Résumé SQL partie 1

Creation d'un utilisateur:

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

GRANT ALL PRIVILEGES : permet d'attribuer tous les droits .
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), pour toutes les tables de cette base (grâce à *).
TO 'BADZI' : 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.

Creation d'une base de donnée :

```
CREATE DATABASE name CARACTER SET 'utf8';
```

Connection :

```
Mysql –u user –p database
```

Suppression :

```
DROP DATABASE IF EXISTS name;
```

Afficher les erreurs :

```
SHOW WARNINGS;
```

Utiliser une base de données :

```
USE name ;
```

Pour utiliser moteur de son choix :

```
ENGINE = moteur ; ex : INNODB, MYISAM…
```

Création de table :

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

Création de table :

```
CREATE TABLE Animal (
    id SMALLINT NOT NULL AUTO_INCREMENT,
    espece VARCHAR(40) NOT NULL,
    sexe ENUM('male','femelle'),
    date_naissance DATETIME NOT NULL,
    nom VARCHAR(30),
    commentaires TEXT,
    PRIMARY KEY (id)
)
ENGINE=INNODB;
```

Voir toutes les tables d'une db :

```
SHOW TABLES;
```

Voir la structure d'une table :

```
DESCRIBE Animal;
```

Suppression d'une table :

```
DROP TABLE Animal;
```

Renommer une table :

```
ALTER TABLE ancien_nom
RENAME nouveau_nom ;
```

Modification d'une table :

```
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
```

Ajout et suppression de colonnes

Ajout :

```
ALTER TABLE nom_table
ADD [COLUMN] nom_colonne description_colonne;
```

Suppression :

```
ALTER TABLE nom_table
DROP [COLUMN] nom_colonne;
```

Modification de colonnes

Changement du nom de la colonne :

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

Changement de type de données :

```
ALTER TABLE nom_table
MODIFY nom_colonne nouvelle_description;
```

Insertion de données :

Insertion sans preciser les colonnes :

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

2eme exemple :

```
INSERT INTO Animal
VALUES (2, 'chat', NULL, '2010-03-24 02:23:00',
'Roucky', NULL);
```

Afficher les requêtes d'une table :

```
SELECT * FROM nom_table;
```

Si on met **NULL** pour id elle s'auto-incremente car definie **NOT NULL** et **AUTO_INCREMENT**

Insertion en précisant les colonnes :

- 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. Fini donc les **NULL** lorsqu'on n'a pas de valeur à mettre.

Exemple :

```
INSERT INTO Animal (espece, sexe, date_naissance)
VALUES ('tortue', 'femelle', '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', 'femelle');
```

Insertion multiple :

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

Syntaxe alternative MYSQL (propre à MYSQL) :

```
INSERT INTO Animal
SET nom='Bobo', espece='chien', sexe='male',
date_naissance='2010-07-21 15:41:00';
```

Executer des commandes SQL à partir d'un fichier :

```
SOURCE monFichier.sql;
Ou
\. monFichier.sql;
Ou
SOURCE sous-dossier\monFichier.sql;
Execute les fichiers SQL dans le repertoire courant
Sinon
SOURCE C:\Document and Settings\dossierX\monFichier.sql;
```

LOAD DATA IN FILE :

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

Exemple :

```
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);
```

Sélection de données :

Sélection de certaines colonnes :

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

Sélection de toutes les colonnes d'une table :

```
SELECT *
FROM nom_table;
```

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 :

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

Les opérateurs de comparaison :

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

'=' égale, '<' inférieur, '<=' inférieur ou égale,
'>' supérieur, '>=' supérieur ou égal,
'<> ou !=' différent, '⇔'égale(valable pour **NULL** aussi)

Combinaison des critères :

AND, OR , XOR, NOT **ou** &&, ||, XOR, !

