

## I.5. 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.

### I.5.1. Définition des colonnes

#### I.5.1.1. 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".

### I.5.1.2. 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`).

- **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`.

### I.5.1.3. Récapitulatif

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

| Caractéristique   | Nom de la colonne     | Type                     | NULL ? |
|-------------------|-----------------------|--------------------------|--------|
| Espèce            | <i>espece</i>         | <code>VARCHAR(40)</code> | Non    |
| Sexe              | <i>sexe</i>           | <code>CHAR(1)</code>     | Oui    |
| Date de naissance | <i>date_naissance</i> | <code>DATETIME</code>    | Non    |
| Commentaires      | <i>commentaires</i>   | <code>TEXT</code>        | Oui    |
| Nom               | <i>nom</i>            | <code>VARCHAR(30)</code> | 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)`, ...

## I.5.2. 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.

### I.5.2.1. 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.

### I.5.2.2. 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 :

|   |   |
|---|---|
| 1 | 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.

### 1.5.2.3. 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.

### 1.5.3. 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**.

#### 1.5.3.0.1. 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).

### I.5.3.0.2. 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.

#### I.5.3.1. 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 :

```
1 ENGINE = moteur;
```

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

```
1 ENGINE = INNODB;
```

### I.5.4. 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    | —  |

#### I.5.4.1. 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.

#### I.5.4.1.1. Création de la table

```
1 CREATE TABLE [IF NOT EXISTS] Nom_table (  
2     colonne1 description_colonne1,  
3     [colonne2 description_colonne2,  
4     colonne3 description_colonne3,  
5     ...,]  
6     [PRIMARY KEY (colonne_clé_primaire)]  
7 )  
8 [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.

#### I.5.4.1.2. 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 :

```
1 nom VARCHAR(30),  
2 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 :

```
1 espece VARCHAR(40) NOT NULL,  
2 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.

```
1 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.

**Exemple :**

```
1  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).

### I.5.4.2. 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 :

```
1  CREATE TABLE Animal (  
2      id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,  
3      espece VARCHAR(40) NOT NULL,  
4      sexe CHAR(1),  
5      date_naissance DATETIME NOT NULL,  
6      nom VARCHAR(30),  
7      commentaires TEXT,  
8      PRIMARY KEY (id)  
9  )  
10 ENGINE=INNODB;
```



Je n'ai pas gardé la valeur par défaut pour le champ *espece*, car je trouve que ça n'a pas beaucoup de sens dans ce contexte. C'était juste un exemple pour vous montrer la syntaxe.

### I.5.4.3. 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.

```
1  SHOW TABLES;      -- liste les tables de la base de données  
2
```

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

### I.5.5. 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.

```
1 DROP TABLE Animal;
```

#### I.5.5.1. 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.



## I.6. 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.

### I.6.1. 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 :

```
1 ALTER TABLE nom_table ADD ... -- permet d'ajouter quelque chose
   (une colonne par exemple)
2
3 ALTER TABLE nom_table DROP ... -- permet de retirer quelque chose
4
5 ALTER TABLE nom_table CHANGE ...
6 ALTER TABLE nom_table MODIFY ... -- permettent de modifier une
   colonne
```

#### I.6.1.0.1. Créons une table pour faire joujou

Dans la seconde partie de ce tutoriel, nous devrons 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` :

```
1 CREATE TABLE Test_tuto (
2     id INT NOT NULL,
3     nom VARCHAR(10) NOT NULL,
4     PRIMARY KEY(id)
5 );
```

## I.6.2. Ajout et suppression d'une colonne

### I.6.2.1. Ajout

On utilise la syntaxe suivante :

```
1 ALTER TABLE nom_table
2 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 :

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

Un petit `DESCRIBE Test_tuto;` vous permettra de vérifier les changements apportés.

### I.6.2.2. Suppression

La syntaxe de `ALTER TABLE ... DROP ...` est très simple :

```
1 ALTER TABLE nom_table
2 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.

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

## I.6.3. Modification de colonne

### I.6.3.1. Changement du nom de la colonne

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

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

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

```
1 ALTER TABLE Test_tuto
2 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.

