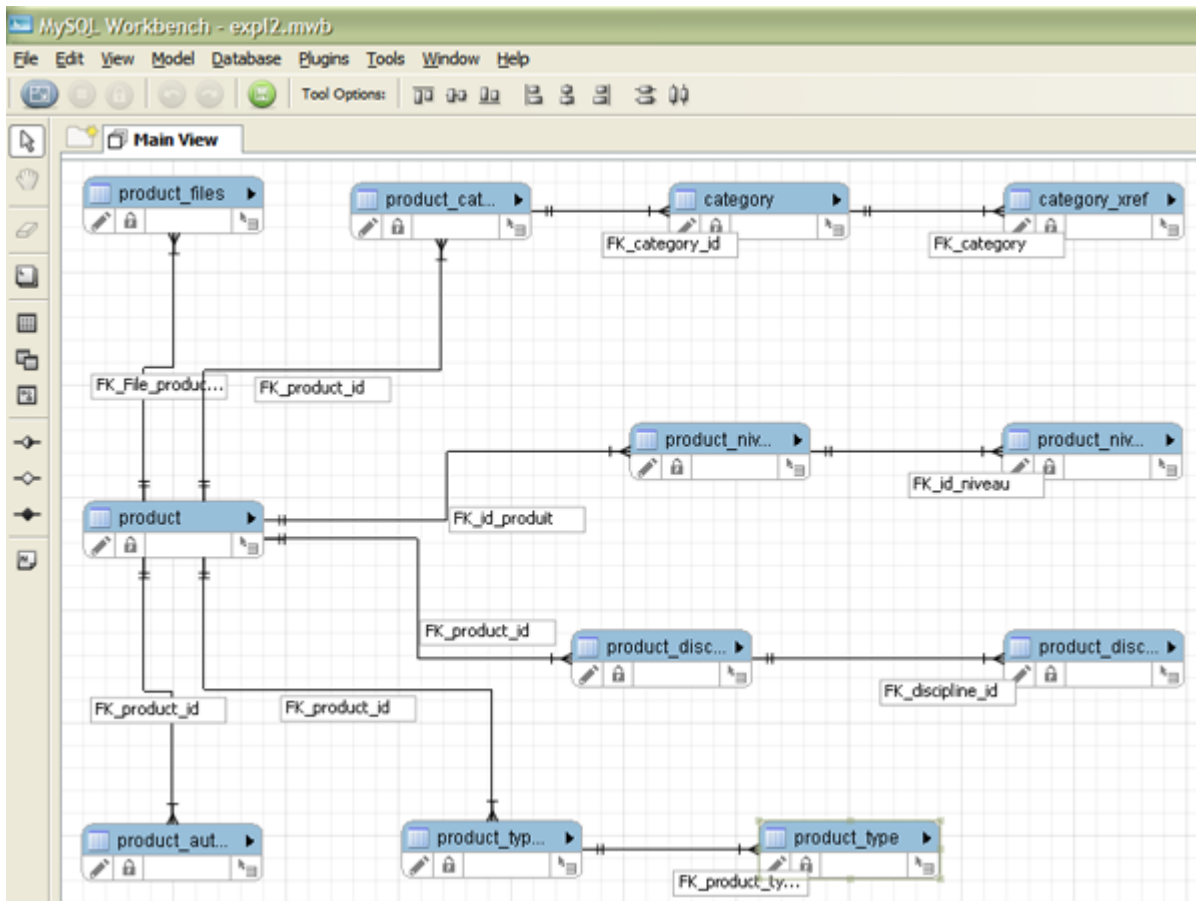
### **Création de la vue pour les employés de Province**

```
CREATE ALGORITHM= MERGE SQL SECURITY DEFINER VIEW `v_secretaire_autre` AS  
SELECT nom, prenom, tel_perso, tel_bureau, statut FROM employe  
WHERE ville <> 'Paris';
```

### **Masquer la complexité du schéma**

L'équipe de développement doit écrire un moteur de recherches pour une application de commerce électronique. Voici un extrait des tables de la base de données impliquées dans la recherche des produits du site.

## Partie 2 : Les fonctions et procédures stockées



La difficulté est de générer la bonne requête avec les bonnes jointures (plus d'une dizaine), à chaque recherche. Une solution pour faciliter le travail des développeurs est de créer une vue qui fait référence à toutes les tables impliquées dans la recherche des produits. Les recherches se feront donc sur cette vue, avec des requêtes plus simples à écrire.

La vue est créée avec l'algorithme *TEMPTABLE*, les verrous sur les tables sous-jacentes seront libérés plus rapidement ce qui permettra de moins pénaliser les autres requêtes. *TEMPTABLE* a la particularité de rendre la vue non modifiable, mais cela ne gêne pas du tout, car la vue n'est accédée qu'en lecture.