Cas de plusieurs opérateurs :

Usage de parenthèse indispensable
Exemple : (rouge AND vert) OR bleu

Tester la valeur NULL :

```
SELECT *
FROM Animal
WHERE nom <=> NULL; -- sélection des animaux
sans nom
-- OU
SELECT *
FROM Animal
WHERE nom IS NULL;

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

Tri des données :

```
SELECT *
FROM Animal
WHERE espece='chien'
ORDER BY date_naissance[DESC ou ASC];
```

Trier sur plusieurs colonnes :

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.

Eliminer les doublons :

```
SELECT DISTINCT espece
FROM Animal;
```

Restreindre des résultats :

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

Exemple :

```
SELECT *
FROM Animal
ORDER BY id
LIMIT 6 OFFSET 3;\\=> id :4,5,6,7,8,9
```

Recherche approximative : (LIKE)

'%' en SQL ⇔ '*' en linux
'_' en SQL ⇔ '?' en linux

Exemple :

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

Exclure une chaine de caractère :

Exemple :

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

LIKE : insensible à la casse.
LIKE BINARY : Sensible à la casse.

Recherche dans un intervalle :

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

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

SET de critères :

```
SELECT *
FROM Animal
WHERE nom = 'Moka'
   OR nom = 'Bilba'
   OR nom = 'Tortilla'
   OR nom = 'Balou'
   OR nom = 'Dana'
   OR nom = 'Redbul'
   OR nom = 'Gingko' ;
```

⇔

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

Suppression et modification des données :

Sauvegarde d'une base de donnée :

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

Reccupération en cas de suppression de DATABASE :

```
Mysql -u root -p nomDB < sauvegarde
```

`mysqldump` : il s'agit donc 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'effectuera 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.

Suppression :

```
DELETE FROM nom_table  //Irreversible
WHERE critères;
```

Modification :

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

Exemple:

```
UPDATE Animal
SET sexe='femelle', nom='Pataude'
WHERE id=21;
```

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

Union de plusieurs requêtes :

Syntaxe:

```
SELECT ...
UNION
SELECT ...
UNION
SELECT ...
....
UNION
SELECT ...
```

Exemple :

```
SELECT Animal.*
FROM Animal
INNER JOIN Espece
ON Animal.espece_id = Espece.id
WHERE Espece.nom_courant = 'Chat'
UNION
SELECT Animal.*
FROM Animal
INNER JOIN Espece
ON Animal.espece_id = Espece.id
WHERE Espece.nom_courant = 'Tortue d''Hermann';
```

Résumé SQL partie 2

Création et suppression des index

Ajout des index lors de la création des tables :

Index dans la description de la colonne :

Seuls les index "classiques" et uniques peuvent être créés de cette manière.

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

Lorsque vous mettez un index **UNIQUE** sur une table, vous ne mettez pas seulement un index, vous ajoutez surtout une **contrainte.**

Liste d'INDEX :

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

Exemple :

```
CREATE TABLE Animal (
id SMALLINT NOT NULL AUTO_INCREMENT,
espece VARCHAR(40) NOT NULL,
sexe ENUM('male','femelle'),
date_naissance DATETIME NOT NULL,
nom VARCHAR(30),
commmentaires TEXT,
PRIMARY KEY (id),
INDEX ind_date_naissance (date_naissance), -- index sur la date de naissance
INDEX ind_nom_espece (nom(10), espece) -- index sur le nom (le chiffre entre parenthèses étant le nombre de caractères pris en compte) et l'espèce
)
ENGINE=INNODB;
```

Autre exemple :

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

Ajout des index après la création de la table :

Ajout avec ALTER TABLE :

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

Ajout avec CREATE INDEX :

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

Ajout de UNIQUE après création de l'index :

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

Suppression d'index :

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

Recherche avec FULLTEXT :

un index FULLTEXT ne peut être défini que pour une table utilisant le moteur MyISAM ;
un index FULLTEXT ne peut être défini que sur une colonne de type CHAR, VARCHAR ou TEXT
les index "par la gauche" ne sont pas pris en compte par les index FULLTEXT

Il existe trois types de recherche FULLTEXT : la recherche naturelle, la recherche avec booléen, et enfin la recherche avec expansion de requête.

Recherche naturelle :

```
SELECT * -- Vous mettez évidemment les colonnes que vous voulez.
FROM nom_table
WHERE MATCH(colonne1[, colonne2, ...]) -- La ou les colonnes (index FULLTEXT correspondant nécessaire).
AGAINST ('chaîne recherchée'); -- La chaîne de caractère recherchée, entre guillemets bien sûr.
```

Recherche avec booléens :

```
SELECT *
FROM nom_table
WHERE MATCH(colonne)
AGAINST('chaîne recherchée' IN BOOLEAN MODE); -- IN BOOLEAN MODE à l'intérieur des parenthèses !
```

Recherche avec extension de requête :