### I.6.3.2. Changement du type de données

Les mots-clés **CHANGE** et **MODIFY** peuvent être utilisés pour changer le type de donnée de la colonne, mais aussi changer la valeur par défaut ou ajouter/supprimer une propriété **AUTO\_INCREMENT**. Si vous utilisez **CHANGE**, vous pouvez, 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 :

```
1 ALTER TABLE nom_table
2 CHANGE ancien_nom nouveau_nom nouvelle_description;
3
4 ALTER TABLE nom_table
5 MODIFY nom_colonne nouvelle_description;
```

Des exemples pour illustrer :

```
1 ALTER TABLE Test_tuto
2 CHANGE prenom nom VARCHAR(30) NOT NULL; -- Changement du type +
   changement du nom
3
4 ALTER TABLE Test_tuto
5 CHANGE id id BIGINT NOT NULL; -- Changement du type sans renommer
6
```

```
7 ALTER TABLE Test_tuto
8 MODIFY id BIGINT NOT NULL AUTO_INCREMENT; -- Ajout de
    l'auto-incrémentation
9
10 ALTER TABLE Test_tuto
11 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](#) .

---

### I.6.3.3. 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)

## I.7. 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 pourrons tester toutes sortes de ~~tortures~~ requêtes dans la suite de ce tutoriel.



### I.7.1. 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.

#### I.7.1.1. 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).

```
1 INSERT INTO Animal
2 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 :

```
1 INSERT INTO Animal
2 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).

```
1 INSERT INTO Animal
2 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 :

```
1 SELECT * FROM Animal;
```

Deux choses importantes à retenir ici.

- *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 ?

### 1.7.1.2. 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.

## I. MySQL et les bases du langage SQL

- 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 :

```
1 INSERT INTO Animal (espece, sexe, date_naissance)
2   VALUES ('tortue', 'F', '2009-08-03 05:12:00');
3 INSERT INTO Animal (nom, commentaires, date_naissance, espece)
4   VALUES ('Choupi', 'Né sans oreille gauche',
5           '2010-10-03 16:44:00', 'chat');
6 INSERT INTO Animal (espece, date_naissance, commentaires, nom,
7   sexe)
8   VALUES ('tortue', '2009-06-13 08:17:00', 'Carapace bizarre',
9           'Bobosse', 'F');
```

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

### I.7.1.3. Insertion multiple

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

```
1 INSERT INTO Animal (espece, sexe, date_naissance, nom)
2   VALUES ('chien', 'F', '2008-12-06 05:18:00', 'Caroline'),
3           ('chat', 'M', '2008-09-11 15:38:00', 'Bagherra'),
4           ('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 !

### I.7.2. Syntaxe alternative de MySQL

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

```
1 INSERT INTO Animal
2   SET nom='Bobo', espece='chien', sexe='M',
3       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.

### 1.7.3. 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.

#### 1.7.3.1. 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 :

```
1 SOURCE monFichier.sql;
```



Ou

```
1 \. 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.

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

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

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

### 1.7.3.2. 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 :

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 :

```
1 LOAD DATA [LOCAL] INFILE 'nom_fichier'
2 INTO TABLE nom_table
3 [FIELDS
4   [TERMINATED BY '\t']
5   [ENCLOSED BY '']
6   [ESCAPED BY '\\'] ]
7 ]
8 [LINES
9   [STARTING BY '']
10  [TERMINATED BY '\n']
11 ]
```

```
12 [IGNORE nombre LINES]
13 [(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 (si si 🍌). 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, 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 auto-incrémentée), *nom*, *prenom*, *date\_naissance* et *adresse* (qui peut être **NULL**).

|  |  |
|--|--|
|  |  |
|--|--|

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* :

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

## I.7.4. 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,

### I.7.4.1. 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).

👁 Contenu masqué n°1

Vous n'avez alors qu'à taper :

```
1 SOURCE remplissageAnimal.sql;
```

### I.7.4.2. 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`.

👁 Contenu masqué n°2



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

Exécutez ensuite la commande suivante :

```
1 LOAD DATA LOCAL INFILE 'animal.csv'  
2 INTO TABLE Animal  
3 FIELDS TERMINATED BY ';' ENCLOSED BY '"'  
4 LINES TERMINATED BY '\n' -- ou '\r\n' selon l'ordinateur et le  
   programme utilisés pour créer le fichier  
5 (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*.

```
1 SELECT * FROM Animal;
```

Nous pouvons maintenant passer au chapitre suivant !

---

### I.7.4.3. En résumé

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

```
1 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).

