Administrez vos bases de données avec MySQL

Par Taguan

www.siteduzero.com

Dernière mise à jour le 27/02/2012

Sommaire

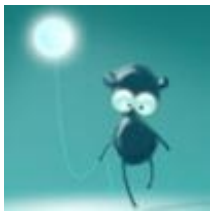
Sommaire 1
Partager 3
Administrez vos bases de données avec MySQL 5
Quelques exemples d'applications 5
Points abordés dans ce tutoriel 5
Partie 1 : MySQL et les bases du langage SQL 7
Introduction 7
Concepts de base 7
Base de données 7
SGBD 7
SGBDR 8
Le langage SQL 9
Présentation succincte de MySQL... 9
Un peu d'histoire 9
Mise en garde 9
... et de ses concurrents 10
Oracle database 10
PostgreSQL 10
MS Access 10
SQLite 10
Organisation d'une base de données 10
Installation de MySQL 11
Avant-propos 12
Ligne de commande 12
Interface graphique 12
Pourquoi utiliser la ligne de commande ? 13
Installation du logiciel 13
Windows 13
Mac OS 15
Linux 16
Connexion à MySQL 16
Connexion au client 16
Déconnexion 18
Syntaxe SQL et premières commandes 18
"Hello World !" 18
Syntaxe 18
Un peu de math 22
Encodage 22
Utilisateur 23
Les types de données 25
Types numériques 25
Nombres entiers 25
Nombres décimaux 26
Types alphanumériques 27
Chaînes de type texte 27
Chaînes de type binaire 28
SET et ENUM 28
Types temporels 31
DATE, TIME et DATETIME 31
YEAR 33
TIMESTAMP 33
La date par défaut 33
Création d'une base de données 33
Avant-propos : conseils et conventions 34
Conseils 34
Conventions 34
Mise en situation 35
Dossier de travail 35
Création et suppression d'une base de données 36
Création 36
Suppression 36
Utilisation d'une base de données 37
Création de tables 38
Définition des colonnes 38
Type de colonne 38
NULL or not NULL ? 38
Récapitulatif 38
Introduction aux clés primaires 39
Identité 39
Clé primaire 39
Auto-incrémentation 39
Les moteurs de tables 40
Préciser un moteur lors de la création de la table 40
Syntaxe de CREATE TABLE 40

Syntaxe ... 41
Application : création de "Animal" ... 42
Vérifications ... 42
Suppression d'une table ... 43
Modification d'une table ... 44
Syntaxe de la requête ... 44
Ajout et suppression d'une colonne ... 44
Ajout ... 44
Suppression ... 45
Modification de colonne ... 45
Changement du nom de la colonne ... 45
Changement du type de données ... 45
Insertion de données ... 46
Syntaxe de INSERT ... 47
Insertion sans préciser les colonnes ... 47
Insertion en précisant les colonnes ... 48
Insertion multiple ... 49
Syntaxe alternative de MySQL ... 49
Utilisation de fichiers externes ... 50
Exécuter des commandes SQL à partir d'un fichier ... 50
Insérer des données à partir d'un fichier formaté ... 51
Remplissage de la base ... 52
Exécution de commandes SQL ... 52
LOAD DATA INFILE ... 53
Sélection de données ... 55
Syntaxe de SELECT ... 55
Sélectionner toutes les colonnes ... 55
La clause WHERE ... 56
Les opérateurs de comparaison ... 56
Combinaisons de critères ... 57
Sélection complexe ... 58
Le cas de NULL ... 59
Tri des données ... 60
Tri ascendant ou descendant ... 61
Trier sur plusieurs colonnes ... 61
Éliminer les doublons ... 62
Restreindre les résultats ... 62
Exemple ... 62
Exemple avec un seul paramètre ... 63
Syntaxe alternative ... 64
Élargir les possibilités de la clause WHERE ... 64
Recherche approximative ... 65
Sensibilité à la casse ... 66
Recherche dans les numériques ... 66
Recherche dans un intervalle ... 67
Set de critères ... 67
Suppression et modification de données ... 68
Sauvegarde d'une base de données ... 69
Suppression ... 70
Modification ... 71
Partie 2 : Index, jointures et sous-requêtes ... 72
Index ... 72
Qu'est-ce qu'un index ? ... 72
Intérêt des index ... 73
Désavantages ... 73
Index sur plusieurs colonnes ... 73
Index sur des colonnes de type alphanumérique ... 75
Les différents types d'index ... 76
Index UNIQUE ... 77
Index FULLTEXT ... 77
Création et suppression des index ... 77
Ajout des index lors de la création de la table ... 77
Ajout des index après création de la table ... 79
Complément pour la création d'un index UNIQUE - le cas des contraintes ... 80
Suppression d'un index ... 81
Recherches avec FULLTEXT ... 81
Comment fonctionne la recherche FULLTEXT ? ... 82
Les types de recherche ... 82
Clés primaires et étrangères ... 89
Clés primaires, le retour ... 90
Choix de la clé primaire ... 90
Création d'une clé primaire ... 91
Suppression de la clé primaire ... 92
Clés étrangères ... 92
Création ... 93
Suppression d'une clé étrangère ... 94
Modification de notre base ... 95
La table Espece ... 95
La table Animal ... 96
Jointures ... 100
Principe des jointures et notion d'alias ... 100

Principe des jointures ... 100
Notion d'alias ... 102
Jointure interne ... 103
Syntaxe ... 104
Pourquoi "interne" ? ... 106
Jointure externe ... 107
Jointures par la gauche ... 107
Jointures par la droite ... 108
Syntaxes alternatives ... 110
Jointures avec USING ... 110
Jointures sans JOIN ... 110
Exemples d'application et exercices ... 110
A/ Commençons par du facile ... 111
B/ Compliquons un peu les choses ... 113
C/ Et maintenant, le test ultime ! ... 115
Sous-requêtes ... 119
Sous-requêtes dans le FROM ... 119
Les règles à respecter ... 120
Sous-requêtes dans les conditions ... 121
Comparaisons ... 121
Conditions avec IN et NOT IN ... 125
Conditions avec ANY, SOME et ALL ... 126
Sous-requêtes corrélées ... 127
Jointures et sous-requêtes pour l'insertion, la modification et la suppression de données ... 132
Insertion ... 132
Sous-requête pour l'insertion ... 132
Modification ... 134
Utilisation des sous-requêtes ... 134
Modification avec jointure ... 136
Suppression ... 137
Utilisation des sous-requêtes ... 137
Suppression avec jointure ... 137
Union de plusieurs requêtes ... 138
Syntaxe ... 139
Les règles ... 139
UNION ALL ... 142
LIMIT et ORDER BY ... 143
LIMIT ... 143
ORDER BY ... 145
Options des clés étrangères ... 147
Option sur suppression des clés étrangères ... 148
Petit rappel ... 148
Suppression d'une référence ... 148
Option sur modification des clés étrangères ... 150
Utilisation de ces options dans notre base ... 151
Modifications ... 152
Suppressions ... 152
Les requêtes ... 153
Violation de contrainte d'unicité ... 153
Ignorer les erreurs ... 154
Insertion ... 154
Modification ... 155
LOAD DATA INFILE ... 155
Remplacer l'ancienne ligne ... 156
Remplacement de plusieurs lignes ... 157
LOAD DATA INFILE ... 158
Modifier l'ancienne ligne ... 158
Syntaxe ... 158
Exemple ... 158
Attention : plusieurs contraintes d'unicité sur la même table ... 159
Partie 3 : Fonctions : nombres, chaînes et agrégats ... 160
Rappels et introduction ... 161
Rappels et manipulation simple de nombres ... 161
Rappels ... 161
Combiner les données avec des opérations mathématiques ... 163
Définition d'une fonction ... 164
Fonctions scalaires VS fonctions d'agrégation ... 166
Quelques fonctions générales ... 167
Informations sur l'environnement actuel ... 167
Informations sur la dernière requête ... 168
Convertir le type de données ... 170
Fonctions scalaires ... 171
Manipulation de nombres ... 172
Arrondis ... 172
Exposants et racines ... 174
Hasard ... 175
Divers ... 175
Manipulation de chaînes de caractères ... 176
Longueur et comparaison ... 176
Retrait et ajout de caractères ... 177
Recherche et remplacement ... 180

Concaténation ... 182
FIELD(), une fonction bien utile pour le tri ... 183
Code ASCII ... 183
Exemples d'application et exercices ... 184
On commence par du facile ... 184
Puis on corse un peu ... 185
Fonctions d'agrégation ... 186
Fonctions statistiques ... 187
Nombre de lignes ... 187
Minimum et maximum ... 188
Somme et moyenne ... 189
Concaténation ... 190
Principe ... 190
Syntaxe ... 190
Exemples ... 191
Regroupement ... 192
Regroupement sur un critère ... 192
Voir d'autres colonnes ... 193
La règle SQL ... 193
Le cas MySQL ... 194
Tri des données ... 195
Et les autres espèces ? ... 196
Regroupement sur plusieurs critères ... 197
Super-agrégats ... 199
Conditions sur les fonctions d'agrégation ... 201
Optimisation ... 202
Exercices sur les agrégats ... 203
Du simple... ... 203
1. Combien de races avons-nous dans la table Race ? ... 203
2. De combien de chiens connaissons-nous le père ? ... 203
3. Quelle est la date de naissance de notre plus jeune femelle ? ... 203
4. En moyenne, combien de pattes ont nos animaux nés avant le premier janvier 2010 ? ... 203
5. Combien avons-nous de perroquets mâles et femelles, et quels sont leur nom (en une seule requête bien sûr) ? ... 204
...Vers le complexe ... 204
1. Quelles sont les races dont nous ne possédons aucun individu ? ... 204
2. Quelles sont les espèces (triées par ordre alphabétique du nom latin) dont nous possédons moins de cinq mâles ? ... 204
3. Combien de mâles et de femelles avons-nous de chaque race, avec un compte total intermédiaire pour les races (mâles et femelles confon 205
4. Combien de pattes, par espèce et au total, aurons-nous si nous prenons Parlotte, Spoutnik, Caribou, Cartouche, Cali, Canaille, Yoda, Zamb 205
Partie 4 : Fonctions : manipuler les dates ... 207
Obtenir la date/l'heure actuelle ... 207
Rappels ... 207
Date ... 207
Heure ... 207
Date et heure ... 207
Timestamp ... 207
Année ... 208
Date actuelle ... 208
Heure actuelle ... 209
Date et heure actuelles ... 209
Les fonctions ... 209
Qui peut le plus, peut le moins ... 210
Timestamp unix ... 211
Formater une donnée temporelle ... 211
Extraire une information précise ... 212
Informations sur la date ... 212
Informations sur l'heure ... 215
Formater une date facilement ... 216
Format ... 216
Exemples ... 217
Fonction supplémentaire pour l'heure ... 218
Formats standards ... 219
Créer une date à partir d'une chaîne de caractères ... 220
Calculs sur les données temporelles ... 222
Différence entre deux dates/heures ... 222
Ajout et retrait d'un intervalle de temps ... 223
Ajout d'un intervalle de temps ... 224
Soustraction d'un intervalle de temps ... 227
Divers ... 229
Créer une date/heure à partir d'autres informations ... 229
Convertir un TIME en secondes, et vice versa ... 230
Dernier jour du mois ... 231
Exercices ... 231
Commençons par le format ... 232
Passons aux calculs ... 235
Et pour finir, mélangeons le tout ... 237

Administrez vos bases de données avec MySQL

Par  Taguan

Mise à jour : 23/02/2012
Difficulté : Intermédiaire Durée d'étude : 1 mois, 15 jours

Vous avez de nombreuses données à traiter et vous voulez les organiser correctement, avec un outil adapté ?
Les bases de données ont été créées pour vous !

Ce tutoriel porte sur **MySQL**, qui est un Système de Gestion de Bases de Données Relationnelles (abrégé SGBDR). C'est-à-dire un logiciel qui permet de gérer des bases de données, et donc de gérer de grosses quantités d'informations. Il utilise pour cela le langage SQL.
Il s'agit d'un des SGBDR les plus connus et les plus utilisés (Wikipédia et Adobe utilisent par exemple MySQL). Et c'est certainement le SGBDR le plus utilisé à ce jour pour réaliser des sites web dynamiques. C'est d'ailleurs MySQL qui est présenté dans le tutoriel Concevez votre site web avec PHP et MySQL écrit par Mathieu Nebra, fondateur de ce site.

MySQL peut donc s'utiliser seul, mais est la plupart du temps combiné à un autre langage de programmation : PHP par exemple pour de nombreux sites web, mais aussi Java, Python, C++, et beaucoup, beaucoup d'autres.

Différentes façons d'utiliser MySQL

Quelques exemples d'applications

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

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

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

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

Points abordés dans ce tutoriel

La conception et l'utilisation de bases de données est un vaste sujet, il a fallu faire des choix sur les thèmes à aborder. Voici les

compétences que ce tutoriel vise à vous faire acquérir :

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

Partie 1 : MySQL et les bases du langage SQL

Dans cette partie, vous commencerez par apprendre quelques définitions indispensables, pour ensuite installer MySQL sur votre ordinateur.
Les commandes de base de MySQL seront alors expliquées (création de tables, insertion, sélection et modification de données, etc.)

Introduction

Avant de pouvoir joyeusement jouer avec des données, il vous faut connaître quelques concepts de base.

À la fin de ce chapitre, vous devriez :

- savoir ce qu'est un SGBD, un SGBDR, une base de données, et comment y sont représentées les données ;
- en connaître un peu plus sur MySQL et ses concurrents ;
- savoir ce qu'est le langage SQL et à quoi il sert.

Concepts de base
Base de données

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

Prenons l'exemple d'un site web avec un système de news et de membres. On va utiliser une base de données MySQL pour stocker toutes les données du site : les news (avec la date de publication, le titre, le contenu, éventuellement l'auteur, ...) et les membres (leur nom, leur email, ...).
Tout ceci va constituer notre base de données pour le site. Mais il ne suffit pas que la base de données existe. Il faut aussi pouvoir **la gérer, interagir avec cette base**. Il faut pouvoir envoyer des message à MySQL (messages qu'on appellera "**requêtes**"), afin de pouvoir ajouter des news, modifier des membres, supprimer, et tout simplement afficher des éléments de la base.

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

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

SGBD

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

Le paradigme client - serveur

La plupart de SGBD sont basés sur un **modèle Client - Serveur**. C'est à dire que la base de données se trouve sur un serveur qui ne sert qu'à ça, et pour interagir avec cette base de données, il faut utiliser un logiciel "client" qui va interroger le serveur, et transmettre la réponse que le serveur lui aura donnée. Le serveur peut être installé sur une machine différente du client, et c'est souvent le cas lorsque les bases de données sont importantes. Ce n'est cependant pas obligatoire, ne sautez pas sur votre petit frère pour lui emprunter son ordinateur. Dans ce tutoriel, nous installerons les logiciels serveur et client sur un seul et même ordinateur.
Par conséquent, lorsque vous installez un SGBD basé sur ce modèle (et c'est le cas de MySQL), vous installez en réalité deux choses (au moins) : le serveur, et le client. Et chaque requête (insertion/modification/lecture de données) est faite par l'intermédiaire du client. Jamais vous ne discuterez directement avec le serveur (d'ailleurs, il ne comprendrait rien à ce que vous diriez).
Et vous avez donc besoin d'un langage pour discuter avec le client, pour lui donner les requêtes que vous souhaitez effectuer. Dans le cas de MySQL, ce langage est le SQL.

SGBDR

Le R de SGBDR signifie "**relationnel**". Un SGBDR est un SGBD qui implémente la théorie relationnelle. MySQL impléménte la théorie relationnelle, et est donc un SGBDR.