```
SELECT *
FROM Livre
WHERE MATCH(titre, auteur)
AGAINST ('Daniel' WITH QUERY EXPANSION);
```

Clé primaire et retour :

Création d'une clé primaire :

Lors de la création de la table :

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

Exemple :

```
CREATE TABLE Animal (
id SMALLINT AUTO_INCREMENT PRIMARY KEY,
espece VARCHAR(40) NOT NULL,
sexe ENUM('male','femelle'),
date_naissance DATETIME NOT NULL,
nom VARCHAR(30),
commentaires TEXT
)E
NGINE=InnoDB;
Ou
CREATE TABLE Animal (
id SMALLINT AUTO_INCREMENT,
espece VARCHAR(40) NOT NULL,
sexe ENUM('male','femelle'),
date_naissance DATETIME NOT NULL,
nom VARCHAR(30),
commentaires TEXT,
[CONSTRAINT [symbole_contrainte]] PRIMARY KEY (id)
-- comme pour les index UNIQUE, CONSTRAINT est facultatif
)
ENGINE=InnoDB;
```

Après création de la table :

Si vous créez une clé primaire sur une table existante, assurez-vous que la (les) colonne(s) sur laquelle (lesquelles) vous voulez l'ajouter en contiennent pas **NULL**.

```
ALTER TABLE Animal
ADD [CONSTRAINT [symbole_contrainte]] PRIMARY KEY (id);
```

Suppression d'une clé primaire :

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

Création d'une clé etrangaire :

Lors de la création de la table :

```
CREATE TABLE [IF NOT EXISTS] Nom_table (
colonne1 description_colonne1,
[colonne2 description_colonne2,
colonne3 description_colonne3,
...,]
[ [CONSTRAINT [symbole_contrainte]] FOREIGN KEY
(colonne(s)_clé_secondaire) REFERENCES table_référence
(colonne(s)_référence)]
)[
ENGINE=moteur];
```

Après création de la table :

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

Suppression d'une clé etrangère :

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

Jointures :

Jointure interne

```
SELECT Especе.description

FROM Especе
INNER JOIN Animal
ON Especе.id = Animal.especе_id
WHERE Animal.nom = 'Cartouche';
```

⇔

```
SELECT espece_id FROM Animal WHERE nom = 'Cartouche';
SELECT description FROM Especе WHERE id = 1;
```

SELECT Especе.description : je sélectionne la colonne *description* de la table *Especе*.
FROM Especе : je travaille sur la table *Especе*.
INNER JOIN Animal : je la joints (avec une jointure interne) à la table *Animal*.
ON Especе.id = Animal.especе_id : la jointure se fait sur les colonnes *id* de la table *Especе* et *especе_id* de la table *Animal*, qui doivent donc correspondre.
WHERE Animal.nom = 'Cartouche' : dans la table résultant de la jointure, je sélectionne les lignes qui ont la valeur « Cartouche » dans la colonne *nom* venant de la table *Animal*.

Syntaxe :

```
SELECT * -- comme d'habitude, vous
sélectionnez les colonnes que vous voulez
FROM nom_table1
[INNER] JOIN nom_table2 – INNER explicite le fait
qu'il s'agit d'une jointure interne, mais c'est facultatif
ON colonne_table1 = colonne_table2 – sur quelles colonnes
se fait la jointure
-- vous pouvez mettre
colonne_table2 = colonne_table1, l'ordre n'a pas d'importance
[WHERE …]
[ORDER BY …] – les clauses habituelles
sont bien sûr utilisables !
[LIMIT …]
```

Les alias :

Pour renommer colonne

Exemple :

```
SELECT e.id,
       e.description,
       a.nom
FROM Especе AS e – On donne l'alias « e » à Especе
INNER JOIN Animal AS a – et l'alias « a » à Animal.
ON e.id = a.especе_id
WHERE a.nom LIKE 'C%';
```

Jointure externe :

Comme je viens de vous le dire, une jointure externe permet de sélectionner également les lignes pour lesquelles il n'y a pas de correspondance dans une des tables jointes. MySQL permet deux types de jointures externes : les jointures par la gauche, et les jointures par la droite.

Jointures par la gauche :