### Extrait de la vue " moteur de recherches "

```
CREATE ALGORITHM = TEMPTABLE
```

```
VIEW moteur_de_recherche
```

```
AS
```

```
SELECT
```

```
    s.product_name as nom_produit,
```

```
    ... (les champs nécessaires)
```

```
FROM product s
```

```
    LEFT JOIN product_files sf ON s.product_id=sf.file_product_id
```

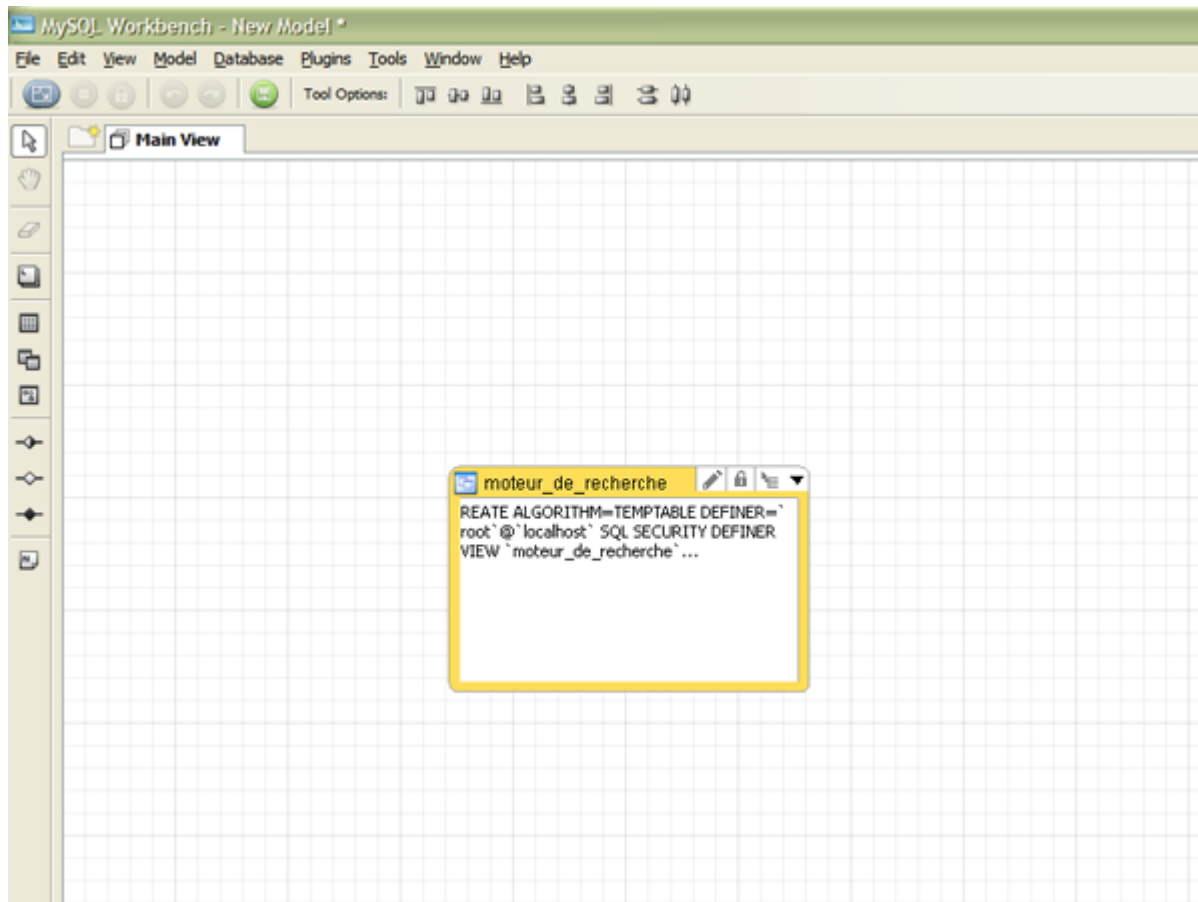


## Partie 2 : Les fonctions et procédures stockées

LEFT JOIN ..... (toutes les tables impliquées)

WHERE ... ;

Cette astuce permet de "simplifier" de façon assez significative le schéma.



Un dernier mot pour noter que cette solution ne permet pas d'améliorer les performances de la recherche, dans le sens où la requête qui génère la vue est lancée à chaque appel de cette dernière.

### **Modifier automatiquement des données sélectionnées**

Pour ce troisième exemple, nous allons nous intéresser au schéma (là encore très simplifié) d'une application qui permet de vendre des produits en France et au Royaume-Uni, en euro, livres et dollars. Cette application possède une table **produit**, qui contient le produit (son identifiant) et son prix hors taxe en euro.

### **Structure de la table " produit "**

```
CREATE TABLE produit
```

```
(
```

```
  id_produit MEDIUMINT(8) UNSIGNED NOT NULL AUTO_INCREMENT,
```

```
  prix_ht DECIMAL(6,2) DEFAULT NULL,
```

```
  PRIMARY KEY (id_produit)
```

);

Nous disposons également des tables **devise** et **tva** qui gèrent respectivement le taux de change des devises et la TVA de différents pays.

### Structures des tables " devise " et " tva "

#### # Table devise

```
CREATE TABLE devise
```

```
(
```

```
    devise ENUM('Euro', 'Dollar', 'Livre') NOT NULL,
```

```
    valeur DECIMAL(6,5) DEFAULT NULL,
```

```
    PRIMARY KEY (devise)
```

```
);
```

```
INSERT INTO devise VALUES ('Livre',0.66017);
```

```
INSERT INTO devise VALUES ('Dollar',1.29852);
```

```
INSERT INTO devise VALUES ('Euro',1);
```

#### # Table tva

```
CREATE TABLE tva
```

```
(
```

```
    pays ENUM('France', 'Royaume-Uni') NOT NULL,
```

```
    normal DECIMAL(3,1) DEFAULT NULL,
```

```
    reduit DECIMAL(3,1) DEFAULT NULL,
```

```
    PRIMARY KEY (pays)
```

```
);
```

```
INSERT INTO tva VALUES ('Royaume-Uni',17.5,5.0);
```

```
INSERT INTO tva VALUES ('France',19.6,5.5);
```

Le besoin est le suivant : disposer simplement des prix TTC pour chaque pays. On va donc créer deux vues par pays qui nous permettront de disposer des prix TTC en fonction de la devise.