## Contenu masqué

### Contenu masqué n°1

|    |   |
|----|---|
| 1  | <b>INSERT INTO</b> Animal (espece, sexe, date_naissance, nom, commentaires) <b>VALUES</b> |
| 2  | ('chien', 'F', '2008-02-20 15:45:00' , 'Canaille', <b>NULL</b> ),                         |
| 3  | ('chien', 'F', '2009-05-26 08:54:00' , 'Cali', <b>NULL</b> ),                             |
| 4  | ('chien', 'F', '2007-04-24 12:54:00' , 'Rouquine', <b>NULL</b> ),                         |
| 5  | ('chien', 'F', '2009-05-26 08:56:00' , 'Fila', <b>NULL</b> ),                             |
| 6  | ('chien', 'F', '2008-02-20 15:47:00' , 'Anya', <b>NULL</b> ),                             |
| 7  | ('chien', 'F', '2009-05-26 08:50:00' , 'Louya' , <b>NULL</b> ),                           |
| 8  | ('chien', 'F', '2008-03-10 13:45:00', 'Welva' , <b>NULL</b> ),                            |
| 9  | ('chien', 'F', '2007-04-24 12:59:00' , 'Zira' , <b>NULL</b> ),                            |
| 10 | ('chien', 'F', '2009-05-26 09:02:00', 'Java' , <b>NULL</b> ),                             |
| 11 | ('chien', 'M', '2007-04-24 12:45:00' , 'Balou' , <b>NULL</b> ),                           |
| 12 | ('chien', 'M', '2008-03-10 13:43:00' , 'Pataud' , <b>NULL</b> ),                          |
| 13 | ('chien', 'M', '2007-04-24 12:42:00' , 'Bouli', <b>NULL</b> ),                            |
| 14 | ('chien', 'M', '2009-03-05 13:54:00', 'Zoulou' , <b>NULL</b> ),                           |
| 15 | ('chien', 'M', '2007-04-12 05:23:00' , 'Cartouche' , <b>NULL</b> ),                       |
| 16 | ('chien', 'M', '2006-05-14 15:50:00', 'Zambo', <b>NULL</b> ),                             |
| 17 | ('chien', 'M', '2006-05-14 15:48:00' , 'Samba' , <b>NULL</b> ),                           |
| 18 | ('chien', 'M', '2008-03-10 13:40:00', 'Moka' , <b>NULL</b> ),                             |
| 19 | ('chien', 'M', '2006-05-14 15:40:00', 'Pilou' , <b>NULL</b> ),                            |
| 20 | ('chat', 'M', '2009-05-14 06:30:00' , 'Fiero', <b>NULL</b> ),                             |
| 21 | ('chat', 'M', '2007-03-12 12:05:00' , 'Zonko', <b>NULL</b> ),                             |
| 22 | ('chat', 'M', '2008-02-20 15:45:00' , 'Filou', <b>NULL</b> ),                             |
| 23 | ('chat', 'M', '2007-03-12 12:07:00' , 'Farceur', <b>NULL</b> ),                           |
| 24 | ('chat', 'M', '2006-05-19 16:17:00' , 'Caribou' , <b>NULL</b> ),                          |
| 25 | ('chat', 'M', '2008-04-20 03:22:00' , 'Capou', <b>NULL</b> ),                             |
| 26 | ('chat', 'M', '2006-05-19 16:56:00' , 'Raccou',<br>'Pas de queue depuis la naissance');   |

[Retourner au texte.](#)

### Contenu masqué n°2

|  |  |
|--|--|
|  |  |
|--|--|

[Retourner au texte.](#)

## I.8. 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).

Motivés ? Alors c'est parti !!! 🍊

### I.8.1. 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

```
1 SELECT 'Hello World !';  
2 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.

```
1 SELECT colonne1, colonne2, ...  
2 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 :

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

### 1.8.1.1. Sélectionner toutes les colonnes

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

```
1 SELECT *
2 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.

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.

## 1.8.2. 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 :

```
1 SELECT *
2 FROM Animal
3 WHERE espece='chien';
```



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

### 1.8.2.1. Les opérateurs de comparaison

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

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

Exemples :

```
1 SELECT *
2 FROM Animal
3 WHERE date_naissance < '2008-01-01'; -- Animaux nés avant 2008
4
5 SELECT *
6 FROM Animal
7 WHERE espece <> 'chat'; -- Tous les animaux sauf les chats
```

### 1.8.2.2. 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 :

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

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



#### I.8.2.2.1. 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 :

```
1 SELECT *
2 FROM Animal
3 WHERE espece='chat'
4     AND sexe='F';
5 -- OU
6 SELECT *
7 FROM Animal
8 WHERE espece='chat'
9     && sexe='F';
```

#### I.8.2.2.2. OR

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

```
1 SELECT *
2 FROM Animal
3 WHERE espece='tortue'
4     OR espece='perroquet';
5 -- OU
6 SELECT *
7 FROM Animal
8 WHERE espece='tortue'
9     || 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).