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

Numéro	Nom	Prénom	Email

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

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

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

- la sélection (ou restriction) : obtenir les lignes de A répondant à certains critères ;
- la projection : obtenir une partie des attributs des lignes de A ;
- l'union - $A \cup B$: obtenir tout ce qui se trouve dans la relation A ou dans la relation B ;
- l'intersection - $A \cap B$: obtenir tout ce qui se trouve à la fois dans la relation A et dans la relation B ;
- la différence - $A - B$: obtenir ce qui se trouve dans la relation A mais pas dans la relation B ;
- la jointure - $A \bowtie B$: obtenir l'ensemble des lignes provenant de la liaison de la relation A et de la relation B à l'aide d'une information commune.

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

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

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

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

Le langage SQL

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

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

Présentation succincte de MySQL...

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

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

Un peu d'histoire

Le développement de MySQL commence en 1994 par David Axmark et Michael Widenius. EN 1995, la société MySQL AB est fondée par ces deux développeurs, et Allan Larsson. C'est la même année que sort la première version officielle de MySQL. En 2008, MySQL AB est rachetée par la société Sun Microsystems, qui est elle-même rachetée par Oracle Corporation en 2010.
On craint alors la fin de la gratuité de MySQL, étant donné que Oracle Corporation édite un des grands concurrents de MySQL : Oracle Database, qui lui est payant (et très cher). Oracle a cependant promis de continuer à développer MySQL et de conserver la double licence GPL (libre) et commerciale jusqu'en 2015 au moins.

David Axmark, fondateur de MySQL

Mise en garde

MySQL est très utilisé, surtout par les débutants. Vous pourrez faire de nombreuses choses avec ce logiciel, et il convient tout à

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

... et de ses concurrents

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

Oracle database

Oracle, édité par Oracle Corporation (qui, je rappelle, édite également MySQL) est un SGBDR payant. Son coût élevé fait qu'il est principalement utilisé par des entreprises.
Oracle gère très bien de grands volumes de données. Il est inutile d'acheter une licence oracle pour un projet de petite taille, car les performances ne seront pas bien différentes de celles de MySQL ou un autre SGBDR. Par contre, pour des projet conséquents (plusieurs centaines de Go de données), Oracle sera bien plus performant.
Par ailleurs, Oracle dispose d'un langage procédural très puissant (du moins plus puissant que le langage procédural de MySQL) : le PL/SQL.

PostgreSQL

Comme MySQL, PostgreSQL est un logiciel Open Source. Il est cependant moins utilisé, notamment par les débutants, car moins connu. La raison de cette méconnaissance réside sans doute en partie dans le fait que PostgreSQL a longtemps été disponible uniquement sous Unix. La première version windows n'est apparue qu'à la sortie de la version 8.0 du logiciel, en 2005.
PostgreSQL a longtemps été plus performant que MySQL, mais ces différences tendent à diminuer. MySQL semble être aujourd'hui équivalent à PostgreSQL en terme de performances sauf pour quelques opérations telles que l'insertion de données et la création d'index.
Le langage procédural utilisé par PostgreSQL s'appelle le PL/pgSQL.

MS Access

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

SQLite

La particularité de SQLite est de ne pas utiliser le schéma client-serveur utilisé par la majorité des SGBDR. SQLite stocke toutes les données dans de simples fichiers. Par conséquent, il ne faut pas installer de serveur de base de données, ce qui n'est pas toujours possible (certains hébergeurs web ne le permettent pas par exemple).
Pour de très petits volumes de données, SQLite est très performant. Cependant, le fait que les informations soient simplement stockées dans des fichiers rend le système difficile à sécuriser (autant au niveau des accès, qu'au niveau de la gestion de plusieurs utilisateurs utilisant la base simultanément).

Organisation d'une base de données

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

Toutes les factures ensemble, tous les contrats ensemble, etc. Ensuite on subdivise : les factures d'éléctricité, les factures pour la

voiture. Ou dans l'autre sens, tous les papiers concernant la voiture ensemble, puis subdivision en taxes, communication avec l'assureur, avec le garagiste, ...

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

Comme je vous l'ai dit plus haut, on représente les données sous forme de **tables**. Une base va donc contenir plusieurs tables (elle peut n'en contenir qu'une bien sûr, mais c'est rarement le cas). Si je reprends mon exemple précédent, on a donc une table représentant des clients (donc des personnes).
Chaque table définit un certain nombre de **colonnes**, qui sont les caractéristiques de l'objet représenté par la table (les attributs de l'en-tête dans la théorie relationelle). On a donc ici une colonne "nom", une colonne "prénom", une "email" et une colonne numéro qui nous permet d'identifier les clients individuellement (les nom et prénom ne suffisent pas toujours).

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

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

Installons donc tout ça sur l'ordinateur, et c'est parti !!! 

Installation de MySQL

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

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

Avant-propos

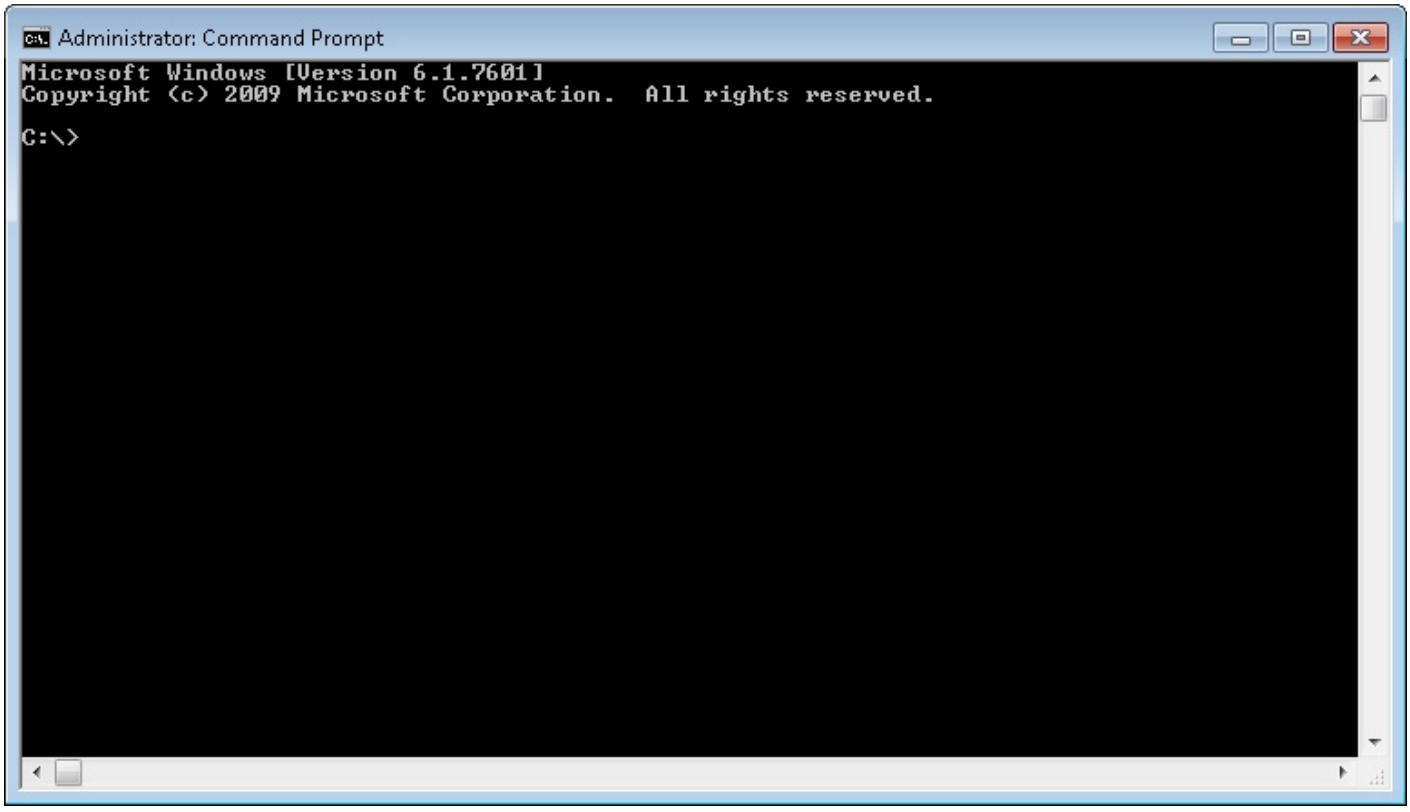
Il existe plusieurs manières d'utiliser MySQL. La première, que je vais utiliser tout au long du tutoriel, est l'utilisation en ligne de commande.

Ligne de commande

Mais qu'est-ce donc ? 

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

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



Interface graphique

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

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

Pourquoi utiliser la ligne de commande ?

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

Deux raisons :

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

Bien sûr, si vous voulez utiliser une interface graphique, je ne peux guère vous en empêcher. Mais je vous encourage vivement à commencer par utiliser la ligne de commande, ou au minimum de faire l'effort de décortiquer les requêtes que vous laisserez l'interface graphique construire pour vous. Ceci afin de pouvoir les écrire vous-même le jour où vous en aurez besoin (et ce jour viendra, je vous le prédis).

Installation du logiciel

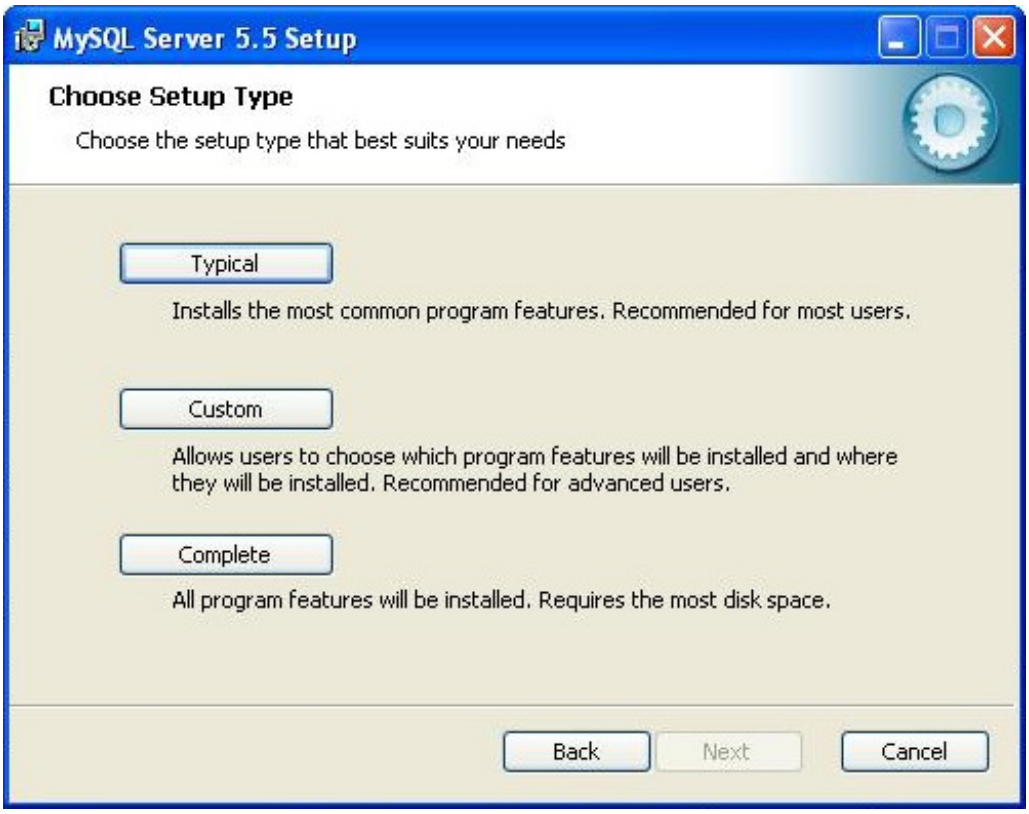
Pour télécharger MySQL, vous pouvez vous rendre sur le site suivant :

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

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

Windows

Téléchargez MySQL avec l'installeur (MSI Installer), puis exécutez le fichier téléchargé. L'installeur démarre et vous guide lors de l'installation.
Lorsqu'il vous demande de choisir entre trois types d'installation, choisissez "Typical". Cela installera tout ce dont nous pourrions avoir besoin.



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

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

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

Mac OS

Téléchargez l'archive DMG qui vous convient (32 ou 64 bit), double-cliquez ensuite sur ce .dmg pour ouvrir l'image disque. Vous devriez y trouver 4 fichiers dont deux .pkg. Celui qui nous intéresse s'appelle mysql-5.5.9-osx10.6-x86_64.pkg (les chiffres peuvent changer selon la version de MySQL téléchargée et votre ordinateur). Ouvrez ce fichier, qui est en fait l'installateur de MySQL, et suivez les instructions.

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

Tapez les commandes et exécutez les instructions suivantes :

Code : Console

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

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

Code : Console

```
bg
```

- Tapez Ctrl + D
- Quittez le terminal

MySQL est prêt à être utilisé !

Configuration

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

Code : Console

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

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

Code : Console

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

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

Linux

N'ayant pas d'installation Linux à portée de main, je ne peux malheureusement pas vous détailler la marche à suivre pour cette plateforme, mais vous devriez trouver de nombreuses informations sur internet en cas de problème et/ou question.

Connexion à MySQL

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

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

Connexion au client

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

Code : Console

```
mysql
```

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

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

L'hôte et l'utilisateur ont des valeurs par défaut, et ne sont donc pas toujours indispensables. La valeur par défaut de l'hôte est "localhost", ce qui signifie que le serveur est sur le même ordinateur que le client. C'est bien notre cas, donc nous n'aurons pas à préciser ce paramètre. Pour le nom d'utilisateur, la valeur par défaut dépend de votre système. Sous Windows, l'utilisateur courant est "ODBC", tandis que pour les systèmes Unix (Mac et Linux), il s'agit de votre nom d'utilisateur (le nom qui apparait dans l'invite de commande).
Pour votre première connexion à MySQL, il vous faudra vous connecter avec l'utilisateur "root", pour lequel vous avez normalement défini un mot de passe (si vous ne l'avez pas fait, inutile d'utiliser ce paramètre, mais ce n'est pas très sécurisé). Par la suite, nous créerons un nouvel utilisateur.
Pour chacun des trois paramètres, deux syntaxes sont possibles :

Code : Console

```
########
# Hôte #
########

--hote=nom_hote

# ou

-h nom_hote
```

```
########
# User #
########

--user=nom_utilisateur

# ou

-u nom_utilisateur

################
# Mot de passe #
################

--password=password

# ou

-ppassword
```

Remarquez l'absence d'espace entre `-p` et le mot de passe. C'est voulu (mais uniquement pour ce paramètre-là), et souvent source d'erreurs.
La commande complète pour se connecter est donc :

Code : Console

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

# ou

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

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

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

J'utiliserai uniquement les paramètres courts à partir de maintenant. Choisissez ce qui vous convient le mieux.
Notez que pour le mot de passe, il vous est possible (et c'est même très conseillé) de préciser uniquement que vous utilisez le paramètre, sans lui donner de valeur :

Code : Console

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

Apparaissent alors dans la console les mots suivants :

Code : Console

```
Enter password:
```

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

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

Code : Console

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

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

Déconnexion

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

Syntaxe SQL et premières commandes

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

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

"Hello World !"

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

Code : SQL

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

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

Code : Console

```
+-------------+
|Hello World !|
+-------------+
|Hello World !|
+-------------+
```

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

Syntaxe

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

Fin d'une instruction

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

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

Code : SQL

```
SELECT 100
```

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

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

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

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

Voyons donc ça !

Instructions sur plusieurs lignes

Tapez les commandes suivantes dans votre console, retour à la ligne compris.

Code : Console

```
SELECT 1,
'Hello World !',
'troisième ligne',
'quatrième',
'cinquième',
'sixième',
'bon, vous avez compris...'
;
```

C'est quand même plus clair que tout sur une ligne :

Code : Console

```
SELECT 1, 'Hello World !', 'troisième ligne', 'quatrième', 'cinquième', 'sixième',
```

N'hésitez pas à écrire vos commandes en plusieurs lignes, vous verrez beaucoup plus clair d'une part, et d'autre part, ça vous permettra de trouver vos erreurs plus facilement.