## Partie 2 : Les fonctions et procédures stockées

La vue **produit\_france**, contient les produits, le prix TTC et le prix TTC réduit qui correspond à l'ajout de la TVA réduite. Les prix sont en euros.

### Création de la vue " produit\_france "

```
CREATE VIEW `produit_france` AS

SELECT `produit`.`id_produit` AS `produit`,

    ROUND((((`produit`.`prix_ht` * `tva`.`normal`) / 100) + `produit`.`prix_ht`) *
    `devise`.`valeur`),2) AS `Prix_ttc_€`,

    ROUND((((`produit`.`prix_ht` * `tva`.`reduit`) / 100) + `produit`.`prix_ht`) *
    `devise`.`valeur`),2) AS `Prix_ttc_reduit_€`

FROM ((`produit` JOIN `tva`) JOIN `devise`)

WHERE ((`tva`.`pays` = 'France') AND (`devise`.`devise` = 'Euro'));
```

Certains clients préférant la monnaie de l'oncle Sam, une deuxième vue, **produit\_france\_dollar**, est nécessaire pour avoir les prix en dollar.

### Création de la vue " produit\_france\_dollar "

```
CREATE VIEW `produit_france_dollar` AS

SELECT `produit`.`id_produit` AS `produit`,

    ROUND((((`produit`.`prix_ht` * `tva`.`normal`) / 100) + `produit`.`prix_ht`) *
    `devise`.`valeur`),2) AS `Prix_ttc_$`,

    ROUND((((`produit`.`prix_ht` * `tva`.`reduit`) / 100) + `produit`.`prix_ht`) *
    `devise`.`valeur`),2) AS `Prix_ttc_reduit_$`

FROM ((`produit` JOIN `tva`) JOIN `devise`)

WHERE ((`tva`.`pays` = 'France') AND (`devise`.`devise` = 'Dollar'));
```

Même principe pour le Royaume-Uni :

### Création de la vue " produit\_royaume\_uni "

```
CREATE VIEW `produit_royaume_uni` AS

SELECT `produit`.`id_produit` AS `produit`,

    ROUND((((`produit`.`prix_ht` * `tva`.`normal`) / 100) + `produit`.`prix_ht`) *
    `devise`.`valeur`),2) AS `Net_price_£`,

    ROUND((((`produit`.`prix_ht` * `tva`.`reduit`) / 100) + `produit`.`prix_ht`) *
    `devise`.`valeur`),2) AS `reduced_Net_price_£`

FROM ((`produit` JOIN `tva`) JOIN `devise`)
```

## Partie 2 : Les fonctions et procédures stockées

```
WHERE ((`tva`.`pays` = 'Royaume-Uni') AND (`devise`.`devise` = 'Livre'));
```

Avec les prix en dollars :

### Création de la vue " produit\_royaux\_uni\_dollar "

```
CREATE VIEW `produit_royaume_uni_dollar` AS
SELECT `produit`.`id_produit` AS `produit`,
       ROUND((((`produit`.`prix_ht` * `tva`.`normal`) / 100) + `produit`.`prix_ht`) *
`devise`.`valeur`),2) AS `Net_price_$`,
       ROUND((((`produit`.`prix_ht` * `tva`.`reduit`) / 100) + `produit`.`prix_ht`) *
`devise`.`valeur`),2) AS `reduced_Net_price_$`
FROM ((`produit` JOIN `tva`) JOIN `devise`)
WHERE ((`tva`.`pays` = 'Royaume-Uni') AND (`devise`.`devise` = 'Dollar'));
```

### Conserver la structure d'une table si elle doit être modifiée

La problématique est de mettre à jour le schéma de l'application en changeant la structure de certaines tables.

Changer le schéma a comme principal impact d'obliger de modifier les requêtes de l'application. Il sera donc nécessaire de les identifier pour les mettre à jour à leur tour, ce qui peut rapidement devenir fastidieux. Au travers de l'exemple qui suit, nous allons créer une vue qui va masquer le changement de table ce qui nous évite de modifier les requêtes applicatives. Une nouvelle version de l'application pourra utiliser la nouvelle table sans être obligée d'utiliser la vue, on assure ainsi la compatibilité ascendante.

Ma table de départ est la table *livre*.

### Structure de la table " livre "

```
CREATE TABLE `livre`
(
  `id_livre` CHAR(17) NOT NULL,
  `auteur` CHAR(50) DEFAULT NULL,
  PRIMARY KEY (`id_livre`)
);
```

Les requêtes, du côté de l'application, sont les suivantes:

### Requêtes applicatives de la table " livre "

## Partie 2 : Les fonctions et procédures stockées

```
SELECT id_livre FROM livre;
```

```
SELECT auteur FROM livre;
```

```
SELECT * FROM livre;
```

De cette structure où je ne peux gérer que des livres, j'en crée une autre qui m'offre plus de souplesse, la table **produit** :

### Structure de la table " produit "

```
CREATE TABLE `produit`  
(  
  `id_produit` MEDIUMINT(9) NOT NULL AUTO_INCREMENT,  
  `isbn` CHAR(17) DEFAULT NULL,  
  `auteur` CHAR(50) DEFAULT NULL,  
  PRIMARY KEY (`id_produit`),  
  UNIQUE KEY `isbn` (`isbn`)  
);
```

Les seuls produits disponibles sont mes livres, je remplis donc ma table produit avec le contenu de la table **livre** :

### Import de données dans la table " produit "

```
INSERT INTO produit (isbn, auteur) SELECT id_livre, auteur FROM livre;
```

La dernière phase consiste à créer la vue « **livre** », il me faut donc au préalable effacer la table du même nom. Les vues et les tables partageant le même espace de nom.