#### I.8.2.2.3. NOT

Sélection de tous les animaux femelles sauf les chiennes.

```
1 SELECT *
2 FROM Animal
3 WHERE sexe='F'
4     AND NOT espece='chien';
5 -- OU
6 SELECT *
7 FROM Animal
8 WHERE sexe='F'
9     AND ! espece='chien';
```

#### I.8.2.2.4. XOR

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

```
1 SELECT *
2 FROM Animal
3 WHERE sexe='M'
4     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.

#### I.8.2.3. 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 !

### I.8.2.3.1. 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.

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.

Et voilà, vous avez bien défini les différents niveaux, il n'y a plus qu'à écrire la requête avec les bons opérateurs logiques !

👁 Contenu masqué n°3

### I.8.2.4. Le cas de NULL

Vous vous souvenez sans doute de la liste des opérateurs de comparaison que je vous ai présentée (sinon, retournez au début de la partie sur la clause **WHERE**). Vous avez probablement été un peu étonnés de voir dans cette liste l'opérateur **<=>** : égal (valable aussi pour **NULL**). D'autant plus que j'ai fait un peu semblant de rien et ne vous ai pas donné d'explication sur cette mystérieuse précision "aussi valable pour **NULL**" 🍊. Mais je vais me rattraper maintenant ! En fait, c'est très simple, le marqueur **NULL** (qui représente donc "pas de valeur") est un peu particulier. En effet, vous ne pouvez pas tester directement **colonne = NULL**. Essayons donc :

```
1 SELECT *
2 FROM Animal
3 WHERE nom = NULL; -- sélection des animaux sans nom
4
5 SELECT *
6 FROM Animal
```

## I. MySQL et les bases du langage SQL

```
7 WHERE commentaires <> NULL; -- sélection des animaux pour lesquels
  un commentaire existe
```

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

```
1 SELECT *
2 FROM Animal
3 WHERE nom <=> NULL; -- sélection des animaux sans nom
4 -- OU
5 SELECT *
6 FROM Animal
7 WHERE nom IS NULL;
8
9 SELECT *
10 FROM Animal
11 WHERE commentaires IS NOT NULL; -- sélection des animaux pour
    lesquels un commentaire existe
```

Cette fois-ci, ça fonctionne parfaitement !

| id | espece | sexe | date_nais-<br>sance    | nom  | commentaires |
|----|--------|------|------------------------|------|--------------|
| 4  | tortue | F    | 2009-08-03<br>05:12:00 | NULL | NULL         |
| 9  | tortue | NULL | 2010-08-23<br>05:18:00 | NULL | NULL         |

| id | espece | sexe | date_nais-<br>sance    | nom     | commentaires                             |
|----|--------|------|------------------------|---------|--|
| 1  | chien  | M    | 2010-04-05<br>13:43:00 | Rox     | Mordille<br>beaucoup                     |
| 5  | chat   | NULL | 2010-10-03<br>16:44:00 | Choupi  | Né sans oreille<br>gauche                |
| 6  | tortue | F    | 2009-06-13<br>08:17:00 | Bobosse | Carapace<br>bizarre                      |
| 35 | chat   | M    | 2006-05-19<br>16:56:00 | Raccou  | Pas de queue<br>depuis la nais-<br>sance |
| 52 | tortue | F    | 2006-03-15<br>14:26:00 | Redbul  | Insomniaque                              |
| 55 | tortue | M    | 2008-03-15<br>18:45:00 | Relou   | Surpoids                                 |

### I.8.3. Tri des données

Lorsque vous faites un **SELECT**, les données sont récupérées dans un ordre défini par MySQL, mais qui n'a aucun sens pour vous. Vous avez sans doute l'impression que MySQL renvoie tout simplement les lignes dans l'ordre dans lequel elles ont été insérées, mais ce n'est pas exactement le cas. En effet, si vous supprimez des lignes, puis en ajoutez de nouvelles, les nouvelles lignes viendront remplacer les anciennes dans l'ordre de MySQL. Or, bien souvent, vous voudrez trier à votre manière. Par date de naissance par exemple, ou bien par espèce, ou par sexe, etc.

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

Par exemple, pour trier par date de naissance :

```
1 SELECT *
2 FROM Animal
3 WHERE espece='chien'
4 ORDER BY date_naissance;
```

Et hop ! Vos données sont triées, les plus vieux chiens sont récupérés en premier, les jeunes à la fin.

#### I.8.3.1. Tri ascendant ou descendant

Tout ça c'est bien beau, j'ai mes chiens triés du plus vieux au plus jeune. Et si je veux le contraire ? Pour déterminer le sens du tri effectué, SQL possède deux mots-clés : **ASC** pour ascendant, et **DESC** pour descendant. Par défaut, si vous ne précisez rien, c'est un tri ascendant qui est effectué : du plus petit nombre au plus grand, de la date la plus ancienne à la plus récente, et pour les chaînes de caractères et les textes, c'est l'ordre alphabétique normal qui est utilisé. Si par contre vous utilisez le mot **DESC**, l'ordre est inversé : plus grand nombre d'abord, date la plus récente d'abord, et ordre anti-alphabétique pour les caractères.



Petit cas particulier : les **ENUM** sont des chaînes de caractères, mais sont triés selon l'ordre dans lequel les possibilités ont été définies. Si par exemple on définit une colonne **exemple** **ENUM('a', 'd', 'c', 'b')**, l'ordre **ASC** sera 'a', 'd', 'c' puis 'b' et l'ordre **DESC** 'b', 'c', 'd' suivi de 'a'.