Démonstration

Code : Console

```
SELECT 1,
'Hello World !',,
'troisième ligne'
;
```

Résultat :

Code : Console

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that cor
'troisième ligne'' at line 2
```

MySQL me donne le numéro de la ligne à laquelle je me suis trompée (2), et me dit que c'est au niveau de 'troisième ligne'. Je regarde la ligne en question, et effectivement, une virgule m'avait échappé à la fin de la ligne 2 (juste avant 'troisième ligne' donc). Si vous mettez tout en une ligne, invariablement MySQL vous dira que l'erreur se situe sur la ligne 1. Forcément...
Notez que bien souvent, il vous faudra remonter à la ligne précédant celle signalée. En effet, si MySQL tombe sur quelque chose qui ne lui plaît pas en début de ligne sept, c'est peut-être parce que vous avez oublié ou ajouté quelque chose (une parenthèse, une virgule, un mot-clé) à la ligne six.

Plusieurs instructions sur une ligne

Copiez-collez la ligne suivante dans votre console et appuyer sur Entrée.

Code : Console

```
SELECT 1; SELECT 'Hello World !'; SELECT 'Troisième instruction';
```

Ces trois instructions sont exécutées dans l'ordre dans lequel vous les avez écrites :

Code : Console

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

+---------------+
| Hello World ! |
+---------------+
| Hello World ! |
+---------------+
1 row in set (0.00 sec)

+------------------------+
| Troisième instruction  |
+------------------------+
| Troisième instruction  |
+------------------------+
1 row in set (0.00 sec)
```

Commentaires

Les commentaires sont des parties de code qui ne sont pas interprétées. Ils servent principalement à vous repérer dans votre code. En SQL, les commentaires sont introduits par -- (deux tirets). Cependant, MySQL déroge un peu à la règle SQL et accepte deux syntaxes :

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

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

Chaînes de caractères

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

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

La commande suivante sert à afficher "Bonjour petit Zéro !"

Code : SQL

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

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

Code : SQL

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

Code : SQL

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

Quelques autres caractères spéciaux :

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

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

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

Code : SQL

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

Code : SQL

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

Code : SQL

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

Un peu de math

Mysql est également doué en calcul :

Code : SQL

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

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

Code : Console

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

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

Code : SQL

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

Résultat :

Code : Console

```
+---------+------+
| (5+3)*2 |5+3*2 |
+---------+------+
|      16 |   11 |
+---------+------+
```

Encodage

L'encodage d'un fichier ou d'un système définit le jeu de caractères qui va être utilisé par ce fichier/système. Par défaut, vous pourrez toujours lire les 26 lettres de l'alphabet, les chiffres, et les signes de ponctuation et d'opération classiques. Par contre, pour les caractères spéciaux, souvent spécifiques à une langue, il faut définir l'encodage correspondant. Par exemple, pour les caractères accentués (é, è, à, ...), vous pouvez utiliser le jeux de caractères ISO-8859-15. Pour utiliser des caractères hébreux, ce sera ISO-8859-8.
Il existe également l'encodage UTF8, qui permet d'utiliser à peu près tous les caractères existants. Son désavantage est que les textes prennent un peu plus de place en mémoire. C'est cet encodage-là que je vous propose d'utiliser, afin de parer toute éventualité.
Pour prévenir MySQL que vous utiliserez l'encodage UTF8, il suffit de taper la commande suivante :

Code : SQL

```
SET NAMES 'utf8';
```

Et voilà, mysql se chargera d'encoder les données correctement.

Utilisateur

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

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

Code : SQL

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

Je décortique donc rapidement :

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

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

Code : Console

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

Ca y est, vous aller pouvoir faire joujou avec vos données ! Il vous reste un dernier chapitre de pure théorie. Courage ! 

Les types de données

Nous avons vu dans l'introduction, qu'une base de données contenait des **tables**, qui elles-mêmes sont organisées en **colonnes**, dans lesquelles sont stockées nos données.
En SQL (et dans la plupart des langages informatiques), les données sont séparées en plusieurs **types** (par exemple : texte, nombre entier, date...). Lorsque l'on définit une colonne dans une table de la base, il faut donc lui donner un type, et toutes les données stockées dans cette colonne devront correspondre au type de la colonne. Nous allons donc voir les différents types de données existant dans MySQL.

Il est important de bien comprendre les usages et particularités de chaque type de données, afin de **choisir le meilleur type possible** lorsque vous définissez les colonnes de vos tables. En effet, choisir un mauvais type de données pourrait résulter en :

- un gaspillage de mémoire (ex. : si vous stockez de toutes petites données dans une colonne faite pour stocker de grosses quantités de données) ;
- des problèmes de performance (ex. : il est plus rapide de faire une recherche sur un nombre que sur une chaîne de caractères) ;
- un comportement contraire à celui attendu (ex. : trier sur un nombre stocké comme tel, ou sur un nombre stocké comme une chaîne de caractères ne donnera pas le même résultat) ;
- l'impossibilité d'utiliser des fonctionnalités particulières à un type de données (ex; : stocker une date comme une chaîne de caractères vous prive des nombreuses fonctions temporelles disponibles).

Types numériques

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

Nombres entiers

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

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

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

L'attribut UNSIGNED

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

Limiter la taille d'affichage et l'attribut ZEROFILL

Il est possible de préciser le nombre de chiffres minimum **à l'affichage** d'une colonne de type `INT` (ou un de ses dérivés). Il suffit alors de préciser ce nombre entre parenthèses : `INT(x)`. Notez bien que cela ne change pas les capacités de stockage dans la

colonne. Si vous déclarez un `INT(2)`, vous pourrez toujours y stocker 45282 par exemple. Simplement, si vous stockez un nombre avec un nombre de chiffres inférieur au nombre défini, le caractère par défaut sera ajouté à gauche du chiffre, pour qu'il prenne la bonne taille. Sans précision, le caractère par défaut est l'espace.

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

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

Code : SQL

```
INT(4) ZEROFILL
```

Vous aurez l'affichage suivant :

Nombre stocké	Nombre affiché
45	0045
4156	4156
785164	785164

Nombres décimaux

Cinq mots-clés peuvent permettre a une colonne de stocker des nombres décimaux : `DECIMAL`, `NUMERIC`, `FLOAT`, `REAL` et `DOUBLE`.

NUMERIC et DECIMAL

`NUMERIC` et `DECIMAL` sont équivalents, et prennent jusqu'à deux paramètres : la précision et l'échelle.

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

Dans un champ `DECIMAL(5,3)`, on peut donc stocker des nombres de 5 chiffres significatifs maximum, dont 3 chiffres sont après la virgule. Par exemple : 12.354, -54.258, 89.2 ou -56.
`DECIMAL(4)` équivaut à écrire `DECIMAL(4, 0)`.

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

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

FLOAT, DOUBLE et REAL

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

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

Valeurs exactes VS valeurs approchées

Les nombres stockés en tant que `NUMERIC` ou `DECIMAL` sont stockés sous forme de chaînes de caractères. Par conséquent, c'est la valeur exacte qui est stockée. Par contre, les types `FLOAT`, `DOUBLE` et `REAL` sont stockés sous forme de nombres, et c'est une valeur approchée qui est stockée.
Cela signifie que si vous stockez par exemple 56,6789 dans une colonne de type `FLOAT`, en réalité, MySQL stockera une valeur qui se rapproche de 56,6789 (par exemple, 56,678900000000000001). Cela peut poser problème pour des comparaison notamment (56,678900000000000001 n'étant pas égal à 56,6789). S'il est nécessaire de conserver la précision exacte de vos données (l'exemple type est celui des données bancaires), il est donc conseillé d'utiliser un type numérique à valeur exacte (`NUMERIC` ou `DECIMAL` donc).

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

Types alphanumériques
Chaînes de type texte

CHAR et VARCHAR

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

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

Texte	CHAR(5)	Mémoire requise	VARCHAR(5)	Mémoire requise
''	' '	5 octets	''	1 octet
'tex'	'tex '	5 octets	'tex'	4 octets
'texte'	'texte'	5 octets	'texte'	6 octets
'texte trop long'	'texte'	5 octets	'texte'	6 octets

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

TEXT

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

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

Type	Longueur maximale	Mémoire occupée
`TINYTEXT`	2^8 caractères	Longueur de la chaîne + 1 octet
`TEXT`	2^16 caractères	Longueur de la chaîne + 2 octets
`MEDIUMTEXT`	2^24 caractères	Longueur de la chaîne + 3 octets
`LONGTEXT`	2^32 caractères	Longueur de la chaîne + 4 octets

Chaînes de type binaire

Comme les chaînes de type texte que l'on vient de voir, une chaîne binaire n'est rien d'autre qu'une suite de caractères. On distingue cependant deux différences entre les chaînes binaires et les chaînes de texte.

- Dans une chaîne de texte, on ne peut utiliser qu'un set précis de caractères. Ce set est défini par l'encodage de la base de données. Dans une chaîne binaire par contre tous les caractères sont autorisés.
- Le traitement des caractères de la chaîne ne s'effectuent pas de la même manière. Une chaîne binaire traitera l'octet lui-même, ou plutôt le code représentant le caractère stocké, tandis que la chaîne de texte traitera les caractères selon des paramètres définis dans l'environnement local. Une des conséquences est qu'un tri en MySQL sur une chaîne binaire tiendra compte de la casse, tandis qu'un tri sur une chaîne de texte n'en tiendra pas compte.

Décortiquons un peu tout ceci... Un octet représente une quantité de données stockée en mémoire. Dans un octet, on peut stocker 256 valeurs différentes, c'est-à-dire 2^8 valeurs. En effet un octet correspond également à huit bits. Chaque bit pouvant prendre la valeur 0 ou 1, donc deux valeurs possibles, pour huit bits, on a bien 2^8 possibilités. On représente donc souvent les valeurs possibles d'un octet par les nombres 0 à 255.
Lorsque l'on stocke une chaîne binaire, chaque caractère de la chaîne est stocké dans un octet. Il y a donc 256 caractères binaires différents. Et lorsque l'on traite une chaîne binaire, c'est la valeur décimale stockée dans l'octet (donc le nombre entre 0 et 255) qui est traitée directement.
Par contre, dans une chaîne de texte (non binaire), la manière dont sont stockés les caractères dépend des paramètres du serveur. Par exemple, quelqu'un qui utilise des caractères cyrilliques utilisera un paramétrage différent de celui qui utilise l'alphabet latin (comme nous). D'autre part, pour une chaîne de texte, 'A' et 'a' représentent tous les deux la lettre 'a'. Tandis qu'une chaine binaire aura stocké 'A' comme le caractère 65, et 'a' comme le caractère 97. Ils ne seront donc pas du tout traités comme équivalents. Et enfin, pour terminer, les caractères autorisés par une chaîne non binaire sont moins nombreux. On parle de caractères "affichables" et "non-affichables". Les valeurs 0 à 31 et 127 à 255 sont non-affichables, tandis que les valeurs 32 à 126 sont affichables (l'alphabet majuscule et minuscule, les chiffres, les signes de ponctuation et d'opération notamment).

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

Les types binaires sont définis de la même façon que les types de chaînes de texte. `VARBINARY(x)` et `BINARY(x)` permettent de stocker des chaînes binaires de *x* caractères maximum (avec une gestion de la mémoire identique à `VARCHAR(x)` et `CHAR(x)`). Et pour les chaînes plus longues, il existe les types `TINYBLOB`, `BLOB`, `MEDIUMBLOB` et `LONGBLOB`, également avec les mêmes limites de stockage que les types `TEXT`.

SET et ENUM

ENUM

Une colonne de type `ENUM` est une colonne pour laquelle on définit un certain nombre de valeurs autorisées, de type "chaînes de caractères". Par exemple, si l'on définit une colonne espèce de la manière suivante :

Code : SQL

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

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

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

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

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

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

Valeur	Index
`NULL`	**`NULL`**
`''`	0
'chat'	1
'chien'	2
'tortue'	3

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

Un `ENUM` peut avoir maximum 65535 valeurs possibles

SET

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

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

Code : SQL

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

On pourra stocker dans cette colonne :

- `''` (chaîne vide) ;
- `'chat'` ;
- `'chat,tortue'` ;

- `'chat,chien,tortue'` ;
- `'chien,tortue'` ;
- ...

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

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

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

Code : SQL

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

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

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

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

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

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

Pour stocker 'chat' et 'tortue' dans un champ, on peut donc utiliser 'chat,tortue' ou 101 (addition des nombres binaires correspondants) ou 5 (addition des nombres décimaux correspondants).
Notez que cette utilisation des binaires a pour conséquence que l'ordre dans lequel vous rentrez vos valeurs n'a pas d'importance. Que vous écriviez 'chat,tortue' ou 'tortue,chat' ne fait aucune différence. Lorsque vous récupérerez votre champ, vous aurez 'chat,tortue' (même ordre que lors de la définition du champ, donc).

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

Avertissement

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

Pourquoi avoir inventé ces types propres à MySQL ?

La plupart des SGBD implémente ce qu'on appelle des contraintes d'assertions, qui permettent de définir les valeurs que peuvent prendre une colonne (par exemple, on pourrait définir une contrainte pour une colonne contenant un âge, devant être compris entre 0 et 130).
MySQL n'implémente pas ce type de contrainte, et a par conséquent créé deux types de données spécifiques (**SET** et ENUM), pour pallier en partie ce manque.

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

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

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

Code : SQL

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

Tout se passe plutôt bien tant que vos éléments appartiennent aux catégories que vous avez définies au départ. Et puis tout à coup, vous vous retrouvez avec un élément qui ne correspond à aucune de vos catégories, mais qui devrait plutôt se trouver dans la catégorie "Entrées". Avec **SET** ou ENUM, il vous faut modifier la colonne *categorie* pour ajouter "Entrées" aux valeurs possibles. Or, une des règles de base à respecter lorsque l'on conçoit une base de données, c'est que **la structure de la base (donc les tables, les colonnes) ne doit pas changer lorsque l'on ajoute des données**. Par conséquent, tout ce qui est susceptible de changer doit être une donnée, et non faire partie de la structure de la base.

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

Pour éviter ENUM

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

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

Types temporels

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

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

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

DATE, TIME et DATETIME

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

DATE

Pour entrer une date, l'ordre des données est la seule contrainte. Il faut donner d'abord l'année (deux ou quatre chiffres), ensuite le mois (deux chiffres) et pour finir, le jour (deux chiffres), sous forme de nombre ou de chaîne de caractères. S'il s'agit d'une chaîne de caractères, n'importe quelle ponctuation peut être utilisée pour délimiter les partie (ou aucune). Voici quelques exemples d'expressions correctes (A représente les années, M les mois et J les jours) :

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

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

- si l'année donnée est entre 00 et 69, on utilisera le 21e siècle, on ira donc de 2000 à 2069 ;
- par contre, si l'année est comprise entre 70 et 99, on utilisera le 20e siècle, donc entre 1970 et 1999.

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

DATETIME

Très proche de DATE, ce type permet de stocker une heure, en plus d'une date. Pour entrer un DATETIME, c'est le même principe que pour DATE : pour la date, année-mois-jour, et pour l'heure, il faut donner d'abord l'heure, ensuite les minutes, puis les secondes. Si on utilise une chaîne de caractères, il faut séparer la date et l'heure par une espace. Quelques exemples corrects (H représente les heures, M les minutes et S les secondes) :

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

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

TIME

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

- 'HH:MM:SS'
- 'HHH:MM:SS'

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

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

YEAR

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

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

TIMESTAMP

Par définition, le timestamp d'une date est le nombre de secondes écoulées depuis le 1er janvier 1970, 0h0min0s (TUC) et la date en question.
Les timestamps étant stockés sur 4 octets, il existe une limite supérieure : le 19 janvier 2038 à 3h14min7s. Par conséquent, vérifiez bien que vous êtes dans l'intervalle de validité avant d'utiliser un timestamp.
Par ailleurs, et il faut bien avouer que c'est un peu contre-intuitif, dans MySQL, les timestamps ne sont pas stockés sous forme d'un nombre, mais de la même manière qu'un `DATETIME`, sous format numérique (alors que les `DATETIME` sont stockés comme des chaînes de caractères, même si elles peuvent être insérées grâce à un nombre) : AAAAMMJJHHMMSS. Méfiez-vous, car ce n'est pas le cas dans beaucoup de langages informatiques, et c'est source de nombreuses erreurs (on ne peut pas stocker directement un timestamp PHP dans un champ de type **`TIMESTAMP`** d'une base de données SQL par exemple).