### Création de la vue " livre "

```
DROP TABLE livre;
```

```
CREATE VIEW livre AS SELECT isbn AS id_livre, auteur FROM produit;
```

Les changements sont transparents pour les trois requêtes de mon application.

### Conclusion

Voici un petit tour d'horizon sur les vues, qui nous l'espérons aura contribué à affiner votre vision sur ce sujet. Il est certain que ces tables virtuelles amènent une certaine souplesse au schéma et il serait dommage de ne pas en profiter. Cependant, ce n'est pas non plus une

## ***Partie 2 : Les fonctions et procédures stockées***

solution miracle, car ajouter des objets peut rapidement rendre le schéma complexe. Maintenant à vous de voir dans quels cas les vues pourront vous être utiles.

## PROCEDURES ET FONCTIONS STOCKEES

Les procédures stockées sont disponibles depuis la version 5 de MySQL, et permettent d'automatiser des actions, qui peuvent être très complexes.

Une procédure stockée est en fait une **série d'instructions SQL** désignée par un **nom**. Lorsque l'on crée une procédure stockée, on l'enregistre **dans la base de données** que l'on utilise, au même titre qu'une table par exemple. Une fois la procédure créée, il est possible d'**appeler** celle-ci, par son nom. Les instructions de la procédure sont alors exécutées.

Contrairement aux requêtes préparées, qui ne sont gardées en mémoire que pour la session courante, les procédures stockées sont, comme leur nom l'indique, **stockées de manière durable**, et font bien **partie intégrante de la base de données** dans laquelle elles sont enregistrées.

- [Création et utilisation d'une procédure](#)
- [Les paramètres d'une procédure stockée](#)
- [Suppression d'une procédure](#)
- [Avantages, inconvénients et usage des procédures stockées](#)

### [Création et utilisation d'une procédure](#)

Voyons tout de suite la syntaxe à utiliser pour créer une procédure :

```
1  CREATE PROCEDURE nom_procedure ([parametre1 [, parametre2, ...]])
2  corps de la procédure;
```

Décodons tout ceci.

- CREATE PROCEDURE : sans surprise, il s'agit de la commande à exécuter pour créer une procédure. On fait suivre cette commande du nom que l'on veut donner à la nouvelle procédure.
- ([parametre1 [, parametre2, ...]]) : après le nom de la procédure viennent des parenthèses. **Celles-ci sont obligatoires** ! À l'intérieur de ces parenthèses, on définit les éventuels paramètres de la procédure. Ces paramètres sont des variables qui pourront être utilisées par la procédure.
- corps de la procédure : c'est là que l'on met le **contenu** de la procédure, ce qui va être exécuté lorsqu'on lance la procédure. Cela peut être soit **une seule requête**, soit **un bloc d'instructions**.

## Partie 2 : Les fonctions et procédures stockées

Les noms des procédures stockées ne sont pas sensibles à la casse.

### Procédure avec une seule requête

Voici une procédure toute simple, sans paramètres, qui va juste afficher toutes les races d'animaux.

```
1  CREATE PROCEDURE afficher_races_requete() -- pas de paramètres dans
2  les parenthèses
   SELECT id, nom, espece_id, prix FROM Race;
```

### Procédure avec un bloc d'instructions

Pour délimiter un bloc d'instructions (qui peut donc contenir plus d'une instruction), on utilise les mots BEGIN et END.

```
1  BEGIN
2      -- Série d'instructions
3  END;
```

**Exemple** : reprenons la procédure précédente, mais en utilisant un bloc d'instructions.

```
1  CREATE PROCEDURE afficher_races_bloc() -- pas de paramètres dans les
2  parenthèses
3  BEGIN
4      SELECT id, nom, espece_id, prix FROM Race;
   END;
```

Malheureusement...

```
1  ERROR 1064 (42000): You have an error in your SQL syntax; check
   the manual
```



```
that corresponds to your MySQL server version for the right synta
x to use near " at line 3
```

Que s'est-il passé ? La syntaxe semble correcte...

Les mots-clés sont bons, il n'y a pas de paramètres mais on a bien mis les parenthèses, BEGIN et END sont tous les deux présents. Tout cela est correct, et pourtant, nous avons visiblement omis un détail.

Peut-être aurez-vous compris que le problème se situe au niveau du caractère ; : en effet, un ; termine une instruction SQL. Or, on a mis un ; à la suite de SELECT \* FROM Race;. Cela semble logique, mais pose problème puisque c'est le premier ; rencontré par l'instruction CREATE PROCEDURE, qui naturellement pense devoir s'arrêter là. Ceci déclenche une erreur puisqu'en réalité, l'instruction CREATE PROCEDURE n'est pas terminée : le bloc d'instructions n'est pas complet !

Comment faire pour écrire des instructions à l'intérieur d'une instruction alors ?

Il suffit de changer le délimiteur !