```
1 SELECT *
2 FROM Animal
3 WHERE espece='chien'
4       AND nom IS NOT NULL
5 ORDER BY nom DESC;
```

### 1.8.3.2. Trier sur plusieurs colonnes

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

```
1 SELECT *
2 FROM Animal
3 ORDER BY espece, date_naissance;
```



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

Vous pouvez trier sur autant de colonnes que vous voulez.

### 1.8.4. Éliminer les doublons

Il peut arriver que MySQL vous donne plusieurs fois le même résultat. Non pas parce que MySQL fait des bêtises, mais tout simplement parce que certaines informations sont présentes plusieurs fois dans la table.

Petit exemple très parlant : vous voulez savoir quelles sont les espèces que vous possédez dans votre élevage. Facile, une petite requête :

```
1 SELECT espece
2 FROM Animal;
```

En effet, vous allez bien récupérer toutes les espèces que vous possédez, mais si vous avez 500 chiens, vous allez récupérer 500 lignes 'chien'. Un peu embêtant lorsque la table devient bien remplie.

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

```
1 SELECT DISTINCT espece
2 FROM Animal;
```

Ceci devrait gentiment vous ramener quatre lignes avec les quatre espèces qui se trouvent dans la table. C'est quand même plus clair non ?

Attention cependant, pour éliminer un doublon, il faut que toute la ligne **sélectionnée** soit égale à une autre ligne du jeu de résultats. Ça peut paraître logique, mais cela en perd plus d'un.

Ne seront donc prises en compte que les colonnes que vous avez précisées dans votre **SELECT**. Uniquement *espece* donc, dans notre exemple.

## 1.8.5. Restreindre les résultats

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

### 1.8.5.1. Syntaxe

**LIMIT** s'utilise avec deux paramètres.

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

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

Exemple :

```
1 SELECT *
2 FROM Animal
3 ORDER BY id
4 LIMIT 6 OFFSET 0;
5
6 SELECT *
7 FROM Animal
8 ORDER BY id
9 LIMIT 6 OFFSET 3;
```

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

| id | espece | sexe | date_nais-<br>sance    | nom            | commentaires            |
|----|--------|------|------------------------|----------------|-------------------------|
| 1  | chien  | M    | 2010-04-05<br>13:43:00 | Rox            | Mordille<br>beaucoup    |
| 2  | chat   | NULL | 2010-03-24<br>02:23:00 | Roucky         | NULL                    |
| 3  | chat   | F    | 2010-09-13<br>15:02:00 | Schtroumpfette | NULL                    |
| 4  | tortue | F    | 2009-08-03<br>05:12:00 | NULL           | NULL                    |
| 5  | chat   | NULL | 2010-10-03<br>16:44:00 | Choupi         | Né sans oreil<br>gauche |

## I. MySQL et les bases du langage SQL

|   |        |   |                        |         |                     |
|---|--------|---|------------------------|---------|---------------------|
| 6 | tortue | F | 2009-06-13<br>08:17:00 | Bobosse | Carapace<br>bizarre |
|---|--------|---|------------------------|---------|---------------------|

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

| id | espece | sexe | date_nais-<br>sance    | nom      | commentaires              |
|----|--------|------|------------------------|----------|---------------------------|
| 4  | tortue | F    | 2009-08-03<br>05:12:00 | NULL     | NULL                      |
| 5  | chat   | NULL | 2010-10-03<br>16:44:00 | Choupi   | Né sans oreille<br>gauche |
| 6  | tortue | F    | 2009-06-13<br>08:17:00 | Bobosse  | Carapace<br>bizarre       |
| 7  | chien  | F    | 2008-12-06<br>05:18:00 | Caroline | NULL                      |
| 8  | chat   | M    | 2008-09-11<br>15:38:00 | Bagherra | NULL                      |
| 9  | tortue | NULL | 2010-08-23<br>05:18:00 | NULL     | NULL                      |

Exemple avec un seul paramètre :

```
1 SELECT *
2 FROM Animal
3 ORDER BY id
4 LIMIT 10;
```

Cette requête est donc équivalente à :