```
SELECT Animal.nom AS nom_animal, Race.nom AS race
FROM Animal – Table
de gauche
LEFT [OUTER] JOIN Race – Table
de droite
ON Animal.race_id = Race.id
WHERE Animal.nom LIKE 'C%'
ORDER BY Race.nom, Animal.nom;
```

Lorsque l'on fait une jointure par la gauche (grâce aux mots-clés **LEFT JOIN** ou **LEFT OUTER JOIN**), cela signifie que l'on veut toutes les lignes de la table de gauche (sauf restrictions dans une clause **WHERE** bien sûr), même si certaines n'ont pas de correspondance avec une ligne de la table de droite.

Jointures par la droite :

```
SELECT Animal.nom AS nom_animal, Race.nom AS race
FROM Animal – Table
de gauche
RIGHT JOIN Race – Table
de droite
ON Animal.race_id = Race.id
WHERE Race.espece_id = 2
ORDER BY Race.nom, Animal.nom;
```

Les jointures par la droite (**RIGHT JOIN** ou **RIGHT OUTER JOIN**), c'est évidemment le même principe, sauf que ce sont toutes les lignes de la table de droite qui sont sélectionnées même s'il n'y a pas de correspondance dans la table de gauche.

Jointures avec USING :

```
SELECT *
FROM Table1
[INNER | LEFT | RIGHT] JOIN Table2 USING (colonneJ);
-- equivalent à
SELECT *
FROM Table1
[INNER | LEFT | RIGHT] JOIN Table2 ON Table1.colonneJ =
Table2.colonneJ ;
```

Lorsque les colonnes qui servent à joindre les deux tables ont **le même nom**, vous pouvez utiliser la clause **USING** au lieu de la clause **ON**.

Si la jointure se fait sur plusieurs colonnes, il suffit de lister les colonnes en les séparant par des virgules : **USING** (colonne1, colonne2, …)

Jointures sans JOIN :

Ce n'est cependant possible que pour les jointures internes.

Syntaxe plutôt ambigüe donc déconseillé

```
SELECT *
FROM Table1, Table2
WHERE Table1.colonne1 = Table2.colonne2;
-- équivalent à
SELECT *
FROM Table1
[INNER] JOIN Table2
ON Table1.colonne1 = Table2.colonne2 ;
```

Conditions avec ANY, SOME et ALL :

Les conditions avec **IN** et **NOT IN** sont un peu limitées, puisqu'elles ne permettent que des comparaisons de type « est égal » ou « est différent ». Avec **ANY** et **ALL**, on va pouvoir utiliser les autres comparateurs (« plus grand/petit que », etc.).

Uniquement avec requête

ANY : veut dire « au moins une des valeurs ».
SOME : est un synonyme de **ANY**.
ALL : signifie « toutes les valeurs ».

Sous requite corrélées :

Une sous-requête corrélée est une sous-requête qui fait référence à une colonne (ou une table) qui n'est pas définie dans sa clause **FROM**, mais bien ailleurs dans la requête dont elle fait partie.

Exemple :

```
SELECT nom_courant
FROM Espece
WHERE id = ANY (
SELECT id
FROM Animal
WHERE Animal.espece_id = Espece.id
AND race_id IS NOT NULL
) ;
```

Sous requête pour l'insertion :

```
INSERT INTO nom_table
[(colonne1, colonne2, …)]
SELECT [colonne1, colonne2, …]
FROM nom_table2 ;
```

Sous requête pour modification :

Exemple :

```
UPDATE Animal
SET race_id = ( SELECT id
FROM Race
WHERE nom = 'Nebelung'
AND espece_id = 2
)
WHERE nom = 'Cawette' ;
```

Une limitation importante des sous-requête est qu'on ne peut pas modifier un élément d'une table que l'on utilise dans une sous-requête.

Sous requête pour suppression :

```
DELETE
FROM Animal
WHERE nom = 'Carabistouille'
AND espece_id = (
SELECT id
FROM Espece
WHERE nom_courant = 'Chat'
) ;
```

Les limitations sur **DELETE** sont les mêmes que pour **UPDATE** : on ne peut supprimer de lignes d'une table qui est utilisée dans une sous-requête.

Suppression avec jointures :

Exemple :