La date par défaut

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

Type	Date par défaut ("zéro")
`DATE`	`'0000-00-00'`
`DATETIME`	`'0000-00-00 00:00:00'`
`TIME`	`'00:00:00'`
`YEAR`	`0000`
`TIMESTAMP`	`00000000000000`

Un exception toutefois, si vous insérez un `TIME` qui dépasse l'intervalle de validité, MySQL ne le remplacera pas par le "zéro", mais par la plus proche valeur appartenant à l'intervalle de validité (-838:59:59 ou 838:59:59 donc).
La théorie pure est maintenant terminée, nous allons passer à la création d'une base de données, et de tables dans celle-ci.

Création d'une base de données

Ça y est, le temps est venu d'écrire vos premières lignes de commande.
Dans ce chapitre plutôt court, je vous donnerai pour commencer quelques conseils indispensables. Ensuite, je vous présenterai la problématique sur laquelle nous allons travailler tout au long de ce tutoriel : la base de données d'un élevage d'animaux.
Pour finir, nous verrons comment créer, et supprimer, une base de données.

La partie pure théorie est donc bientôt finie. Gardez la tête et les mains à l'intérieur du véhicule, et c'est parti !

Avant-propos : conseils et conventions

Conseils

Noms de tables et de colonnes

N'utilisez jamais, au grand jamais, d'espaces ou d'accents dans vos noms de bases, tables ou colonnes. Au lieu d'avoir une colonne "date de naissance", préférez "date_de_naissance" ou "date_naissance". Et au lieu d'avoir une colonne "prénom", utilisez "prenom". Avouez que ça reste lisible... Et ça vous évitera pas mal d'ennuis.
Evitez également d'utiliser des mots réservés comme nom de colonnes/tables/bases. Par "mot réservé", j'entends un mot-clé SQL, donc un mot qui sert à définir quelque chose dans le langage SQL. Bien sûr, je peux difficilement vous faire la liste des mots à éviter. Parmi les plus fréquents : `date`, `text`, **`type`**. Ajoutez donc une précision à vos noms dans ces cas-là (date_naissance, text_article ou type_personnage par exemple).

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

Soyez cohérent

Vous vous y retrouverez bien mieux si vous restez cohérent dans votre base. Par exemple, mettez tous vos noms de tables au singulier. Ou tous vos noms de tables au pluriel. Choisissez, mais tenez vous-y. Même chose pour les noms de colonnes. Et lorsqu'un nom de table ou de colonne nécessite plusieurs mots, séparez les toujours avec '_' (ex : date_naissance). Ou toujours avec une majuscule (ex : dateNaissance). Ce ne sont que quelques exemples de situations dans lesquelles vous devez décider d'une marche à suivre, et la garder tout au long de votre projet (voire pour tous vos projets futurs). Vous gagnerez énormément de temps en prenant de telles habitudes.

Conventions

Mots-clés

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

Noms de bases, de tables et de colonnes

Je viens de vous dire que les mots-clés SQL seront écrits en majuscule pour les différencier du reste, donc évidemment, les noms de bases, tables et colonnes seront écrit en minuscule.
Toutefois, par habitude et parce que je trouve cela plus clair, je mettrai une majuscule à la première lettre de mes noms de tables (et uniquement les tables, pas la base de données ni les colonnes). Notez que MySQL n'est pas nécessairement sensible à la casse en ce qui concerne les noms de tables et de colonnes. En fait, il est très probable que si vous travaillez sous Windows, MySQL ne soit pas sensible à la casse pour les noms de tables et de colonnes. Sous Mac et Linux par contre, c'est le contraire qui est le plus probable.
Quoiqu'il en soit, j'utiliserai des majuscule pour la première lettre de mes tables. Libre à vous de me suivre ou non.

Options facultatives

Lorsque je commencerai à vous montrer les commandes SQL à utiliser pour interagir avec votre base de données, vous verrez que certaines commandes ont des options facultatives. Dans ces cas-là, j'utiliserai des crochets [ ] pour indiquer ce qui est

facultatif. La même convention est utilisée dans la documentation officielle MySQL (et beaucoup d'autres documentations d'ailleurs). La requête suivante signifie donc que vous pouvez commander votre glace vanille toute seule, ou avec du chocolat, ou avec de la chantilly, ou avec du chocolat ET de la chantilly.

Code : Autre

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

Mise en situation

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

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

Dossier de travail

Histoire de s'y retrouver un peu, je vais vous demander de créer un dossier spécialement pour ce tutoriel. C'est dans ce dossier que nous mettrons les fichiers de requêtes à exécuter, les sauvegardes de notre base de données, etc.
Je vous propose de l'appeler sdzMysql. Créez-le où vous le sentez. Une fois cela fait, retournez dans la console, mais avant de vous connecter à mysql, allez dans votre dossier sdzMysql. Pour ce faire, il suffit d'utiliser la commande suivante :

Code : Console

```
cd chemin_du_dossier
```

Donc par exemple, sous Windows, cela peut ressembler à quelque chose comme ça (si vous avez mis le dossier dans le répertoire `Mes Documents`) :

Code : Console

```
cd C:\"Documents and Settings"\user\"My Documents"\sdzMysql
```

Si vous n'êtes pas sûr du chemin de votre dossier, il y a deux solutions.

- Soit vous allez voir les propriétés de votre dossier dans l'explorateur (clic droit puis "properties" ou quelque chose du genre qui dépend de votre environnement).
- Soit vous tapez "cd " dans la console, puis cliquez sur votre dossier dans l'explorateur et le faites glisser dans la fenêtre de la console. Le chemin vers le dossier s'écrira alors automatiquement, et vous n'aurez plus qu'à taper Entrée

Lorsque vous êtes dans votre dossier, vous pouvez vous connecter à mysql comme d'habitude, et nous allons pouvoir commencer les choses sérieuses.

Code : Console

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

Création et suppression d'une base de données

Création

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

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

Code : SQL

```
CREATE DATABASE nom_base;
```

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

Cependant attendez avant de créer votre base de données "elevage". Vous vous souvenez que je vous ai demandé de définir un jeu de caractères particulier pour MySQL : l'UTF-8 ? Hé bien nous allons également définir notre base de données comme encodée en UTF-8. Voici donc la commande complète à taper pour créer votre base :

Code : SQL

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

Suppression

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

Code : SQL

```
DROP DATABASE elevage;
```

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

Code : Console

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

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

Code : SQL

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

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

Code : Console

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

Si elle n'existe pas, vous aurez :

Code : Console

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

Pour afficher les warnings de MySQL, il faut utiliser la commande

Code : SQL

```
SHOW WARNINGS;
```

Cette commande affiche un tableau :

Code : Console

```
+-------+------+-----------------------------------------------------------+
| Level | Code | Message                                                   |
+-------+------+-----------------------------------------------------------+
| Note  | 1008 | Can't drop database 'elevage'; database doesn't exist |
+-------+------+-----------------------------------------------------------+
```

Utilisation d'une base de données

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

Code : SQL

```
USE elevage
```

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

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

Code : Console

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

C'était le dernier chapitre avec autant de théorie pour cette partie. Promis ! A partir de maintenant, vous allez pouvoir pratiquer, manipuler vos données, chipoter, essayer, inventer, et plein d'autres trucs trop cool !

Création de tables

Dans ce chapitre, nous allons créer, étape par étape, une table *Animal*, qui servira à stocker les animaux présents dans notre élevage.
Soyez gentils avec cette table car c'est elle qui vous accompagnera tout au long de la première partie (on apprendra à jongler avec plusieurs tables dans la deuxième partie).

Pour commencer, il faudra définir de quelles colonnes (et leur type) la table sera composée. Ne négligez pas cette étape, c'est la plus importante. Une base de données mal conçue est un cauchemar à utiliser.
Ensuite, petit passage obligé par de la théorie : vous apprendrez ce qu'est une clé primaire et à quoi ça sert, et découvrirez cette fonctionnalité exclusive de MySQL que sont les moteurs de table.
Enfin, la table *Animal* sera créée, et la requête de création des tables décortiquée. Et dans la foulée, nous verrons également comment supprimer une table.

Définition des colonnes

Type de colonne

Avant de choisir le type des colonnes, il faut choisir les colonnes que l'on va définir. On va donc créer une table "Animal". Qu'est-ce qui caractérise un animal ? Son espèce, son sexe, sa date de naissance. Quoi d'autre ? Une éventuelle colonne "commentaires" qui peut servir de fourre-tout. Si l'on est 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 mot, choisi parmi trois. On pourrait donc penser qu'un `ENUM` avec trois possibilités est une bonne idée. Il n'est cependant pas impossible que d'ici quelques temps nous élevions également des perroquets, ou des harfangs des neiges. Auquel cas, il nous faudra changer l'`ENUM` et lui rajouter des champs, ce qui est un peu lourd. Donc il vaut sans doute mieux 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'espèce ne contiennent qu'un mot, mais harfang des neiges 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 suffisamment faible pour décider d'utiliser un `ENUM`.
- **Date de naissance** : pas besoin de réfléchir beaucoup ici. Il s'agit d'une date, donc soit un `DATETIME`, soit une `DATE`. L'heure de la naissance est-elle importante ? Disons que oui, du moins pour les soins lors des premiers jours. `DATETIME` donc !
- **Commentaires** : de nouveau un type alphanumérique évidemment, mais on a ici aucune idée de la longueur. Ce sera sans doute succinct mais il faut prévoir un minimum de place quand même. Ce sera donc un champ `TEXT`.
- **Nom** : plutôt facile à déterminer. On prendra simplement un `VARCHAR(30)`. On ne pourra pas appeler nos tortues "Petite maison dans la prairie verdoyante", mais c'est amplement suffisant pour "Rox" ou "Roucky".

NULL or not NULL ?

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

- **Espèce** : un éleveur qui se respecte connait l'espèce des animaux qu'il élève. On n'autorisera donc pas la colonne espèce 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 connait 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 peut avoir entre 1 et 8 petits d'un coup. Allez-y pour inventer 8 noms originaux comme ça !), il vaut mieux autoriser cette colonne à être **`NULL`**.

Récapitulatif

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

Caractéristique	Nom de la colonne	Type	NULL?
Espèce	espece	VARCHAR(40)	Non
Sexe	sexe	ENUM('male','femelle')	Oui
Date de naissance	date_naissance	DATETIME	Non
Commentaires	commentaires	TEXT	Oui
Nom	nom	VARCHAR(30)	Oui

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 ? Evidemment, on pourrait s'arranger pour que deux animaux n'aient jamais le même nom. Mais imaginez la situation suivante : une chatte vient de donner naissance à sept petits. On ne peut pas encore définir leur sexe, on n'a pas encore trouvé de nom pour certains d'entre eux et il n'y a encore aucun commentaire à faire à leur propos. Ils auront donc exactement les mêmes caractéristiques. Pourtant, ce ne sont pas les mêmes individus. Il faut donc les différencier. Pour cela, on va ajouter une colonne à notre table.

Identité

Imaginez que quelqu'un ait le même nom de famille que vous, le même prénom, soit né dans la même ville et ait la même taille. En dehors de la photo et de la signature, quelle sera la différence entre vos deux cartes d'identité ? Son numéro !
Suivant le même principe, on va donner à chaque animal un **numéro d'identité**. La colonne qu'on ajoutera s'appelera donc *id*, et il s'agira d'un `INT`. Selon la taille de l'élevage (la taille actuelle mais aussi la taille qu'on imagine qu'il pourrait avoir dans le futur !), il peut être plus intéressant d'utiliser un `SMALLINT`, voire un `MEDIUMINT`. Comme il est peu probable que l'on dépasse les 65000 animaux, on utilisera `SMALLINT`. Attention, il faut bien considérer tous les animaux qui entreront un jour dans la base, pas uniquement le nombre d'animaux présents en même temps dans l'élevage. En effet, si l'on supprime pour une raison ou une autre un animal de la base, il n'est pas question de réutiliser son numéro d'identité.

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

Clé primaire

La clé primaire d'une table est une contrainte d'unicité, composée d'une ou plusieurs colonne(s). 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 insèrerez une nouvelle ligne dans la table, MySQL vérifiera que vous insérez bien une *id*, et que cette *id* n'existe pas encore dans la table. Si vous ne respectez pas ces deux contraintes, MySQL n'insèrera pas la ligne et vous renverra une erreur.

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

Code : Console

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

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

Auto-incrémentation

Il faut donc, pour chaque animal, décider d'une valeur pour id. Le plus simple, et le plus logique, est de donner le numéro 1 au premier individu enregistré, puis le numéro 2 au second, etc.
Mais si vous ne vous souvenez pas quel numéro vous avez utilisé en dernier, pour insérer un nouvel animal il faudra récupérer cette information dans la base, ensuite seulement vous pourrez ajouter une ligne en lui donnant comme id la dernière id utilisée + 1.
C'est bien sûr faisable, mais c'est fastidieux.. Heureusement, il est possible de demander à MySQL de faire tout ça pour nous ! Comment ? En utilisant l'auto-incrémentation des colonnes. Incrémenter veut dire "ajouter une valeur fixée". Donc, si l'on déclare qu'une colonne doit s'auto-incrémenter (grâce au mot-clé `AUTO_INCREMENT`), plus besoin de chercher quelle valeur on va mettre dedans lors de la prochaine insertion. MySQL va chercher ça tout seul comme un grand en prenant la dernière valeur insérée et en l'incrémentant de 1.

Les moteurs de tables

Les moteurs de tables sont une spécificité de MySQL. Ce sont des moteurs de stockage. Ca permet de gérer différemment les tables selon l'utilité qu'on en a. Je ne vais pas vous détailler tous les moteurs de tables existants. Si vous voulez plus d'informations, je vous renvoie à la documentation officielle.
Les deux moteurs les plus connus sont **MyISAM** et **InnoDB**.

MyISAM

C'est le moteur par défaut. Les commandes d'insertion et sélection de données sont particulièrement rapides sur les tables utilisant ce moteur. Cependant, il ne gère pas certaines fonctionnalités importantes comme les clés étrangères (qui permettent de vérifier l'intégrité d'une référence d'une table à une autre table, plus d'infos viendront dans la deuxième partie du cours) ou les transactions (qui permettent de faire des séries de modifications "en bloc", donc toute la série de modifications est effectuée ou aucune modification de la série, plus d'infos viendront dans la troisième partie du cours).

InnoDB

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

Préciser un moteur lors de la création de la table

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

Code : SQL

```
ENGINE = moteur;
```

En remplaçant bien sûr "moteur" par le nom du moteur que nous voulons utiliser. Donc pour InnoDB :

Code : SQL

```
ENGINE = INNODB;
```

Syntaxe de CREATE TABLE

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

Caractéristique	Nom du champ	Type	NULL ?	Divers
Numéro d'identité	id	`SMALLINT`	Non	Clé primaire + auto-incrément
Espèce	espece	`VARCHAR(40)`	Non	-

Sexe	sexe	ENUM('male','femelle')	Oui	-
Date de naissance	date_naissance	DATETIME	Non	-
Commentaires	commentaires	TEXT	Oui	-
Nom	nom	VARCHAR(30)	Oui	-

Syntaxe

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

Création de la table

Code : SQL

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

Le `IF NOT EXISTS` est facultatif (d'où l'utilisation de crochet [ ]), 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 après. Nous verrons comment plus tard.