### Délimiteur

Ce qu'on appelle délimiteur, c'est tout simplement (par défaut), le caractère ;. C'est-à-dire le caractère qui permet de **délimiter les instructions**. Or, il est tout à fait possible de définir le délimiteur manuellement, de manière à ce que ; ne signifie plus qu'une instruction se termine. Auquel cas le caractère ; pourra être utilisé à l'intérieur d'une instruction, et donc pourra être utilisé dans le corps d'une procédure stockée.

Pour changer le délimiteur, il suffit d'utiliser cette commande :

```
1 DELIMITER |
```

À partir de maintenant, vous devrez utiliser le caractère | pour signaler la fin d'une instruction. ; ne sera plus compris comme tel par votre session.

```
1 SELECT 'test' |
```

test

test

DELIMITER n'agit que pour la **session courante**.

Vous pouvez utiliser le (ou les) caractère(s) de votre choix comme délimiteur. Bien entendu, il vaut mieux choisir quelque chose qui ne risque pas d'être utilisé dans une instruction. Bannissez donc les lettres, chiffres, @ (qui servent pour les variables utilisateurs) et les \ (qui servent à échapper les caractères spéciaux).

Les deux délimiteurs suivants sont les plus couramment utilisés :

```
1 DELIMITER //
```

```
2 DELIMITER |
```

Bien ! Ceci étant réglé, reprenons !

### Création d'une procédure stockée

```
1 DELIMITER | -- On change le délimiteur
```

```
2 CREATE PROCEDURE afficher_races() -- toujours pas de paramètres, toujours des parent
```

```
3 BEGIN
```

```
4 SELECT id, nom, espece_id, prix
```

```
5 FROM Race; -- Cette fois, le ; ne nous embêtera pas
```

```
6 END | -- Et on termine bien sûr la commande CREATE PROCEDURE par no
```

Cette fois-ci, tout se passe bien. La procédure a été créée.

Lorsqu'on utilisera la procédure, quel que soit le délimiteur défini par DELIMITER, les instructions à l'intérieur du corps de la procédure seront bien délimitées par ;. En effet, lors de la création d'une procédure, celle-ci est interprétée – on dit aussi "parsée" – par le serveur MySQL et le parseur des procédures stockées interprétera toujours ; comme délimiteur. Il n'est pas influencé par la commande DELIMITER.

Les procédures stockées n'étant que très rarement composées d'une seule instruction, on utilise presque toujours un bloc d'instructions pour le corps de la procédure.

### Utilisation d'une procédure stockée

## Partie 2 : Les fonctions et procédures stockées

Pour appeler une procédure stockée, c'est-à-dire déclencher l'exécution du bloc d'instructions constituant le corps de la procédure, il faut utiliser le mot-clé **CALL**, suivi du nom de la procédure appelée, puis de parenthèses (avec éventuellement des paramètres).

```
1  CALL afficher_races()| -- le délimiteur est toujours | !!!
```

id	nom	espece_id	prix
1	Berger allemand	1	485.00
2	Berger blanc suisse	1	935.00
3	Singapura	2	985.00
4	Bleu russe	2	835.00
5	Maine coon	2	735.00
7	Sphynx	2	1235.00
8	Nebelung	2	985.00
9	Rottweiler	1	600.00

Le bloc d'instructions a bien été exécuté (un simple **SELECT** dans ce cas).

### [Les paramètres d'une procédure stockée](#)

Maintenant que l'on sait créer une procédure et l'appeler, intéressons-nous aux paramètres.

#### Sens des paramètres

Un paramètre peut être de trois sens différents : entrant (IN), sortant (OUT), ou les deux (INOUT).

- **IN** : c'est un paramètre "entrant". C'est-à-dire qu'il s'agit d'un paramètre dont la valeur est fournie à la procédure stockée. Cette valeur sera utilisée pendant la procédure (pour un calcul ou une sélection par exemple).
- **OUT** : il s'agit d'un paramètre "sortant", dont la valeur va être établie au cours de la procédure et qui pourra ensuite être utilisé en dehors de cette procédure.
- **INOUT** : un tel paramètre sera utilisé pendant la procédure, verra éventuellement sa valeur modifiée par celle-ci, et sera ensuite utilisable en dehors.

#### Syntaxe

## Partie 2 : Les fonctions et procédures stockées

Lorsque l'on crée une procédure avec un ou plusieurs paramètres, chaque paramètre est défini par trois éléments.

- Son sens : entrant, sortant, ou les deux. Si aucun sens n'est donné, il s'agira d'un paramètre IN par défaut.
- Son nom : indispensable pour le désigner à l'intérieur de la procédure.
- Son type : INT, VARCHAR(10),...

### Exemples

#### Procédure avec un seul paramètre entrant

Voici une procédure qui, selon l'*id* de l'espèce qu'on lui passe en paramètre, affiche les différentes races existant pour cette espèce.

```
1  DELIMITER |                                -- Facultatif si votre délimiteur
2  est toujours |
3  CREATE PROCEDURE afficher_race_selon_espece (IN p_espece_id INT) --
4  Définition du paramètre p_espece_id
5  BEGIN
6      SELECT id, nom, espece_id, prix
7      FROM Race
8      WHERE espece_id = p_espece_id;          -- Utilisation du
paramètre
9  END |
10 DELIMITER ;                                -- On remet le délimiteur par
défaut
```

Notez que, suite à la création de la procédure, j'ai remis le délimiteur par défaut ;. Ce n'est absolument pas obligatoire, vous pouvez continuer à travailler avec | si vous préférez.