```
DELETE Animal
FROM Animal
INNER JOIN Espece
ON Animal.espece_id = Espece.id
WHERE Animal.nom = 'Carabistouille'
AND Espece.nom_courant = 'Chat';
```

Les règles :

Nombre des colonnes :

Il est absolument indispensable que toutes les requêtes unies renvoient le même nombre de colonnes.

Type et ordre des colonnes :

Pour la cohérence doit être compatible.

UNIO ALL :

Chaque résultat n'apparait qu'une seule fois. Pour la simple et bonne raison que lorsque vous faites **UNION**, les doublons sont effacés. En fait, **UNION** est équivalent à **UNION DISTINCT**. Si vous voulez conserver les doublons, il vous faut utiliser

Exemple :

```
SELECT * FROM Espece
UNION ALL
SELECT * FROM Espece ;
```

UNION ALL.

LIMIT et ORDER BY :

LIMIT :

Exemple 1:

```
SELECT id, nom
FROM Race
LIMIT 3
UNION
SELECT id, nom_latin
FROM Espece ;
```

Vous avez bien trois noms de races, suivi de toutes les espèces.

Exemple 2 :

```
SELECT id, nom
FROM Race
UNION
SELECT id, nom_latin
FROM Espece
LIMIT 2 ;
```

Visiblement, ce n'est pas ce que nous voulions… En fait, **LIMIT** a été appliqué à l'ensemble des résultats, après **UNION**. Par conséquent, si l'on veut que **LIMIT** ne porte que sur la dernière requête, il faut le préciser. Pour ça, il suffit d'utiliser des parenthèses.

Exemple 3 :

```
SELECT id, nom
FROM Race
UNION
(SELECT id, nom_latin
FROM Espece
LIMIT 2) ;
```

Et voilà.

ORDER BY :

```
SELECT id, nom
FROM Race
UNION
SELECT id, nom_latin
FROM Espece
ORDER BY nom DESC;
```

Il faut bien mettre ici **ORDER BY** nom, et surtout pas **ORDER BY** Race.nom ou **ORDER BY** Especе.nom_latin. En effet, l'**ORDER BY** agit sur l'ensemble de la requête, donc en quelque sorte, sur une table intermédiaire composée des résultats des deux requêtes unies. Cette table n'est pas nommée, et ne possède que deux colonnes : *id* et *nom* (définies par la première clause **SELECT** rencontrée).

Vous pouvez bien sûr combiner **LIMIT** et **ORDER BY**.

```
(SELECT id, nom
FROM Race
LIMIT 6)
UNION
(SELECT id, nom_latin
FROM Espece
LIMIT 3)
ORDER BY nom
LIMIT 5;
```

Option sur suppression d'une clé etrangère :

```
ALTER TABLE nom_table
ADD [CONSTRAINT fk_col_ref]
FOREIGN KEY (colonne)
REFERENCES table_ref(col_ref)
ON DELETE {RESTRICT | NO ACTION | SET NULL | CASCADE}; -- <--Nouvelle option !
```

Exemple :

```
ALTER TABLE Animal
ADD CONSTRAINT fk_race_id
FOREIGN KEY (race_id)
REFERENCES Race(id)
ON DELETE SET NULL;
```

RESTRICT :

C'est le comportement par défaut. Si l'on essaye de supprimer une valeur référencée par une clé étrangère, l'action est avortée et on obtient une erreur.

NO ACTION :

C'est sans doute le comportement que vous utiliserez le moins, puisque ça enlève tout simplement la sécurité. Vous ne pourrez toujours pas insérer des données qui n'existent pas dans la table de référence, mais vous n'aurez aucune difficulté à supprimer une référence.
Par conséquent, vous risquez de vous retrouver avec des incohérences dans votre base.

SET NULL :

Si on choisi **SET NULL**, alors tout simplement, **NULL** est substitué aux valeurs dont la référence est supprimée.
Pour reprendre notre exemple, en supprimant la race des Boxers, tous les animaux auxquels on a attribué cette race verront la valeur de leur *race_id* passer à **NULL**;

CASCADE :

Ce dernier comportement est aussi le plus risqué (et le plus violent !). En effet, cela supprime purement et simplement toutes les lignes qui référençaient la valeur supprimée !
Il y a cependant de nombreuses situations dans lesquelles c'est utile. Prenez par exemple un forum sur un site internet. Vous avez une table *Sujet*, et une table *Message*, avec une colonne *sujet_id* . Avec **ON DELETE CASCADE**, il vous suffit de supprimer un sujet pour que tous les messages de ce sujets soient également supprimés.

Option sur modification d'une clé etrangère :

Et l'option permettant de définir le comportement en cas de modification est donc **ON UPDATE** {**RESTRICT** | **NO** ACTION | **SET NULL** | **CASCADE**}. Les quatre comportement possibles sont exactement les mêmes que pour la suppression.
RESTRICT : empêche la modification si elle casse la contrainte (comportement par défaut).
NO ACTION : permet la modification dans tous les cas.
SET NULL : mets **NULL** partout où la valeur modifiée était référencée.
CASCADE : modifie également la valeur là où elle est référencée.

Exemple :