```
1 SELECT *
2 FROM Animal
3 ORDER BY id
4 LIMIT 10 OFFSET 0;
```

### I.8.5.2. Syntaxe alternative

MySQL accepte une autre syntaxe pour la clause **LIMIT**. Ce n'est cependant pas la norme SQL donc idéalement vous devriez toujours utiliser la syntaxe officielle. Vous vous apercevrez toutefois que cette syntaxe est énormément usitée, je ne pouvais donc pas ne pas la mentionner

```
1 SELECT *
2 FROM Animal
3 ORDER BY id
```



```
4 LIMIT [decalage, ]nombre_de_lignes;
```

Tout comme pour la syntaxe officielle, le décalage n'est pas obligatoire, et vaudra 0 par défaut. Si vous le précisez, n'oubliez pas la virgule entre le décalage et le nombre de lignes désirées.

---

### I.8.5.3. En résumé

- La commande **SELECT** permet d'afficher des données.
- La clause **WHERE** permet de préciser des critères de sélection.
- Il est possible de trier les données grâce à **ORDER BY**, selon un ordre ascendant (**ASC**) ou descendant (**DESC**).
- Pour éliminer les doublons, on utilise le mot-clé **DISTINCT**, juste après **SELECT**.
- **LIMIT nb\_lignes OFFSET decalage** permet de sélectionner uniquement *nb\_lignes* de résultats, avec un certain décalage.

## Contenu masqué

### Contenu masqué n°3

```
1 SELECT *
2 FROM Animal
3 WHERE date_naissance > '2009-12-31'
4     OR
5     ( espece='chat'
6         AND
7         ( sexe='M'
8             OR
9             ( sexe='F' AND date_naissance < '2007-06-01' )
10        )
11    );
```

[Retourner au texte.](#)

## I.9. Élargir les possibilités de la clause WHERE

Dans le chapitre précédent, vous avez découvert la commande `SELECT`, ainsi que plusieurs clauses permettant de restreindre et d'ordonner les résultats selon différents critères. Nous allons maintenant revenir plus particulièrement sur la clause `WHERE`. Jusqu'ici, les conditions permises par `WHERE` étaient très basiques. Mais cette clause offre bien d'autres possibilités parmi lesquelles :

- la comparaison avec une valeur incomplète (chercher les animaux dont le nom commence par une certaine lettre par exemple) ;
- la comparaison avec un intervalle de valeurs (entre 2 et 5 par exemple) ;
- la comparaison avec un ensemble de valeurs (comparaison avec 5, 6, 10 ou 12 par exemple).

### I.9.1. Recherche approximative

Pour l'instant, nous avons vu huit opérateurs de comparaison :

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

À l'exception de `<=>` qui est un peu particulier, ce sont les opérateurs classiques, que vous retrouverez dans tous les langages informatiques. Cependant, il arrive que ces opérateurs ne soient pas suffisants. En particulier pour des recherches sur des chaînes de caractères. En effet, comment faire lorsqu'on ne sait pas si le mot que l'on recherche est au singulier ou au pluriel par exemple ? Ou si l'on cherche toutes les lignes dont le champ "commentaires" contient un mot particulier ?

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

## I. MySQL et les bases du langage SQL

Deux jokers existent pour `LIKE` :

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

Quelques exemples :

- `'b%'` cherchera toutes les chaînes de caractères commençant par `'b'` (`'brocoli'`, `'bouli'`, `'b'`)
- `'B_'` cherchera toutes les chaînes de caractères contenant une ou deux lettres dont la première est `'b'` (`'ba'`, `'bf'`, `'b'`)
- `'%ch%ne'` cherchera toutes les chaînes de caractères contenant `'ch'` et finissant par `'ne'` (`'chne'`, `'chine'`, `'échine'`, `'le pays le plus peuplé du monde est la Chine'`)
- `'_ch_ne'` cherchera toutes les chaînes de caractères commençant par `'ch'`, éventuellement précédées d'une seule lettre, suivies de zéro ou d'un caractère au choix et enfin se terminant par `'ne'` (`'chine'`, `'chne'`, `'echine'`)

### I.9.1.0.1. Rechercher `'%'` ou `'_'`

Comment faire si vous cherchez une chaîne de caractères contenant `'%'` ou `'_'` ? Évidemment, si vous écrivez `LIKE '%'` ou `'LIKE '`, MySQL vous donnera absolument toutes les chaînes de caractères dans le premier cas, et toutes les chaînes de 0 ou 1 caractère dans le deuxième. Il faut donc signaler à MySQL que vous ne désirez pas utiliser `||%` ou `||_||` en tant que joker, mais bien en tant que caractère de recherche. Pour ça, il suffit de mettre le caractère d'échappement `||\||`, dont je vous ai déjà parlé, devant le `'%'` ou le `'_'`.

Exemple :