Pour l'utiliser, il faut donc passer une valeur en paramètre de la procédure. Soit directement, soit par l'intermédiaire d'une variable utilisateur.

```
1  CALL afficher_race_selon_espece(1);
2  SET @espece_id := 2;
```

```
3 CALL afficher_race_selon_espece(@espece_id);
```

id	nom	espece_id	prix
1	Berger allemand	1	485.00
2	Berger blanc suisse	1	935.00
9	Rottweiller	1	600.00
id	nom	espece_id	prix
3	Singapura	2	985.00
4	Bleu russe	2	835.00
5	Maine coon	2	735.00
7	Sphynx	2	1235.00
8	Nebelung	2	985.00

Le premier appel à la procédure affiche bien toutes les races de chiens, et le second, toutes les races de chats.

J'ai fait commencer le nom du paramètre par "p\_". Ce n'est pas obligatoire, mais je vous conseille de le faire systématiquement pour vos paramètres afin de les distinguer facilement. Si vous ne le faites pas, soyez extrêmement prudents avec les noms que vous leur donnez. Par exemple, dans cette procédure, si on avait nommé le paramètre *espece\_id*, cela aurait posé problème, puisque *espece\_id* est aussi le nom d'une colonne dans la table *Race*. Qui plus est, c'est le nom de la colonne dont on se sert dans la condition WHERE. En cas d'ambiguïté, MySQL interprète l'élément comme étant le paramètre, et non la colonne. On aurait donc eu WHERE 1 = 1 par exemple, ce qui est toujours vrai.

### Procédure avec deux paramètres, un entrant et un sortant

Voici une procédure assez similaire à la précédente, si ce n'est qu'elle n'affiche pas les races existant pour une espèce, mais compte combien il y en a, puis stocke cette valeur dans un paramètre sortant.

```
1 DELIMITER |
2 CREATE PROCEDURE compter_races_selon_espece (p_espece_id INT, OUT
3 p_nb_races INT)
```

```
4 BEGIN
5     SELECT COUNT(*) INTO p_nb_races
6     FROM Race
7     WHERE espece_id = p_espece_id;
8 END |
DELIMITER ;
```

Aucun sens n'a été précisé pour *p\_espece\_id*, il est donc considéré comme un paramètre entrant.

SELECT COUNT(\*) INTO p\_nb\_races. Voilà qui est nouveau ! Comme vous l'avez sans doute deviné, le mot-clé INTO placé après la clause SELECT permet d'**assigner les valeurs sélectionnées** par ce SELECT à des variables, au lieu de simplement afficher les valeurs sélectionnées. Dans le cas présent, la valeur du COUNT(\*) est assignée à *p\_nb\_races*.

Pour pouvoir l'utiliser, il est nécessaire que le SELECT ne renvoie qu'**une seule ligne**, et il faut que le nombre de valeurs sélectionnées et le nombre de variables à assigner **soient égaux** :

**Exemple 1** : SELECT ... INTO correct avec deux valeurs

```
1 SELECT id, nom INTO @var1, @var2
2 FROM Animal
3 WHERE id = 7;
4 SELECT @var1, @var2;
```

```
@var1 @var2
```

```
7      Caroline
```

Le SELECT ... INTO n'a rien affiché, mais a assigné la valeur 7 à *@var1*, et la valeur 'Caroline' à *@var2*, que nous avons ensuite affichées avec un autre SELECT.

**Exemple 2** : SELECT ... INTO incorrect, car le nombre de valeurs sélectionnées (deux) n'est pas le même que le nombre de variables à assigner (une).

```
1  SELECT id, nom INTO @var1
2  FROM Animal
3  WHERE id = 7;
```

```
1  ERROR 1222 (21000): The used SELECT statements have a different number of columns
```

**Exemple 3** : SELECT ... INTO incorrect, car il y a plusieurs lignes de résultats.

```
1  SELECT id, nom INTO @var1, @var2
2  FROM Animal
3  WHERE espece_id = 5;
```

```
1  ERROR 1172 (42000): Result consisted of more than one row
```

Revenons maintenant à notre nouvelle procédure *compter\_races\_selon\_espece()* et exécutons-la. Pour cela, il va falloir lui passer deux paramètres : *p\_espece\_id* et *p\_nb\_races*. Le premier ne pose pas de problème, il faut simplement donner un nombre, soit directement soit par l'intermédiaire d'une variable, comme pour la procédure *afficher\_race\_selon\_espece()*. Par contre, pour le second, il s'agit d'un paramètre sortant. Il ne faut donc pas donner une valeur, mais quelque chose dont la valeur sera déterminée par la procédure (grâce au SELECT ... INTO), et qu'on pourra utiliser ensuite : **une variable utilisateur** !

```
1  CALL compter_races_selon_espece (2, @nb_races_chats);
```

Et voilà ! La variable *@nb\_races\_chats* contient maintenant le nombre de races de chats. Il suffit de l'afficher pour vérifier.

```
1 SELECT @nb_races_chats;
```

```
@nb_races_chats
```

```
5
```