```
ALTER TABLE Animal
ADD CONSTRAINT fk_race_id
FOREIGN KEY (race_id)
REFERENCES Race(id)
ON DELETE SET NULL -- N'oublions pas de remettre le ON DELETE !
ON UPDATE CASCADE;
```

Ignorer les erreurs :

Insertion :

Nous avons mis une contrainte d'unicité (sous la forme d'un index **UNIQUE**) sur la colonne *nom_latin* de la table *Espece*. Donc, si l'on essaye d'insérer la ligne suivante, une erreur sera déclenchée puisqu'il existe déjà une espèce dont le nom latin est '*Canis canis*'.

```
INSERT INTO Espece (nom_courant, nom_latin, description)
VALUES ('Chien en peluche', 'Canis canis', 'Tout doux, propre et silencieux');
```

donne : ERROR 1062 (23000): Duplicate entry 'Canis canis' for key 'nom_latin'

par contre :

```
INSERT IGNORE INTO Espece (nom_courant, nom_latin, description)
VALUES ('Chien en peluche', 'Canis canis', 'Tout doux, propre et silencieux');
```

donne : Query OK, 0 rows affected (0.01 sec)

Modification :

Si l'on essaye de modifier l'espèce des chats, pour lui donner comme nom latin *Canis canis*, une erreur sera déclenchée, sauf si l'on ajoute l'option **IGNORE**.

```
UPDATE Espece
SET nom_latin = 'Canis canis'
WHERE nom_courant = 'Chat';
```

donne : ERROR 1062 (23000): Duplicate entry 'Canis canis' for key 'nom_latin'

```
UPDATE IGNORE Espece
SET nom_latin = 'Canis canis'
WHERE nom_courant = 'Chat';
```

donne : Query OK, 0 rows affected (0.01 sec)

LOAD DATA INFILE :

Syntaxe :

```
LOAD DATA [LOCAL] INFILE 'nom_fichier' IGNORE -- IGNORE se place juste avant INTO, comme dans INSERT
```

Remplacer l'ancienne ligne :

Lorsque vous voulez insérer une ligne dans une table, vous pouvez utiliser la commande bien connue **INSERT INTO**, ou vous pouvez utiliser **REPLACE INTO**. La différence entre ces deux requêtes est la

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

façon qu'elles ont de gérer les contraintes d'unicité
(ça tombe bien, c'est le sujet de ce chapitre !).
Dans le cas d'une insertion qui enfreint une contrainte d'unicité, **REPLACE** ne va ni renvoyer une erreur, ni ignorer l'insertion
(comme **INSERT INTO** [**IGNORE**]). **REPLACE** va purement, simplement et violemment remplacer l'ancienne ligne par la
nouvelle.

Exemple :

```
REPLACE INTO Animal (sexe, nom, date_naissance, espece_id)
VALUES ('femelle', 'Spoutnik', '2010-08-06 15:05:00', 3);
```

Supprime et insert à nouveau la ligne.

Remplacer avec LOAD DATA INFILE :

```
LOAD DATA [LOCAL] INFILE 'nom_fichier' REPLACE -- se place au
même endroit que IGNORE
INTO TABLE nom_table
[FIELDS
[TERMINATED BY '\t']
[ENCLOSED BY '']
[ESCAPED BY '\\' ]
][
LINES
[STARTING BY '']
[TERMINATED BY '\n']
][
IGNORE nombre LINES]
[(nom_colonne,...)];
```

Modifier la ligne :

```
INSERT INTO nom_table [(colonne1, colonne2, colonne3)]
VALUES (valeur1, valeur2, valeur3)
ON DUPLICATE KEY UPDATE colonne2 = valeur2 [, colonne3 = valeur3];
```