Définition des colonnes

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

Code : SQL

```
nom VARCHAR(30),
sexe ENUM('male','femelle')
```

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

Code : SQL

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

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

Code : SQL

```
id SMALLINT 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 :

Code : SQL

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

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

Application : création de "Animal"

Si l'on met tout cela ensemble pour créer la table "Animal" (je rappelle que nous utiliserons le moteur InnoDB), on a donc :

Code : SQL

```
CREATE TABLE Animal (
    id SMALLINT 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;
```

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.

Vérifications

Au cas où vous ne me croiriez pas (et aussi un peu car cela pourrait vous être utile un jour), voilà comment vous pouvez vérifier que vous avez bien créé une jolie table "Animal" avec les six colonnes que vous vouliez.

Voir toutes les tables d'une base de données

La commande suivante vous affiche un tableau contenant toutes les tables créées dans la base de données que vous utilisez :

Code : SQL

```
SHOW TABLES;
```

Résultat :

Code : Console

```
+-----------------+
|Tables_in_elevage|
+-----------------+
|Animal           |
+-----------------+
```

Comme je vous l'ai dit au chapitre précédent, il est possible que votre serveur ne soit pas sensible à la casse pour les noms de champs et de tables. Le cas échéant, il est tout à fait normal d'avoir une table nommée "animal" au lieu de "Animal", même si vous avez mis la majuscule dans la commande de création de la table.

Voir la structure d'une table

Et pour voir la structure détaillée de la table "Animal", on utilise cette commande :

Code : SQL

```
DESCRIBE Animal;
```

Résultat :

Code : Console

```
+--------------+----------------------+----+---+-------+--------------+
|Field         |Type                  |Null|Key|Default|Extra         |
+--------------+----------------------+----+---+-------+--------------+
|id            |smallint(6)           |NO  |PRI|NULL   |auto_increment|
|espece        |varchar(40)           |NO  |   |NULL   |              |
|sexe          |enum('male','femelle')|YES |   |NULL   |              |
|date_naissance|datetime              |NO  |   |NULL   |              |
|nom           |varchar(30)           |YES |   |NULL   |              |
|commentaires  |text                  |YES |   |NULL   |              |
+--------------+----------------------+----+---+-------+--------------+
```

Alors, convaincu ?

Suppression d'une table

La commande pour supprimer une table est la même que celle pour supprimer une base de données. Elle est, bien sûr, à utiliser avec prudence, car irréversible.

Code : SQL

```
DROP TABLE Animal;
```

Tout ça commence à prendre forme. Nous avons donc une base, une table, il ne manque plus que les données. Un dernier petit chapitre sur la manipulation des tables, et nous rentrerons dans le vif du sujet !

Modification d'une table

La création et suppression de tables étant acquises, parlons maintenant des requêtes permettant de modifier une table.
Et 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 par encore, aussi n'en parlerai-je pas ici.

Syntaxe de la requête

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

Code : SQL

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

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

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

Créons une table pour faire joujou

Dans la seconde partie de ce tutoriel, nous 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** :

Code : SQL

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

Ajout et suppression d'une colonne

Ajout

On utilise la syntaxe suivante :

Code : SQL

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

Le `[COLUMN]` est facultatif, donc si à la suite de **ADD**, vous ne précisez pas ce que vous voulez ajouter, MySQL considérera qu'il s'agit d'une colonne.
`description_colonne` correspond à la même chose que lorsque l'on crée une table, donc le type de donnée, éventuellement un **NULL** ou **NOT NULL**, etc.

Ajoutons donc 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 donc :

Code : SQL

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

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

Suppression

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

Code : SQL

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

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

Exemple d'utilisation : nous allons supprimer la colonne *date_insertion*, que nous remercions pour son passage éclair dans notre tuto.

Code : SQL

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

Modification de colonne

Changement du nom de la colonne

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

Code : SQL

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

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

Code : SQL

```sql
ALTER TABLE Test_tuto
CHANGE nom prenom VARCHAR(10) NOT NULL;
```

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

Changement du type de données

Les mots clés `CHANGE` et `MODIFY` peuvent être utilisé pour changer le type de données 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 :

Code : SQL

```
ALTER TABLE nom_table
CHANGE ancien_nom nouveau_nom nouvelle_description;

ALTER TABLE nom_table
MODIFY nom_colonne nouvelle_description;
```

Des exemples pour illustrer :

Code : SQL

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

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

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

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

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

Bonne nouvelle, dans le prochain chapitre, vous commencerez enfin à vraiment manipuler vos données.  Après tout, c'est pour ça que vous êtes là, non ?

Insertion de données

Ce chapitre est consacré à l'insertion de données dans une table. Rien de bien compliqué, mais c'est évidemment crucial. Car 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.

Syntaxe de INSERT

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

Insertion sans préciser les colonnes

Je rappelle pour les distraits que notre table "Animal" est composée de six colonnes : id, espece, sexe, date_naissance, nom et commentaires.

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

Code : SQL

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

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

Code : SQL

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

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

Code : SQL

```
INSERT INTO Animal
VALUES (NULL , 'chat', 'femelle', '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	male	2010-04-05 13:43:00	Rox	Mordille beaucoup
2	chat	NULL	2010-03-24 02:23:00	Roucky	NULL
3	chat	femelle	2010-09-13 15:02:00	Schtroumpfette	NULL

Pour vérifier, vous pouvez utiliser la requête suivante :

Code : SQL

```
SELECT * FROM Animal;
```

Deux choses importantes à retenir ici :

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

Insertion en précisant les colonnes

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

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

Quelques exemples :

Code : SQL

```
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');
```

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

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

4	tortue	femelle	2009-08-03 05:12:00	NULL	NULL
5	chat	NULL	2010-10-03 16:44:00	Choupi	Né sans oreille gauche
6	tortue	femelle	2009-06-13 08:17:00	Bobosse	Carapace bizarre

Insertion multiple

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

Code : SQL

```
INSERT INTO Animal (espece, sexe, date_naissance, nom)
VALUES ('chien', '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);
```

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 !

Id	Espèce	Sexe	Date de naissance	Nom	Commentaires
1	chien	male	2010-04-05 13:43:00	Rox	Mordille beaucoup
2	chat	NULL	2010-03-24 02:23:00	Roucky	NULL
3	chat	femelle	2010-09-13 15:02:00	Schtroumpfette	NULL
4	tortue	femelle	2009-08-03 05:12:00	NULL	NULL
5	chat	NULL	2010-10-03 16:44:00	Choupi	Né sans oreille gauche
6	tortue	femelle	2009-06-13 08:17:00	Bobosse	Carapace bizarre
7	chien	femelle	2008-12-06 05:18:00	Caroline	NULL
8	chat	male	2008-09-11 15:38:00	Bagherra	NULL
9	tortue	NULL	2010-08-23 05:18:00	NULL	NULL

Syntaxe alternative de MySQL

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

Code : SQL

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

Cette syntaxe présente à mes yeux 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 pus 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.

- Cette syntaxe 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.
- Cela ne permet pas l'insertion multiple.

Utilisation de fichiers externes

Maintenant que vous savez insérer des données, je vous propose de remplir un peu cette table, histoire qu'on puisse s'amuser par la suite.
Rassurez-vous, je ne vais pas vous demander d'inventer cinquante bestioles et d'écrire une à une les requêtes permettant des les insérer. Je vous ai prémâché le boulot. De plus, ça nous permettra d'avoir la même chose dans notre base. Ce sera donc plus facile pour vous de vérifier que vos requêtes font bien ce qu'elles doivent.
Et pour éviter d'écrire vous-même toutes les requêtes d'insertion, nous allons donc voir comment on peut utiliser un fichier texte pour interagir avec notre base de données.

Exécuter des commandes SQL à partir d'un fichier

Ecrire 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 devenir pénible vu qu'on ne peut pas vite sélectionner juste la partie qui nous intéresse, faire un copier coller, modifier juste les valeurs, etc.

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

Code : SQL

```
SOURCE monFichier.sql;
```

Ou

Code : SQL

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

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

Attention : le dossier dans lequel MySQL va aller chercher le fichier si vous ne lui donnez pas de chemin est le dossier dans lequel vous étiez lors de votre connexion. Or, je vous ai fait créer un dossier "sdzMysql" sur votre ordinateur, et vous ai fait entrer dedans avant de vous connecter. C'est donc dans ce même dossier qu'il vous faudra stocker les fichiers que vous voulez faire exécuter à MySQL. Vous pouvez aussi bien sûr créer des sous-dossiers pour ranger tout ça, auquel cas il faudra faire :

Code : SQL

```
SOURCE sous-dossier\monFichier.sql;
```

Si votre fichier se trouve complètement ailleurs sur votre ordinateur, il vous est toujours possible de l'exécuter, en fournissant à MySQL le chemin complet du fichier. Par exemple :

Code : SQL

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

Insérer des données à partir d'un fichier formaté

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

Code : CSV

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

Ce type de fichier est facile a produire (et à lire) avec un logiciel de type tableur (Microsoft Excel, ExcelViewer, Numbers...). Et la bonne nouvelle, c'est qu'il est aussi possible de lire ce type de fichier avec MySQL, afin de remplir une table avec les données contenues dans le fichier.
La commande SQL permettant cela est **LOAD DATA** INFILE, dont voici la syntaxe :

Code : SQL

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

Le mot-clé **LOCAL** sert à spécifier si le fichier se trouve côté client (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 stockés 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 vous 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 de 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 à ce que les colonnes absentes acceptent **`NULL`**, ou soient auto-incrémentées, évidemment.

Si je reprends mon exemple, en imaginant que nous avons une table *Personne*, qui contient les colonnes *id* (clé primaire auto-incrémentée), *nom, prenom, date_naissance* et *adresse* (qui peut être **`NULL`**).

Code : CSV

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

Si ce fichier est enregistré sous le nom personne.csv dans votre dossier de travail, il vous suffit de faire la commande suivante pour enregistrer ces trois lignes dans la table *Personne* :

Code : SQL

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

Remplissage de la base

Nous allons utiliser les deux techniques que je viens de vous montrer pour remplir un peu notre base.

Exécution de commandes SQL

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

Secret (cliquez pour afficher)

Code : SQL

```
INSERT INTO Animal (espece, sexe, date_naissance, nom,
commentaires) VALUES
('chien', 'femelle', '2008-02-20 15:45:00' , 'Canaille', NULL),
('chien', 'femelle','2009-05-26 08:54:00'  , 'Cali', NULL),
('chien', 'femelle','2007-04-24 12:54:00' , 'Rouquine', NULL),
('chien', 'femelle','2009-05-26 08:56:00' , 'Fila', NULL),
('chien', 'femelle','2008-02-20 15:47:00' , 'Anya', NULL),
```

```
('chien', 'femelle','2009-05-26 08:50:00' ,'Louya' , NULL),
('chien', 'femelle', '2008-03-10 13:45:00','Welva' , NULL),
('chien', 'femelle','2007-04-24 12:59:00' ,'Zira' , NULL),
('chien', 'femelle', '2009-05-26 09:02:00','Java' , NULL),
('chien', 'male','2007-04-24 12:45:00' ,'Balou' , NULL),
('chien', 'male','2008-03-10 13:43:00' ,'Pataud' , NULL),
('chien', 'male','2007-04-24 12:42:00' , 'Bouli', NULL),
('chien', 'male', '2009-03-05 13:54:00','Zoulou' , NULL),
('chien', 'male','2007-04-12 05:23:00' ,'Cartouche' , NULL),
('chien', 'male', '2006-05-14 15:50:00', 'Zambo', NULL),
('chien', 'male','2006-05-14 15:48:00' ,'Samba' , NULL),
('chien', 'male', '2008-03-10 13:40:00','Moka' , NULL),
('chien', 'male', '2006-05-14 15:40:00','Pilou' , NULL),
('chat', 'male','2009-05-14 06:30:00' , 'Fiero', NULL),
('chat', 'male','2007-03-12 12:05:00' ,'Zonko', NULL),
('chat', 'male','2008-02-20 15:45:00' , 'Filou', NULL),
('chat', 'male','2007-03-12 12:07:00' , 'Farceur', NULL),
('chat', 'male','2006-05-19 16:17:00' ,'Caribou' , NULL),
('chat', 'male','2008-04-20 03:22:00' , 'Capou', NULL),
('chat', 'male','2006-05-19 16:56:00' , 'Raccou', 'Pas de queue
depuis la naissance');
```

Vous n'avez alors qu'à taper :

Code : SQL

```
SOURCE remplissageAnimal.sql;
```

LOAD DATA INFILE

A nouveau, copiez-collez ce qui se trouve dans la balise **<secret>** ci-dessous dans votre éditeur de texte, et enregistrez le fichier. Cette fois, sous le nom animal.csv.

Secret (cliquez pour afficher)

Code : CSV

```
"chat";"male";"2009-05-14 06:42:00";"Boucan";
"chat";"femelle";"2006-05-19 16:06:00";"Callune";
"chat";"femelle";"2009-05-14 06:45:00";"Boule";
"chat";"femelle";"2008-04-20 03:26:00";"Zara";
"chat";"femelle";"2007-03-12 12:00:00";"Milla";
"chat";"femelle";"2006-05-19 15:59:00";"Feta";
"chat";"femelle";"2008-04-
20 03:20:00";"Bilba";"Sourde de l'oreille droite à 80%"
"chat";"femelle";"2007-03-12 11:54:00";"Cracotte";
"chat";"femelle";"2006-05-19 16:16:00";"Cawette";
"tortue";"femelle";"2007-04-01 18:17:00";"Nikki";
"tortue";"femelle";"2009-03-24 08:23:00";"Tortilla";
"tortue";"femelle";"2009-03-26 01:24:00";"Scroupy";
"tortue";"femelle";"2006-03-15 14:56:00";"Lulla";
"tortue";"femelle";"2008-03-15 12:02:00";"Dana";
"tortue";"femelle";"2009-05-25 19:57:00";"Cheli";
"tortue";"femelle";"2007-04-01 03:54:00";"Chicaca";
"tortue";"femelle";"2006-03-15 14:26:00";"Redbul";"Insomniaque"
"tortue";"male";"2007-04-02 01:45:00";"Spoutnik";
"tortue";"male";"2008-03-16 08:20:00";"Bubulle";
"tortue";"male";"2008-03-15 18:45:00";"Relou";"Surpoids"
"tortue";"male";"2009-05-25 18:54:00";"Bulbizard";
"perroquet";"male";"2007-03-04 19:36:00";"Safran";
"perroquet";"male";"2008-02-20 02:50:00";"Gingko";
```

```
"perroquet";"male";"2009-03-26 08:28:00";"Bavard";
"perroquet";"femelle";"2009-03-26 07:55:00";"Parlotte";
```

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

Exécutez ensuite la commande suivante :

Code : SQL

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

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

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

Code : SQL

```sql
SELECT * FROM Animal;
```

Et nous pouvons maintenant passer au chapitre suivant !
Vous pouvez maintenant dire que vous avez une base de données ! Vous ne savez pas encore vraiment vous en servir, mais ça va venir. Lentement mais sûrement.

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 ;
- Elimination des données en double ;
- Récupération de seulement une partie des données (uniquement les 10 premières ligne par exemple).

Motivés ? Alors c'est parti !!! 

Syntaxe de SELECT

La requête qui permet de sélectionner et afficher des données s'appelle **SELECT**. Vous l'avez déjà un peu utilisée dans le chapitre d'installation de MySQL.
Cependant, jusqu'à présent vous n'avez affiché que des données que vous définissiez vous-même.

Code : SQL

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

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

Code : SQL

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

Les mots "colonne1", "colonne2", ... sont à remplacer par le nom des colonnes que l'on désire afficher ; le mot nom_table est à remplacer par le nom de la table dans laquelle sont stockées les données que l'on veut récupérer.

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

Code : SQL

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

Sélectionner toutes les colonnes

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

Code : SQL

```
SELECT *
FROM Animal;
```

Il est cependant déconseillé d'utiliser **SELECT** * trop souvent. Donner explicitement le nom des colonnes dont vous avez

besoin présente deux avantages :

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

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 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 avec critères.