```
1 SELECT *
2 FROM Animal
3 WHERE commentaires LIKE '%\%%';
```

Résultat :

| id | espece | sexe | date_nais-<br>sance    | nom   | commentaires                         |
|----|--------|------|------------------------|-------|--------------------------------------|
| 42 | chat   | F    | 2008-04-20<br>03:20:00 | Bilba | Sourde de<br>l'oreille droite<br>80% |

### I.9.1.0.2. Exclure une chaîne de caractères

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

```
1 SELECT *
2 FROM Animal
3 WHERE nom NOT LIKE '%a%';
```

#### I.9.1.1. Sensibilité à la casse

Vous l'aurez peut-être remarqué en faisant des essais, `LIKE 'chaîne de caractères'` n'est pas sensible à la casse (donc aux différences majuscules-minuscules). Pour rappel, ceci est dû à l'interclassement. Nous avons gardé l'interclassement par défaut du jeu de caractère UTF-8, qui n'est pas sensible à la casse. Si vous désirez faire une recherche sensible à la casse, vous pouvez définir votre chaîne de recherche comme une chaîne de type binaire, et non plus comme une simple chaîne de caractères :

```
1 SELECT *
2 FROM Animal
3 WHERE nom LIKE '%Lu%'; -- insensible à la casse
4
5 SELECT *
6 FROM Animal
7 WHERE nom LIKE BINARY '%Lu%'; -- sensible à la casse
```

#### I.9.1.2. Recherche dans les numériques

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

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

```
1 SELECT *
2 FROM Animal
3 WHERE id LIKE '1%';
```

## I.9.2. Recherche dans un intervalle

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

```
1 SELECT *
2 FROM Animal
3 WHERE date_naissance <= '2009-03-23'
4       AND date_naissance >= '2008-01-05';
```

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

```
1 SELECT *
2 FROM Animal
3 WHERE date_naissance BETWEEN '2008-01-05' AND '2009-03-23';
```

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

## I.9.3. Set de critères

Le dernier opérateur à utiliser dans la clause **WHERE** que nous verrons dans ce chapitre est **IN**. Ce petit mot de deux lettres, bien souvent méconnu des débutants, va probablement vous permettre d'économiser du temps et des lignes.

Imaginons que vous vouliez récupérer les informations des animaux répondant aux doux noms de Moka, Bilba, Tortilla, Balou, Dana, Redbul et Gingko. Jusqu'à maintenant, vous auriez sans doute fait quelque chose comme ça :

```
1 SELECT *
2 FROM Animal
3 WHERE nom = 'Moka'
4       OR nom = 'Bilba'
5       OR nom = 'Tortilla'
6       OR nom = 'Balou'
```

```
7 OR nom = 'Dana'
8 OR nom = 'Redbul'
9 OR nom = 'Gingko';
```

Un peu fastidieux non 🍋 ? Eh bien réjouissez-vous, car **IN** est dans la place ! Cet opérateur vous permet de faire des recherches parmi une liste de valeurs. Parfait pour nous donc, qui voulons rechercher les animaux correspondant à une liste de noms. Voici la manière d'utiliser **IN** :

```
1 SELECT *
2 FROM Animal
3 WHERE nom IN ('Moka', 'Bilba', 'Tortilla', 'Balou', 'Dana',
               'Redbul', 'Gingko');
```

C'est quand même plus agréable à écrire ! 🍋

---

### I.9.3.1. En résumé

- L'opérateur **LIKE** permet de faire des recherches approximatives, grâce aux deux caractères "joker" : '%' (qui représente 0 ou plusieurs caractères) et '\_' (qui représente 0 ou 1 caractère).
- L'opérateur **BETWEEN** permet de faire une recherche sur un intervalle. **WHERE colonne BETWEEN a AND b** étant équivalent à **WHERE colonne >= a AND colonne <= b**.
- Enfin, l'opérateur **IN** permet de faire une recherche sur une liste de valeurs.

## I.10. Suppression et modification de données

Vous savez comment insérer des données, vous savez comment les sélectionner et les ordonner selon les critères de votre choix, il est temps maintenant d'apprendre à les supprimer et les modifier ! Avant cela, un petit détour par le client *mysqldump*, qui vous permet de sauvegarder vos bases de données. Je ne voudrais en effet pas vous lâcher dans le chapitre de suppression de données sans que vous n'ayez la possibilité de faire un backup de votre base. Je vous connais, vous allez faire des bêtises, et vous direz encore que c'est de ma faute... 🍌

### I.10.1. Sauvegarde d'une base de données

Il est bien utile de pouvoir sauvegarder facilement sa base de données, et très important de la sauvegarder régulièrement. Une mauvaise manipulation (ou un méchant pirate 🍌 s'il s'agit d'un site web) et toutes les données peuvent disparaître. MySQL dispose donc d'un outil spécialement dédié à la sauvegarde des données sous forme de fichiers texte : *mysqldump*.

Cette fonction de sauvegarde s'utilise à partir de la console. Vous devez donc être déconnectés de MySQL pour la lancer. Si c'est votre cas, tapez simplement `exit`

Vous êtes maintenant dans la console Windows (ou Mac, ou Linux). La manière classique de faire une sauvegarde d'une base de données est de taper la commande suivante :

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

Décortiquons cette commande.

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

## I. MySQL et les bases du langage SQL

Lancez la commande suivante pour sauvegarder *elevage* dans votre dossier courant (c'est-à-dire le dossier dans lequel vous étiez au moment de la connexion) :