### Procédure avec deux paramètres, un entrant et un entrant-sortant

Nous allons créer une procédure qui va servir à calculer le prix que doit payer un client. Pour cela, deux paramètres sont nécessaires : l'animal acheté (paramètre IN), et le prix à payer (paramètre INOUT). La raison pour laquelle le prix est un paramètre à la fois entrant et sortant est qu'on veut pouvoir, avec cette procédure, calculer simplement un prix total dans le cas où un client achèterait plusieurs animaux. Le principe est simple : si le client n'a encore acheté aucun animal, le prix est de 0. Pour chaque animal acheté, on appelle la procédure, qui ajoute au prix total le prix de l'animal en question. Une fois n'est pas coutume, commençons par voir les requêtes qui nous serviront à tester la procédure. Cela devrait clarifier le principe. Je vous propose d'essayer ensuite d'écrire vous-mêmes la procédure correspondante avant de regarder à quoi elle ressemble.

```
1 SET @prix = 0;           -- On initialise @prix à 0
2
3 CALL calculer_prix (13, @prix); -- Achat de Rouquine
4 SELECT @prix AS prix_intermediaire;
5
6 CALL calculer_prix (24, @prix); -- Achat de Cartouche
7 SELECT @prix AS prix_intermediaire;
8
9 CALL calculer_prix (42, @prix); -- Achat de Bilba
10 SELECT @prix AS prix_intermediaire;
11
12 CALL calculer_prix (75, @prix); -- Achat de Mimi
13 SELECT @prix AS total;
```



## Partie 2 : Les fonctions et procédures stockées

On passe donc chaque animal acheté tour à tour à la procédure, qui modifie le prix en conséquence. Voici quelques indices et rappels qui devraient vous aider à écrire vous-mêmes la procédure.

- Le prix n'est pas un nombre entier.
- Il est possible de faire des additions directement dans un SELECT.
- Pour déterminer le prix, il faut utiliser la fonction COALESCE().

Réponse :

Afficher/Masquer le contenu masqué

Et voici ce qu'affichera le code de test :

prix_intermediaire
485.00
prix_intermediaire
685.00
prix_intermediaire
1420.00
total
1430.00

Voilà qui devrait nous simplifier la vie. Et nous n'en sommes qu'au début des possibilités des procédures stockées !

### [Suppression d'une procédure](#)

Vous commencez à connaître cette commande : pour supprimer une procédure, on utilise DROP (en précisant qu'il s'agit d'une procédure).

**Exemple :**

1	<b>DROP PROCEDURE</b> afficher_races;
---	---------------------------------------

Pour rappel, les procédures stockées ne sont pas détruites à la fermeture de la session mais bien enregistrées comme un élément de la base de données, au même titre qu'une table par exemple.

## Partie 2 : Les fonctions et procédures stockées

Notons encore qu'il n'est pas possible de modifier une procédure directement. La seule façon de modifier une procédure existante est de la supprimer puis de la recréer avec les modifications.

Il existe bien une commande ALTER PROCEDURE, mais elle ne permet de changer ni les paramètres, ni le corps de la procédure. Elle permet uniquement de changer certaines caractéristiques de la procédure, et ne sera pas couverte dans ce cours.

### Avantages, inconvénients et usage des procédures stockées

#### Avantages

Les procédures stockées permettent de **réduire les allers-retours entre le client et le serveur MySQL**. En effet, si l'on englobe en une seule procédure un processus demandant l'exécution de plusieurs requêtes, le client ne communique qu'une seule fois avec le serveur (pour demander l'exécution de la procédure) pour exécuter la totalité du traitement. Cela permet donc un certain **gain en performance**.

Elles permettent également de **sécuriser** une base de données. Par exemple, il est possible de **restreindre les droits des utilisateurs** de façon à ce qu'ils puissent **uniquement exécuter des procédures**. Finis les DELETE dangereux ou les UPDATE inconsidérés. Chaque requête exécutée par les utilisateurs est créée et contrôlée par l'administrateur de la base de données par l'intermédiaire des procédures stockées.

Cela permet ensuite de **s'assurer qu'un traitement est toujours exécuté de la même manière**, quelle que soit l'application/le client qui le lance. Il arrive par exemple qu'une même base de données soit exploitée par plusieurs applications, lesquelles peuvent être écrites avec différents langages. Si on laisse chaque application avoir son propre code pour un même traitement, il est possible que des différences apparaissent (distraction, mauvaise communication, erreur ou autre). Par contre, si chaque application appelle la même procédure stockée, ce risque disparaît.

#### Inconvénients

Les procédures stockées **ajoutent évidemment à la charge sur le serveur de données**. Plus on implémente de logique de traitement directement dans la base de données, moins le serveur est disponible pour son but premier : le stockage de données.

Par ailleurs, certains traitements seront toujours plus simples et plus courts à écrire (et donc à maintenir) s'ils sont développés dans un langage informatique adapté. A fortiori lorsqu'il s'agit de traitements complexes. **La logique qu'il est possible d'implémenter avec MySQL permet de nombreuses choses, mais reste assez basique.**

Enfin, **la syntaxe des procédures stockées diffère beaucoup d'un SGBD à un autre**. Par conséquent, si l'on désire en changer, il faudra procéder à un grand nombre de corrections et d'ajustements.

### Conclusion et usage

Comme souvent, tout est question d'**équilibre**. Il faut savoir utiliser des procédures quand c'est utile, quand on a une bonne raison de le faire. Il ne sert à rien d'en abuser. Pour une base contenant des données ultrasensibles, une bonne gestion des droits des utilisateurs couplée à l'usage de procédures stockées peut se révéler salutaire. Pour une base de données destinée à être utilisée par plusieurs applications différentes, on choisira de créer des procédures pour les traitements généraux et/ou pour lesquels la moindre erreur peut poser de gros problèmes. Pour un traitement long, impliquant de nombreuses requêtes et une logique simple, on peut sérieusement gagner en performance en le faisant dans une procédure stockée (a fortiori si ce traitement est souvent lancé).