La clause WHERE

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

Code : SQL

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

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

Les opérateurs de comparaison

Les opérateurs de comparaison sont les symboles que l'ont 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 :

Code : SQL

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

SELECT *
FROM Animal
WHERE espece <> 'chat'; -- Tous les animaux sauf les chats
```

Combinaisons de critères

Tout ça c'est bien beau, mais comment faire si on veut les chats et les chiens par exemple ? Faut-il faire deux requêtes ?
Non bien sûr, il suffit de combiner les critères. Et pour combiner les critères, 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.

AND

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

Code : SQL

```sql
SELECT *
FROM animal
WHERE espece='chat'
    AND sexe='femelle';
-- OU
SELECT *
FROM animal
WHERE espece='chat'
    && sexe='femelle';
```

OR

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

Code : SQL

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

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

NOT

Sélection de tous les animaux femelles sauf les chiennes.

Code : SQL

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

XOR

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

Code : SQL

```
SELECT *
FROM animal
WHERE sexe='male'
    XOR espece='perroquet';
```

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

Sélection complexe

Lorsque vous utilisez plusieurs critères, et que vous devez donc combiner plusieurs opérateurs logiques, il est extrêmement important de bien structurer la requête. En particulier, il faut placer des parenthèses au bon endroit. En effet, cela n'a pas de sens de mettre plusieurs opérateurs logiques différents sur un même niveau.
Petit exemple simple :
Critères : rouge AND vert OR bleu

Qu'accepte-t-on ?

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

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

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

Avec des parenthèses, pas moyen de se tromper sur ce qu'on désire sélectionner !

Exercice/Exemple

Alors, imaginons une requête bien tordue...

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

Je vous conseille d'essayer d'écrire cette requête tout seul. 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, y a plus qu'à écrire la requête avec les bons opérateurs logiques !

Secret (cliquez pour afficher)

Code : SQL

```
SELECT *
FROM animal
WHERE date_naissance > 2009-12-31
    OR
    ( espece='chat'
        AND
        ( sexe='male'
            OR
            ( sexe='femelle' AND date_naissance < 2007-06-01 )
        )
    );
```

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 sans doute été un peu étonné 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 :

Code : SQL

```
SELECT *
FROM Animal
WHERE nom = NULL; -- sélection des animaux sans nom
```

```sql
SELECT *
FROM Animal
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 :

Code : SQL

```sql
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
```

Cette fois-ci, ça fonctionne parfaitement !

Animaux sans noms

Code : Console

```
+----+--------+---------+---------------------+------+--------------+
| id | espece | sexe    | date_naissance      | nom  | commentaires |
+----+--------+---------+---------------------+------+--------------+
|  4 | tortue | femelle | 2009-08-03 05:12:00 | NULL | NULL         |
|  9 | tortue | NULL    | 2010-08-23 05:18:00 | NULL | NULL         |
+----+--------+---------+---------------------+------+--------------+
```

Animaux avec commentaires

Code : Console

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

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, ou par...
Bon, vous m'avez compris !

Eh bien 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 :

Code : SQL

```
SELECT *
FROM Animal
WHERE espece='chien'
ORDER BY date_naissance;
```

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

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, donc si vous ne précisez rien, c'est un tri ascendant qui est effectué. Donc 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 (donc pour la colonne `sexe ENUM('male','femelle')`, l'ordre par défaut (`ASC`) mettra les mâles en premier, ensuite les femelles).

Code : SQL

```
SELECT *
FROM Animal
WHERE espece='chien'
    AND nom IS NOT NULL
ORDER BY nom DESC;
```

Trier sur plusieurs colonnes

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

Code : SQL

```
SELECT *
FROM Animal
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.

Éliminer les doublons

Il peut arriver que MySQL vous donne plusieurs fois le même résultat. 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 :

Code : SQL

```
SELECT espece
FROM Animal;
```

Et 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.

Code : SQL

```
SELECT DISTINCT espece
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. Ca peut paraître logique, mais cela en perd plus d'un. Ce qui compte sont donc bien les colonnes que vous avez précisées dans votre **SELECT**. Uniquement espece donc, dans notre exemple.

Restreindre les résultats

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

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.

Code : SQL

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

Exemple

Code : SQL

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

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

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

Code : Console

```
+----+--------+---------+---------------------+----------------+-------------------
| id | espece | sexe    | date_naissance      | nom            | commentaires
+----+--------+---------+---------------------+----------------+-------------------
|  1 | chien  | male    | 2010-04-05 13:43:00 | Rox            | Mordille beaucoup
|  2 | chat   | NULL    | 2010-03-24 02:23:00 | Roucky         | NULL
|  3 | chat   | femelle | 2010-09-13 15:02:00 | Schtroumpfette | NULL
|  4 | tortue | femelle | 2009-08-03 05:12:00 | NULL           | NULL
|  5 | chat   | NULL    | 2010-10-03 16:44:00 | Choupi         | Né sans oreille ga
|  6 | tortue | femelle | 2009-06-13 08:17:00 | Bobosse        | Carapace bizarre
+----+--------+---------+---------------------+----------------+-------------------
```

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

Code : Console

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

Exemple avec un seul paramètre

Code : SQL

```sql
SELECT *
FROM Animal
ORDER BY id
LIMIT 10;
```

Cette requête est donc équivalente à :

Code : SQL

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

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

Code : SQL

```
SELECT *
FROM Animal
ORDER BY id
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.
Vous connaissez maintenant les bases de **SELECT**. Sachez que c'est une commande vraiment très puissante, et que ce que je vous ai fait faire pour l'instant n'est qu'une partie de la partie émergée de l'iceberg ! Mais je ne vous en dis pas plus, vous découvrirez tout ça dans le prochain chapitre (un peu) et dans la deuxième partie de ce tutoriel (beaucoup).

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 ordonner les résultats selon différents critères. Nous allons maintenant revenir plus particulièrement sur la clause **WHERE**. Jusque là, 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).

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 **NULL** aussi)

A 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 si l'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.

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, suiviees de zéro ou un caractère au choix puis se terminant par `'ne'` (`'chine'`, `'chne'`, `'echine'`, ...)

Rechercher '%' ou '_'

Comment faire si vous cherchez une chaîne de caractères contenant '%' ou '_' ? Evidemment, 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 :

Code : SQL

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

Résultat :

Code : Console

```
+----+--------+---------+---------------------+-------+---------------------------
| id | espece | sexe    | date_naissance      | nom   | commentaires
+----+--------+---------+---------------------+-------+---------------------------
| 42 | chat   | femelle | 2008-04-20 03:20:00 | Bilba | Sourde de l'oreille droite
+----+--------+---------+---------------------+-------+---------------------------
```

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 :

Code : SQL

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

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). Si vous désirez faire une recherche sensible à la casse, il vous faut définir votre chaîne de recherche comme une chaîne de type binaire, et non plus une simple chaîne de caractères :

Code : SQL

```sql
SELECT *
FROM Animal
WHERE nom LIKE '%Lu%'; -- insensible à la casse

SELECT *
FROM Animal
WHERE nom LIKE BINARY '%Lu%'; -- sensible à la casse
```

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 pur.

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

Code : SQL

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

Recherche dans un intervalle

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

Code : SQL

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

Ca 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 :

Code : SQL

```sql
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

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 :

Code : SQL

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

```
        OR nom = 'Gingko';
```

Un peu fastidieux non  ? Hé 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** :

Code : SQL

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

C'est quand même plus agréable à écrire ! 
Vous commencez à sentir la puissance de **SELECT** ? Et accrochez-vous, car ce n'est pas terminé !

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 lacher dans le chapitre de suppression de données sans que vous n'ayez la possibilité de faire un back-up de votre base. Je vous connais, vous allez faire des bêtises, et vous direz encore que c'est de ma faute...

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. Donc pas quand vous êtes connectés à MySQL. 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 :

Code : Console

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

Décortiquons cette commande.

- `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.

Lancez donc la commande suivante pour sauvegarder "elevage" dans votre dossier courant (c'est-à-dire sdzMysql normalement) :

Code : Console

```
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 sauver 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 la 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 sauvée. Donc, si vous effacez votre base par mégarde, il vous faut d'abord recréer la base de données (avec `CREATE DATABASE` nom_base), puis faire la commande suivante (dans la console) :

Code : Console

```
mysql nom_base < fichier_de_sauvegarde.sql
```

Donc concrètement, dans notre cas (à partir du dossier sdzMysql) :

Code : Console

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

Ou, directement à partir de MySQL :

Code : SQL

```
USE nom_base;
source 'fichier_de_sauvegarde.sql';
```

Donc :

Code : SQL

```
USE elevage;
source 'elevage_sauvegarde.sql';
```

Vous savez maintenant sauvegarder de manière simple vos base 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.

Suppression

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

Code : SQL

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

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

Code : SQL

```
DELETE FROM Animal
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**.

Code : SQL

```
DELETE FROM Animal;
```

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

Modification

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

Code : SQL

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

Par exemple, vous étiez persuadé 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 :