```
1 mysqldump -u sdz -p --opt elevage > elevage_sauvegarde.sql
```

Puis, allez voir dans le dossier. Vous devriez y trouver un fichier `elevage_sauvegarde.sql`. Ouvrez-le avec un éditeur de texte. Vous pouvez voir nombre de commandes SQL qui servent à la création des tables de la base de données, ainsi qu'à l'insertion des données. S'ajoutent à cela quelques commandes qui vont sélectionner le bon encodage, etc.



Vous pouvez bien entendu sauvegarder votre fichier dans un autre dossier que celui où vous êtes au moment de lancer la commande. Il suffit pour cela de préciser le chemin vers le dossier désiré. Ex : `C:\\"Mes Documents"\mysql\sauvegardes\elevage_sauvegarde.sql` au lieu de `elevage_sauvegarde.sql`

La base de données est donc sauvegardée. Notez que la commande pour créer la base elle-même n'est pas sauvegardée. Donc, si vous effacez votre base par mégarde, il vous faut d'abord recréer la base de données (avec `CREATE DATABASE nom_base`), puis exécuter la commande suivante (dans la console) :

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

Concrètement, dans notre cas :

```
1 mysql elevage < elevage_sauvegarde.sql
```

Ou, directement à partir de MySQL :


```
1 USE nom_base;  
2 source fichier_de_sauvegarde.sql;
```

Donc :

```
1 USE elevage;  
2 source elevage_sauvegarde.sql;
```





Vous savez maintenant sauvegarder de manière simple vos bases de données. Notez que je ne vous ai donné ici qu'une manière d'utiliser `mysqldump`. En effet, cette commande possède de nombreuses options. Si cela vous intéresse, je vous renvoie à la [documentation de MySQL](#)  qui sera toujours plus complète que moi.

## I.10.2. Suppression

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

```
1 DELETE FROM nom_table
2 WHERE critères;
```

Par exemple : Zoulou est mort, paix à son âme 🍌 ... Nous allons donc le retirer de la base de données.

```
1 DELETE FROM Animal
2 WHERE nom = 'Zoulou';
```

Et voilà, plus de Zoulou 🍌 !

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

```
1 DELETE FROM Animal;
```



Attention, je le répète, cette opération est **irréversible**. Soyez toujours bien sûrs d'avoir sous la main une sauvegarde de votre base de données au cas où vous regretteriez votre geste (on ne pourra pas dire que je ne vous ai pas prévenus).

## I.10.3. Modification

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

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

Par exemple, vous étiez persuadés que ce petit Pataud était un mâle mais, quelques semaines plus tard, vous vous rendez compte de votre erreur. Il vous faut donc modifier son sexe, mais aussi son nom. Voici la requête qui va vous le permettre :

```
1 UPDATE Animal
2 SET sexe='F', nom='Pataude'
3 WHERE id=21;
```

Vérifiez d'abord chez vous que l'animal portant le numéro d'identification 21 est bien Pataud. J'utilise ici la clé primaire (donc *id*) pour identifier la ligne à modifier, car c'est la seule manière d'être sûr que je ne modifierai que la ligne que je désire. En effet, il est possible que plusieurs animaux aient pour nom "Pataud". Ce n'est a priori pas notre cas, mais prenons tout de suite de bonnes habitudes.



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

La requête suivante changerait donc le commentaire de tous les animaux stockés dans la table *Animal* (ne l'exécutez pas).

```
1 UPDATE Animal
2 SET commentaires='modification de toutes les lignes';
```

---

### I.10.3.1. En résumé

- Le client *mysqldump* est un programme qui permet de sauvegarder facilement ses bases de données.
- La commande `DELETE` permet de supprimer des données.
- La commande `UPDATE` permet de modifier des données.