À vous de voir quelles procédures sont utiles pour **votre application et vos besoins**.

---

### En résumé

- Une procédure stockée est un **ensemble d'instructions** que l'on peut exécuter sur commande.
- Une procédure stockée est un objet de la base de données **stocké de manière durable**, au même titre qu'une table. Elle n'est pas supprimée à la fin de la session comme l'est une requête préparée.
- On peut passer des **paramètres** à une procédure stockée, qui peuvent avoir trois sens : IN (entrant), OUT (sortant) ou INOUT (les deux).
- SELECT ... INTO permet d'assigner des données sélectionnées à des variables ou des paramètres, à condition que le SELECT ne renvoie qu'une seule ligne, et qu'il y ait autant de valeurs sélectionnées que de variables à assigner.
- Les procédures stockées peuvent permettre de **gagner en performance** en diminuant les allers-retours entre le client et le serveur. Elles peuvent également aider à **sécuriser une base de données** et à s'assurer que les traitements sensibles soient toujours exécutés de la même manière.
- Par contre, elle **ajoute à la charge du serveur** et sa syntaxe n'est **pas toujours portable** d'un SGBD à un autre.

**QUELQUES EXERCICES PRATIQUES**

**1. La bibliothèque – procédures et fonctions stockées**

- 1) Créer la BD « biblio » à partir du script fourni.
- 2) Ecrire une fonction qui calcule, pour un adhérent donné, le nombre de jours restant avant d'être en retard. Si l'adhérent n'a pas d'emprunts en cours, on renvoie NULL. Si l'adhérent est en retard, on renvoie un résultat négatif. On prendra en compte la possibilité d'avoir des emprunts avec des dureeMax différentes et des emprunts en cours avec des dates d'emprunt différentes. Dans ce cas, on renverra le nombre de jours restant le plus petit et le nombre de jours de retard le plus grand.
- 3) Utiliser cette fonction pour afficher la situation de tous les adhérents.
- 4) Ecrire une procédure qui permette de lister les emprunts d'un adhérent identifié par son numéro.
- 5) Ecrire une procédure qui affiche les exemplaires disponibles d'un titre.
- 6) Ecrire une procédure qui affiche les titres d'un auteur et le nombre d'exemplaires disponibles par titre.
- 7) Ecrire une procédure qui permette d'enregistrer un emprunt.
- 8) Modifier la table des emprunts : mettez la valeur par défaut de la durée max à 14.
- 9) Ecrire une nouvelle procédure qui enregistre un emprunt et gère tous les cas d'erreur (le livre n'existe pas, l'adhérent n'existe pas, le livre est déjà emprunté, l'adhérent emprunte déjà 3 livres, etc.). La procédure impose la date du jour comme date d'emprunt. La procédure renverra un numéro de code pour chaque erreur.
- 10) Rajouter le trigger qui permet de gérer l'attribut « emprunté », booléen permettant de savoir si un livre est emprunté ou pas (cf. TP précédent).
- 11) Vérifier que ce trigger fonctionne avec la procédure stockée de l'exercice précédent.
- 12) Créer la procédure stockée qui permette d'enregistrer un retour en gérant tous les cas d'erreur (le livre n'existe pas, l'adhérent n'existe pas, le livre n'est pas emprunté, etc.) et en imposant la date du jour comme date de retour. La procédure renverra un numéro de code pour chaque erreur et le nombre de jour de retard s'il y a lieu.

**2. Les chantiers – procédure stockée**

On souhaite enregistrer des bilans sur l'utilisation des voitures. On va conserver, pour chaque voiture, le nombre de visites, le nombre de passager et le nombre de kilomètres effectués par mois.

Créer une table qui permet d'enregistrer ces informations.

## **Partie 2 : Les fonctions et procédures stockées**

Cette table est mise à jour une fois par an. Ecrire une procédure qui permet de remplir cette table à partir des informations qui sont dans la base. La procédure permettra aussi de mettre à jour l'attribut « kilométrage » des véhicules (on lui ajoute les kilomètres parcourus à chaque visite). Enfin la procédure supprimera toutes les visites de l'année.

On utilisera préférentiellement un curseur.

Peut-on se passer d'un curseur ?

### **3. BD Ecoling – fonction stockée**

Charger la BD Ecoling.

Dans la BD Ecoling, écrire une fonction qui permet d'afficher, pour un examen donné, la moyenne des notes, la meilleure note et le ou les noms des élèves, la plus basse note et le ou les noms des élèves (group\_concat), pour chacune des épreuves.

### **4. Programmation classique – procédures stockées**

Ecrire un programme qui permet de résoudre une équation du second degré.

On écrira d'abord une procédure qui résout une équation du premier degré avec l'entête suivante : a (in), b(in), x(out), nbsol(out) ; avec nbsol = -1 dans le cas d'une infinité de solutions.

La procédure équa2 fera appel à la procédure équa1

On écrira une procédure qui sert de programme principal et qui gère l'interface utilisateur.

On testera le programme avec tous les cas possibles : 2 solutions, 1 solution double, 0 solution type équa2, 1 solution simple, 0 solution type équa1, une infinité de solutions.