Code : SQL

```
UPDATE Animal
SET sexe='femelle', nom='Pataude'
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 changera donc le commentaire de tous les animaux stockés dans la table 'Animal'.

Code : SQL

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

N'hésitez pas à sauvegarder régulièrement votre base. C'est si facile avec mysql_dump, ce serait bête de s'en priver. Un jour ou l'autre on fait une fausse manœuvre et là... C'est le drame...

Et voilà, vous avez atteint la fin de cette première partie. Vous êtes maintenant capable de faire les manipulations de base sur les tables et les données. N'hésitez surtout pas à faire 36.000 essais pour bien vous familiariser avec ces premières commandes. Et quand vous serez prêts, rendez-vous dans la deuxième partie ! 

Partie 2 : Index, jointures et sous-requêtes

Cette partie est au moins aussi indispensable que la première. Vous y apprendrez ce que sont les clés et les index, et surtout la manipulation de plusieurs tables dans une même requête grâce aux jointures et aux sous-requêtes.

Index

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

Au programme de ce chapitre :

- Qu'est-ce qu'un index et comment est-il représenté ?
- En quoi un index peut-il accélérer une requête ?
- Quels sont les différents types d'index ?
- Comment créer (et supprimer) un index ?
- Qu'est-ce que la recherche `FULLTEXT` et comment fonctionne-t-elle ?

Qu'est-ce qu'un index ?

Revoyons la définition d'un index.

Citation : Définition

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

Lorsque vous créez un index sur une table, MySQL stocke cet index sous forme d'une table d'index, contenant les colonnes impliquées dans l'index en **triant les lignes** . Petit schéma explicatif dans le cas d'un index sur l'*id* de la table *Animal* (je ne prends que les neuf premières lignes pour ne pas surcharger).

id	Espèce	Sexe	Date de naissance	Nom	Commentaires
2	chat	NULL	2010-03-24 02:23:00	Roucky	NULL
1	chien	male	2010-04-05 13:43:00	Rox	Mordille beaucoup
3	chat	femelle	2010-09-13 15:02:00	Schtroumpfette	NULL
6	tortue	femelle	2009-06-13 08:17:00	Bobosse	Carapace bizarre
9	tortue	NULL	2010-08-23 05:18:00	NULL	NULL
4	tortue	femelle	2009-08-03 05:12:00	NULL	NULL
7	chien	femelle	2008-12-06 05:18:00	Caroline	NULL
8	chat	male	2008-09-11 15:38:00	Bagherra	NULL
5	chat	NULL	2010-10-03 16:44:00	Choupi	Né sans oreille gauche

Les données d'*Animal* ne sont pas stockées suivant un ordre intelligible pour nous. Par contre, l'index sur l'*id* est trié simplement par ordre croissant. Cela permet de grandement accélérer toute recherche faite sur cet id.

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

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

Comme vous pouvez le voir, l'ordre n'est pas du tout le même.

Intérêt des index

Vous devriez avoir compris maintenant que tout l'intérêt des index est d'accélérer les requêtes de sélection qui utilisent comme critères de recherche des colonnes indexées. Par conséquent, si vous savez que dans votre application, vous ferez énormément de recherches sur la colonne *X*, ajoutez donc un index sur cette colonne, vous ne vous en porterez que mieux.
Les index permettent aussi d'assurer l'intégrité des données de la base. Pour cela, il existe en fait plusieurs types d'index différents, et deux types de "clés". Lorsque je parle de garantir l'intégrité de vos données, cela signifie en gros garantir la qualité de vos données. S'assurer que vos données ont du sens. Par exemple, être sûr que vous ne faites pas référence à un client dans la table *Commande*, alors qu'en réalité, ce client n'existe absolument pas dans la table *Client*.

Désavantages

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

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

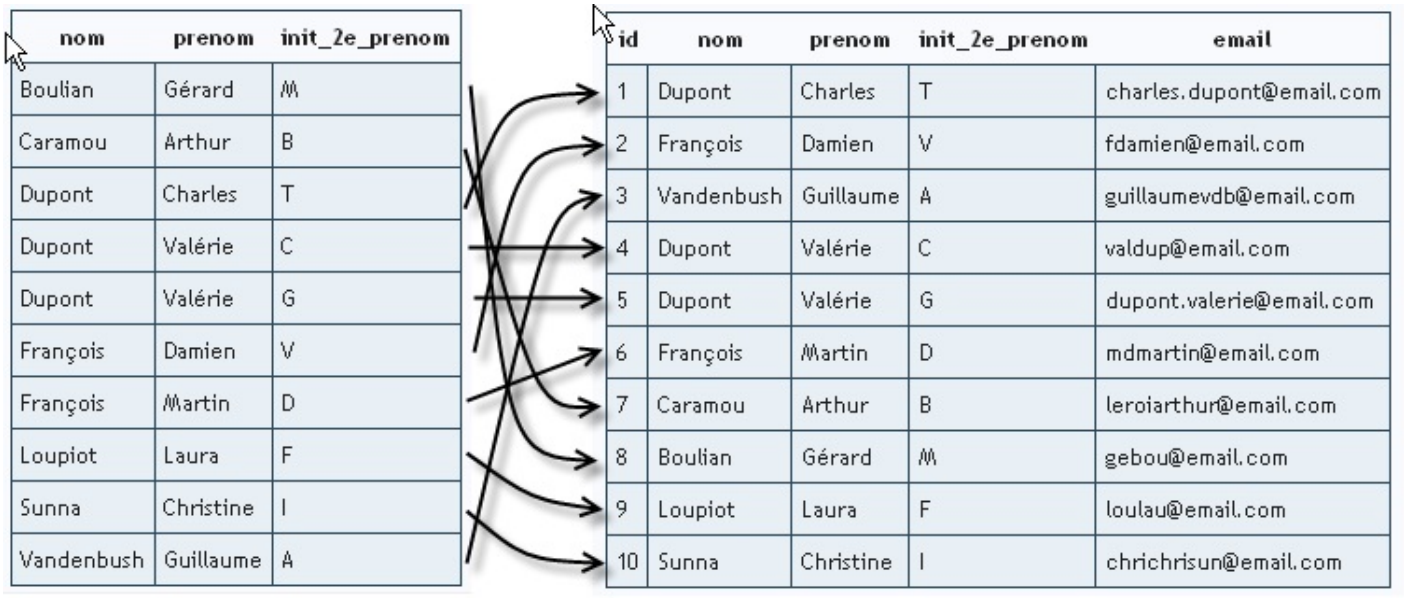
Par conséquent, n'ajoutez pas d'index lorsque ce n'est pas vraiment utile.

Index sur plusieurs colonnes

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

id	nom	prenom	init_2e_prenom	email
1	Dupont	Charles	T	charles.dupont@email.com
2	François	Damien	V	fdamien@email.com
3	Vandenbush	Guillaume	A	guillaumevdb@email.com
4	Dupont	Valérie	C	valdup@email.com
5	Dupont	Valérie	G	dupont.valerie@email.com
6	François	Martin	D	mdmartin@email.com
7	Caramou	Arthur	B	leroiarthur@email.com
8	Boulian	Gérard	M	gebou@email.com
9	Loupiot	Laura	F	loulau@email.com
10	Sunna	Christine	I	chrichrisun@email.com

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



nom	prenom	init_2e_prenom
Boulian	Gérard	M
Caramou	Arthur	B
Dupont	Charles	T
Dupont	Valérie	C
Dupont	Valérie	G
François	Damien	V
François	Martin	D
Loupiot	Laura	F
Sunna	Christine	I
Vandenbush	Guillaume	A

id	nom	prenom	init_2e_prenom	email
1	Dupont	Charles	T	charles.dupont@email.com
2	François	Damien	V	fdamien@email.com
3	Vandenbush	Guillaume	A	guillaumevdb@email.com
4	Dupont	Valérie	C	valdup@email.com
5	Dupont	Valérie	G	dupont.valerie@email.com
6	François	Martin	D	mdmartin@email.com
7	Caramou	Arthur	B	leroiarthur@email.com
8	Boulian	Gérard	M	gebou@email.com
9	Loupiot	Laura	F	loulau@email.com
10	Sunna	Christine	I	chrichrisun@email.com

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

Tirer parti des "index par la gauche"

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

prenom
Arthur
Charles
Christine
Damien
Gérard
Guillaume
Laura
Martin
Valérie
Valérie

nom
Boulian
Caramou
Dupont
Dupont
Dupont
François
François
Loupiot
Sunna
Vandenbush

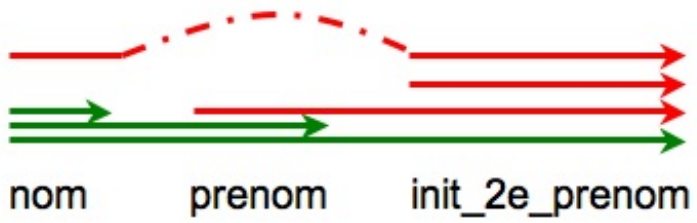
nom	prenom
Boulian	Gérard
Caramou	Arthur
Dupont	Charles
Dupont	Valérie
Dupont	Valérie
François	Damien
François	Martin
Loupiot	Laura
Sunna	Christine
Vandenbush	Guillaume

nom	prenom	init_2e_prenom
Boulian	Gérard	M
Caramou	Arthur	B
Dupont	Charles	T
Dupont	Valérie	C
Dupont	Valérie	G
François	Damien	V
François	Martin	D
Loupiot	Laura	F
Sunna	Christine	I
Vandenbush	Guillaume	A

Est-ce que vous remarquez quelque chose de spécial ?

Oui ! Bien vu ! La table de l'index (*nom, prenom*) correspond exactement aux deux premières colonnes de l'index (*nom, prenom, init_2e_prenom*) ; pas seulement les colonnes mais surtout l'ordre des lignes. Et l'index (*nom*) correspond à la première colonne de l'index (*nom, prenom, init_2e_prenom*).
Or, je vous ai dit que lorsque vous faites une recherche sur le nom, le prénom et l'initiale avec l'index (*nom, prenom, init_2e_prenom*), MySQL regarde d'abord le nom, puis le prénom, et pour finir l'initiale. Donc, si vous ne faites une recherche que sur le nom et le prénom, MySQL va intelligemment utiliser l'index (*nom, prenom, init_2e_prenom*). Il va simplement laisser tomber l'étape de l'initiale du second prénom. Idem si vous faites une recherche sur le nom : MySQL se basera uniquement sur la première colonne de l'index existant.
Par conséquent, pas besoin de définir un index (*nom, prenom*) ou un index (*nom*). Ils sont en quelque sorte déjà présents.

Mais qu'en est-il des index (*prenom*), ou (*prenom, init_2e_prenom*) ? Vous pouvez voir que la table contenant l'index (*prenom*) ne correspond à aucune colonne d'un index existant (au niveau de l'ordre des lignes). Par conséquent, si vous voulez un index sur (*prenom*), il vous faut le créer. Même chose pour (*prenom, init_2e_prenom*) ou (*nom, init_2e_prenom*).



On parle d'index "par la gauche". Donc si l'on prend des sous-parties d'index existant en prenant des colonnes "par la gauche", ces index existent. Mais si l'on commence par la droite ou que l'on "saute" une colonne, ils n'existent pas et doivent éventuellement être créés.

Index sur des colonnes de type alphanumérique

Types CHAR et VARCHAR

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

1e lettre	2e lettre	3e lettre	4e lettre	5e lettre	6e lettre	7e lettre	8e lettre	9e lettre	10e lettre

B	o	u	l	i	a	n			
C	a	r	a	m	o	u			
D	u	p	o	n	t				
D	u	p	o	n	t				
D	u	p	o	n	t				
F	r	a	n	ç	o	i	s		
F	r	a	n	ç	o	i	s		
L	o	u	p	i	o	t			
S	u	n	n	a					
V	a	n	d	e	n	b	u	s	h

nom
Boulian
Caramou
Dupont
Dupont
Dupont
François
François
Loupiot
Sunna
Vandenbush

En quoi cela nous intéresse-t-il ? C'est très simple. Ici, nous n'avons que des noms assez courts. La colonne *nom* peut par exemple être de type `VARCHAR(30)`. Mais imaginez une colonne de type `VARCHAR(150)`, qui contient des titres de livres par exemple. Si l'on met un index dessus, MySQL va indexer jusqu'à 150 caractères. Or, il est fort probable que les 25-30 premiers caractères du titre suffisent à trier ceux-ci. Au pire, un ou deux ne seront pas exactement à la bonne place, mais les requêtes en seraient déjà grandement accélérées.
Il serait donc plutôt pratique de dire à MySQL : "Indexe cette colonne, mais base-toi seulement sur les *x* premiers caractères". Et c'est possible évidemment (sinon je ne vous en parlerais pas), et c'est même très simple. Lorsque l'on créera l'index sur la colonne *titre_livre*, il suffira d'indiquer un nombre entre parenthèses : *titre_livre(25)* par exemple. Ce nombre étant bien sûr le nombre de caractères (dans le sens de la lecture donc à partir de la gauche) à prendre en compte pour l'index.

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

Types BLOB et TEXT (et dérivés)

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

Les différents types d'index

En plus des index "simples", que je viens de vous décrire, il existe trois types d'index qui ont des propriétés particulières. Les index **`UNIQUE`**, les index `FULLTEXT`, et enfin les index `SPATIAL`.
Je ne détaillerai pas les propriétés et utilisations des index `SPATIAL`. Sachez simplement qu'il s'agit d'un type d'index utilisé dans des bases de données recensant des données spatiales (tiens donc !), donc des points, des lignes, des polygones...

Sur ce, c'est parti !

Index UNIQUE

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

Par exemple, vous créez un site internet, et vous voulez le doter d'un espace membre. Chaque membre devra se connecter grâce à ses pseudo et mot de passe. Vous avez donc une table *Membre*, qui contient 4 colonnes : *id*, *pseudo*, *mot_de_passe* et *date_inscription*.
Deux membres peuvent avoir le même mot de passe, pas de problème. Par contre, que se passerait-il si deux membres avaient le même pseudo ? Lors de la connexion, il serait impossible de savoir quel mot de passe utiliser. Et sur quel compte connecter le membre.
Il faut donc absolument éviter que deux membres utilisent le même pseudo. Et pour cela, on va utiliser un index **UNIQUE** sur la colonne *pseudo* de la table *Membre*.

Autre exemple, dans notre table *Animal* cette fois. Histoire de ne pas confondre les animaux, vous prenez la décision de ne pas nommer de la même manière deux animaux de la même espèce. Il peut donc n'y avoir qu'une seule tortue nommée Toto. Un chien peut aussi se nommer Toto, mais un seul. La combinaison (espèce, nom) doit n'exister qu'une et une seule fois dans la table. Pour s'en assurer, il suffit de créer un index unique sur les colonnes *espece* et *nom* (un seul index, sur les deux colonnes).

Contraintes

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

Index FULLTEXT

Un index `FULLTEXT` est utilisé pour faire des recherches de manière puissante et rapide sur un texte. Seules les tables utilisant le moteur de stockage MyISAM peuvent avoir un index `FULLTEXT`, et cela uniquement sur les colonnes de type `CHAR`, `VARCHAR` ou `TEXT` (ce qui est plutôt logique).

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

Création et suppression des index

Les index sont représentés par le mot-clé **INDEX** (surprise ! ) ou **KEY** et peuvent être créés de deux manières :

- soit directement lors de la création de la table ;
- soit en les ajoutant par après.

Ajout des index lors de la création de la table

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

Index dans la description de la colonne

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

Je rappelle que la description de la colonne, c'est là que vous mettez le type de données, si la colonne peut contenir **NULL** ou pas, etc. Il est donc possible, à ce même endroit, de préciser si la colonne est un index.

Code : SQL

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

Quelques petites remarques.

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

Liste d'index

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

Code : SQL

```
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];
```

Par exemple, si l'on avait voulu créer la table *Animal* avec un index sur la date de naissance, et un autre sur les 10 premières lettres du prénom et l'espèce, on aurait pu utiliser la commande suivante :

Code : SQL

```
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;
```

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

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

Code : SQL

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

Avec cette syntaxe-ci, **KEY** ne peut être utilisé que pour les index simples, **INDEX** pour tous les types d'index. Bon, je sais que c'est assez pénible à retenir, mais en gros, utilisez **INDEX** partout, sauf pour définir un index dans la description même de la colonne, et vous n'aurez pas de problème.

Ajout des index après création de la table

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

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

Ajout avec ALTER TABLE

Code : SQL

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

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

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

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

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

Code : SQL

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

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

Ajout avec CREATE INDEX

La syntaxe de **CREATE INDEX** est très simple :

Code : SQL

```
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
```

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

Code : SQL

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

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

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

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

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

Code : SQL

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

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

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

Suppression d'un index

Rien de bien compliqué ici :

Code : SQL

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

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

Recherches avec FULLTEXT

Nous allons maintenant voir comment utiliser la recherche `FULLTEXT`, qui est un outil très puissant, et qui peut se révéler très utile.

Quelques rappels d'abord :

- 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`

Ça, c'est fait ! Nous allons maintenant passer à la recherche proprement dite, mais avant, je vais vous demander d'exécuter les instructions SQL suivantes, qui servent à créer la table que nous utiliserons pour illustrer ce chapitre. Nous sortons ici du contexte de l'élevage d'animaux. En effet, toutes les tables que nous créerons pour l'élevage utiliseront le moteur de stockage InnoDB.

Pour illuster la recherche `FULLTEXT`, je vous propose de créer la table *Livre*, contenant les colonnes *id* (clé primaire), *titre* et *auteur*. Les recherches se feront sur les colonnes *auteur* et *titre*, séparément ou ensemble. Il faut donc créer trois index `FULLTEXT` : (*auteur*), (*titre*) et (*auteur, titre*).

Secret (cliquez pour afficher)

Code : SQL

```
CREATE TABLE Livre (
 id INT PRIMARY KEY AUTO_INCREMENT,
 auteur VARCHAR(50),
 titre VARCHAR(200)
) ENGINE = MyISAM;
```

```
INSERT INTO Livre (auteur, titre)
VALUES ('Daniel Pennac', 'Au bonheur des ogres'),
('Daniel Pennac', 'La Fée Carabine'),
('Daniel Pennac', 'Comme un roman'),
('Daniel Pennac', 'La Petite marchande de prose'),
('Jacqueline Harpman', 'Le Bonheur est dans le crime'),
('Jacqueline Harpman', 'La Dormition des amants'),
('Jacqueline Harpman', 'La Plage d''Ostende'),
('Jacqueline Harpman', 'Histoire de Jenny'),
('Terry Pratchett', 'Les Petits Dieux'),
('Terry Pratchett', 'Le Cinquième éléphant'),
('Terry Pratchett', 'La Vérité'),
('Terry Pratchett', 'Le Dernier héros'),
('Terry Goodkind', 'Le Temple des vents'),
('Jules Verne', 'De la Terre à la Lune'),
('Jules Verne', 'Voyage au centre de la Terre'),
('Henri-Pierre Roché', 'Jules et Jim');

CREATE FULLTEXT INDEX ind_full_titre
ON Livre (titre);

CREATE FULLTEXT INDEX ind_full_aut
ON Livre (auteur);

CREATE FULLTEXT INDEX ind_full_titre_aut
ON Livre (titre, auteur);
```

Comment fonctionne la recherche FULLTEXT ?

Lorsque vous faites une recherche `FULLTEXT` sur une chaîne de caractères, cette chaîne est découpée en mots. Est considérée comme un mot : toute suite de caractères composée de lettres, chiffres, tirets bas _ et apostrophes '. Par conséquent, un mot composé, comme "porte-clés" par exemple, sera considéré comme deux mots : "porte" et "clés".

Lorsque MySQL compare la chaîne de caractères que vous lui avez donnée, et les valeurs dans votre table, il ne tient pas compte de tous les mots qu'il rencontre. Les règles sont les suivantes :

- les mots rencontrés dans au moins la moitié des lignes sont ignorés (règle des 50%) ;
- les mots trop courts (moins de quatre lettres) sont ignorés ;
- et les mots trop communs (en anglais, donc about, after, once, under...) ne sont également pas pris en compte.

Par conséquent, si vous voulez faire des recherches sur une table, il est nécessaire que cette table comporte au moins trois lignes, sinon chacun des mots sera présent dans au moins la moitié des lignes et aucun ne sera pris en compte.

Il est possible de redéfinir la longueur minimale des mots pris en compte, ainsi que la liste des mots trop communs. Je n'entrerai pas dans ces détails ici, vous trouverez ces informations dans la documentation officielle.

Les types de recherche

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

Recherche naturelle

Lorsque l'on fait une recherche naturelle, il suffit qu'un seul mot de la chaîne de caractères recherchée se retrouve dans une ligne pour que celle-ci apparaisse dans les résultats. Attention cependant que le mot **exact** doit se retrouver dans la valeur des colonnes de l'index `FULLTEXT` examiné.

Voici la syntaxe utilisée pour faire une recherche `FULLTEXT` :

Code : SQL

```
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 caractères
recherchée, entre guillemets bien sûr.
```

Si l'on veut préciser qu'on fait une recherche naturelle, on peut ajouter **IN NATURAL LANGUAGE MODE**. Ce n'est cependant pas obligatoire puisque la recherche naturelle est le mode de recherche par défaut.

Code : SQL

```
SELECT *
FROM nom_table
WHERE MATCH(colonne1[, colonne2, ...])
AGAINST ('chaîne recherchée' IN NATURAL LANGUAGE MODE);
```

Donc, par exemple, sur notre table *Livre*, on peut faire :

Code : SQL

```
SELECT *
FROM Livre
WHERE MATCH(auteur)
AGAINST ('Terry');
```

Résultat :

Code : Console

```
+----+-----------------+--------------------------+
| id | auteur          | titre                    |
+----+-----------------+--------------------------+
|  8 | Terry Pratchett | Les Petits Dieux         |
|  9 | Terry Pratchett | Le Cinquième éléphant    |
| 10 | Terry Pratchett | La Vérité                |
| 11 | Terry Pratchett | Le Dernier héros         |
| 12 | Terry Goodkind  | Le Temple des vents      |
+----+-----------------+--------------------------+
```

Code : SQL

```
SELECT *
FROM Livre
WHERE MATCH(titre)
AGAINST ('Petite');

SELECT *
FROM Livre
WHERE MATCH(titre)
AGAINST ('Petit');
```

Résultats :

Code : Console

```
#Première requête (avec "Petite")
+----+---------------+------------------------------+
| id | auteur        | titre                        |
+----+---------------+------------------------------+
|  3 | Daniel Pennac | La Petite marchande de prose |
+----+---------------+------------------------------+

#Seconde requête (avec "Petit")
Empty set (0.25 sec)
```

La deuxième requête (avec "Petit") ne renvoie aucun résultat. En effet, bien que "Petit" se retrouve deux fois dans la table (dans "La **Petit**e marchande de prose" et "Les **Petit**s Dieux"), il s'agit chaque fois d'une partie d'un mot, pas du mot exact.

Code : SQL

```
SELECT *
FROM Livre
WHERE MATCH(auteur)
AGAINST ('Henri');
```

Code : Console

```
+----+---------------------+--------------+
| id | auteur              | titre        |
+----+---------------------+--------------+
| 16 | Henri-Pierre Roché  | Jules et Jim |
+----+---------------------+--------------+
```

Ici par contre, on retrouve bien Henri-Pierre Roché en faisant une recherche sur "Henri", puisque Henri et Pierre sont considérés comme deux mots.

Code : SQL

```
SELECT *
FROM Livre
WHERE MATCH(auteur, titre)
AGAINST ('Jules');

SELECT *
FROM Livre
WHERE MATCH(titre, auteur)
AGAINST ('Jules Verne');
```

Code : Console

```
#Première requête (avec "Jules")
+----+--------------------+------------------------------+
| id | auteur             | titre                        |
```

```
+----+--------------------+------------------------------+
| 14 | Jules Verne        | De la Terre à la Lune        |
| 16 | Henri-Pierre Roché | Jules et Jim                 |
| 15 | Jules Verne        | Voyage au centre de la Terre |
+----+--------------------+------------------------------+

#Seconde requête (avec "Jules Verne")
+----+--------------------+------------------------------+
| id | auteur             | titre                        |
+----+--------------------+------------------------------+
| 14 | Jules Verne        | De la Terre à la Lune        |
| 15 | Jules Verne        | Voyage au centre de la Terre |
| 16 | Henri-Pierre Roché | Jules et Jim                 |
+----+--------------------+------------------------------+
```

Ces deux requêtes retournent les mêmes lignes. Vous pouvez donc voir que l'ordre des colonnes dans **MATCH** n'a aucune importance, du moment qu'un index FULLTEXT existe sur ces deux colonnes. Par ailleurs, la recherche se fait bien sur les deux colonnes, et sur chaque mot séparément, puisque les premières et troisièmes lignes contiennent 'Jules Verne' dans l'auteur, tandis que la deuxième contient uniquement 'Jules' dans le titre.

Par contre, l'ordre des lignes renvoyées par ces deux requêtes n'est pas le même. Lorsque vous utilisez **MATCH...** AGAINST dans une clause **WHERE**, les résultats sont par défaut triés par pertinence. Vous pouvez voir la pertinence attribuée à une ligne en mettant l'expression **MATCH...** AGAINST dans le **SELECT** :

Code : SQL

```
SELECT *, MATCH(titre, auteur) AGAINST ('Jules Verne Lune')
FROM Livre;
```

Résultat :

Code : Console

```
+----+--------------------+------------------------------+-------------------------
| id | auteur             | titre                        | MATCH(titre, auteur) AG
+----+--------------------+------------------------------+-------------------------
|  1 | Daniel Pennac      | Au bonheur des ogres         |
|  2 | Daniel Pennac      | La Fée Carabine              |
|  3 | Daniel Pennac      | La Petite marchande de prose |
|  4 | Jacqueline Harpman | Le Bonheur est dans le crime |
|  5 | Jacqueline Harpman | La Dormition des amants      |
|  6 | Jacqueline Harpman | La Plage d'Ostende           |
|  7 | Jacqueline Harpman | Histoire de Jenny            |
|  8 | Terry Pratchett    | Les Petits Dieux             |
|  9 | Terry Pratchett    | Le Cinquième éléphant        |
| 10 | Terry Pratchett    | La Vérité                    |
| 11 | Terry Pratchett    | Le Dernier héros             |
| 12 | Terry Goodkind     | Le Temple des vents          |
| 13 | Daniel Pennac      | Comme un roman               |
| 14 | Jules Verne        | De la Terre à la Lune        |
| 15 | Jules Verne        | Voyage au centre de la Terre |
| 16 | Henri-Pierre Roché | Jules et Jim                 |
+----+--------------------+------------------------------+-------------------------
```

Plus il y a de mots correspondant à la recherche, plus la ligne est pertinente. Vous aurez compris que lorsque vous faites une recherche (en mettant **MATCH...** AGAINST dans **WHERE** donc), les lignes avec une pertinence supérieure à 0 sont renvoyées.

Recherche avec booléens

La recherche avec booléens possède les caractéristiques suivantes :

- elle ne tient pas compte de la règle des 50%, qui veut qu'un mot présent dans 50% des lignes au moins soit ignoré ;
- elle peut se faire sur une ou des colonne(s) sur laquelle (lesquelles) aucun index `FULLTEXT` n'est défini (ce sera cependant beaucoup plus lent que si un index est présent) ;
- et les résultats ne seront pas triés par pertinence par défaut.

Pour faire une recherche avec booléens, il suffit d'ajouter **`IN BOOLEAN MODE`** après la chaîne recherchée.

Code : SQL

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

La recherche avec booléens permet d'être à la fois plus précis et plus approximatif dans ses recherches.

- Plus précis, car on peut **exiger** que certains mots se trouvent ou soient absents dans la ligne pour la sélectionner. On peut même exiger la présence de **groupes** de mots, plutôt que de rechercher chaque mot séparément.
- Plus approximatif, car on peut utiliser un astérisque * en fin de mot, pour préciser que le mot peut finir de n'importe quelle manière.

Pour exiger la présence ou l'absence de certains mots, on utilise les caractères + et -. Un mot précédé par + devra être présent dans la ligne, et précédé par - ne pourra pas être présent.

Exemple :

Code : SQL

```sql
SELECT *
FROM Livre
WHERE MATCH(titre)
AGAINST ('+bonheur -ogres' IN BOOLEAN MODE);
```

Code : Console

```
+----+--------------------+------------------------------+
| id | auteur             | titre                        |
+----+--------------------+------------------------------+
|  4 | Jacqueline Harpman | Le Bonheur est dans le crime |
+----+--------------------+------------------------------+
```

Seule une ligne est ici sélectionnée, bien que "bonheur" soit présent dans deux. En effet, le second livre dont le titre contient "bonheur" est "Le Bonheur des ogres", qui contient le mot interdit "ogres".

Pour spécifier un groupe de mots exigés, on utilise les doubles guillemets. Tous les mots entre doubles guillemets devront non seulement être présents mais également apparaître dans l'ordre donné, et sans rien entre eux. Il faudra donc que l'on retrouve **exactement** ces mots pour avoir un résultat.

Code : SQL

```sql
SELECT *
FROM Livre
WHERE MATCH(titre)
AGAINST ('"Terre à la Lune"' IN BOOLEAN MODE);

SELECT *
FROM Livre
WHERE MATCH(titre)
AGAINST ('"Lune à la Terre"' IN BOOLEAN MODE);

SELECT *
FROM Livre
WHERE MATCH(titre)
AGAINST ('"Terre la Lune"' IN BOOLEAN MODE);
```

Code : Console

```
#Première requête (avec "Terre à la Lune")
+----+-------------+------------------------+
| id | auteur      | titre                  |
+----+-------------+------------------------+
| 14 | Jules Verne | De la Terre à la Lune  |
+----+-------------+------------------------+

#Seconde requête (avec "Lune à la Terre")
Empty set (0.05 sec)

#Troisième requête (avec "Terre la Lune")
Empty set (0.05 sec)
```

La première requête renverra bien un résultat, contrairement à la seconde (car les mots ne sont pas dans le bon ordre) et à la troisième (car il manque le "à" dans la recherche - ou il y a un "à" en trop dans la ligne, ça dépend du point de vue).
"Voyage au centre de la Terre" n'est pas un résultat puisque seul le mot "Terre" est présent.

Et pour utiliser l'astérisque, il suffit d'écrire le début du mot dont on est sûr, et de compléter avec un astérisque.

Code : SQL

```sql
SELECT *
FROM Livre
WHERE MATCH(titre)
AGAINST ('petit*' IN BOOLEAN MODE);
```

Code : Console

```
+----+-----------------+------------------------------+
| id | auteur          | titre                        |
+----+-----------------+------------------------------+
|  3 | Daniel Pennac   | La Petite marchande de prose |
|  8 | Terry Pratchett | Les Petits Dieux             |
+----+-----------------+------------------------------+
```

"Petite" et "Petits" sont trouvés.

Code : SQL

```sql
SELECT *
FROM Livre
WHERE MATCH(titre, auteur)
AGAINST ('d*' IN BOOLEAN MODE);
```

Code : Console

```
+----+--------------------+------------------------------+
| id | auteur             | titre                        |
+----+--------------------+------------------------------+
|  1 | Daniel Pennac      | Au bonheur des ogres         |
|  2 | Daniel Pennac      | La Fée Carabine              |
|  3 | Daniel Pennac      | La Petite marchande de prose |
| 13 | Daniel Pennac      | Comme un roman               |
|  4 | Jacqueline Harpman | Le Bonheur est dans le crime |
| 11 | Terry Pratchett    | Le Dernier héros             |
|  8 | Terry Pratchett    | Les Petits Dieux             |
|  5 | Jacqueline Harpman | La Dormition des amants      |
+----+--------------------+------------------------------+
```

Chacun des résultats contient au moins un mot commençant par "d" dans son titre ou son auteur ("Daniel", "Dormition"...). Mais qu'en est-il de "Voyage au centre de la Terre" par exemple ? Avec le "de", il aurait dû être sélectionné. Mais c'est sans compter la règle qui dit que les mots de moins de quatre lettres sont ignorés. "De" n'ayant que deux lettres, ce résultat est ignoré.

Pour en finir avec la recherche avec booléens, sachez qu'il est bien sûr possible de mixer ces différentes possibilités. Les combinaisons sont nombreuses.

Code : SQL

```sql
SELECT *
FROM Livre
WHERE MATCH(titre)
AGAINST ('+petit* -prose' IN BOOLEAN MODE); -- mix d'une astérisque
avec les + et -
```

Code : Console

```
+----+-----------------+------------------+
| id | auteur          | titre            |
+----+-----------------+------------------+
|  8 | Terry Pratchett | Les Petits Dieux |
+----+-----------------+------------------+
```

Recherche avec extension de requête

Le dernier type de recherche est un peu particulier. En effet la recherche avec extension de requête se déroule en deux étapes.

1. Une simple recherche naturelle est effectuée.
2. Les résultats de cette recherche sont utilisés pour faire une seconde recherche naturelle.

Un exemple étant souvent plus clair qu'un long discours, passons-y tout de suite.

Une recherche naturelle effectuée avec la chaîne "Daniel" sur les colonnes auteur et titre donnerait ceci :

Code : SQL

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

Code : Console

```
+----+---------------+-----------------------------+
| id | auteur        | titre                       |
+----+---------------+-----------------------------+
|  2 | Daniel Pennac | La Fée Carabine             |
|  1 | Daniel Pennac | Au bonheur des ogres        |
| 13 | Daniel Pennac | Comme un roman              |
|  3 | Daniel Pennac | La Petite marchande de prose |
+----+---------------+-----------------------------+
```

Par contre, si l'on utilise l'extension de requête, en ajoutant `WITH QUERY EXPANSION`, on obtient ceci :

Code : SQL

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

Code : Console

```
+----+--------------------+-----------------------------+
| id | auteur             | titre                       |
+----+--------------------+-----------------------------+
|  3 | Daniel Pennac      | La Petite marchande de prose |
| 13 | Daniel Pennac      | Comme un roman              |
|  1 | Daniel Pennac      | Au bonheur des ogres        |
|  2 | Daniel Pennac      | La Fée Carabine             |
|  4 | Jacqueline Harpman | Le Bonheur est dans le crime |
+----+--------------------+-----------------------------+
```

En effet, dans la seconde étape, une recherche naturelle a été faite avec les chaînes "Daniel Pennac", "La Petite marchande de prose", "Comme un roman", "Au bonheur des ogres" et "La Fée Carabine", puisque ce sont les résultats de la première étape (une recherche naturelle sur "Daniel").
"Le Bonheur est dans le crime" a donc été ajouté aux résultats, à cause de la présence du mot "bonheur" dans son titre ("Bonheur" étant également présent dans "Au bonheur des ogres")

Ainsi s'achève la présentation des requêtes `FULLTEXT`, ainsi que le chapitre sur les index.
Ce fut fort théorique, mais c'est important donc si vous n'avez pas tout compris, prenez soin de le relire ! D'autant plus que le prochain chapitre est complémentaire avec celui-ci.

Clés primaires et étrangères

Maintenant que les index n'ont plus de secret pour vous, nous allons passer à une autre notion très importante : les clés.
Les clés sont, vous allez le voir, intimement liées aux index. Et tout comme `NOT NULL` et les index `UNIQUE`, les clés font partie de ce qu'on appelle les **contraintes**.
Il existe deux types de clés :

- les **clés primaires**, qui ont déjà été survolées lors du chapitre sur la création d'une table, et qui servent à **identifier une ligne** de manière unique ;
- les **clés étrangères**, qui permettent de gérer des **relations entre plusieurs tables**, et garantissent la **cohérence des données**.

Il s'agit à nouveau d'un chapitre avec beaucoup de blabla, mais je vous promets qu'après celui-ci, on s'amusera à nouveau ! Donc un peu de courage. 

Clés primaires, le retour

Les clés primaires ont déjà été introduites dans le chapitre de création des tables. Je vous avais à l'époque donné la définition suivante :

La clé primaire d'une table est une contrainte d'unicité, composée d'une ou plusieurs colonne(s), et qui permet d'identifier de manière unique chaque ligne de la table.

Examinons plus en détail cette définition.

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

Ces quelques considérations résument très bien l'essence des clés primaires. En gros, une clé primaire est un index `UNIQUE` sur un colonne qui ne peut pas être `NULL`.
D'ailleurs, vous savez déjà que l'on définit une clé primaire grâce aux mots-clés `PRIMARY KEY`. Or, nous avons vu dans le précédent chapitre que `KEY` s'utilise pour définir un index. Par conséquent, lorsque vous définissez une clé primaire, pas besoin de définir en plus un index sur la (les) colonne(s) qui compose(nt) celle-ci, c'est déjà fait ! Et pas besoin non plus de rajouter une contrainte `NOT NULL`
Pour le dire différemment, une contrainte de clé primaire est donc une combinaison de deux des contraintes que nous avons vues jusqu'à présent : `UNIQUE` et `NOT NULL`.

Choix de la clé primaire

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

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

Par conséquent, on est bien souvent obligé d'ajouter une colonne pour jouer le rôle de la clé primaire. Cette fameuse colonne *id*, auto-incrémentée, comme nous l'avons fait pour la table *Animal*.

Il y a une autre raison à l'utilisation d'une colonne spéciale auto-incrémentée, de type `INT` (ou un de ses dérivés) pour la clé primaire. En effet, lorsque l'on définit une clé primaire, c'est notamment dans le but d'utiliser un maximum cette clé pour faire des recherches dans la table. Bien sûr, parfois ce n'est pas possible, parfois vous ne connaissez pas l'id du client, et vous êtes obligé de faire une recherche par nom. Cependant, vous verrez bientôt que les clés primaires peuvent servir à faire des recherches de manière **indirecte** sur la table. Du coup, comme les recherches sont beaucoup plus rapides sur des nombres que sur des textes, il est plus intéressant d'avoir une clé primaire composée de colonnes de type `INT`.

Et enfin, il y a également l'argument de l'auto-incrémentation. Si vous devez remplir vous-même la colonne de la clé primaire, étant donné que vous êtes humain (comment ça pas du tout ?), vous risquez de faire une erreur. Or avec une clé primaire auto-incrémentée, vous ne risquez rien. MySQL fait tout pour vous. Et on ne peut définir une colonne comme auto-incrémentée que si elle est de type `INT` et qu'il existe un index dessus.

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

PRIMARY KEY or not PRIMARY KEY

Je me dois de vous dire que d'un point de vue technique, avoir une clé primaire sur chaque table n'est pas obligatoire. Vous pourriez travailler toute votre vie sur une base de données sans aucune clé primaire, et ne jamais voir un message d'erreur à ce propos.
Cependant, d'un point de vue **conceptuel**, ce serait une grave erreur. Ce n'est pas le propos de ce tutoriel que de vous enseigner les étapes de conception d'une base de données, mais je vous en prie, pensez à mettre une clé primaire sur chacune de vos tables. Si leur utilité n'est pas complètement évidente pour vous en ce moment, elle devrait le devenir au fur et à mesure de votre lecture.

Création d'une clé primaire

La création des clés primaires étant extrêmement semblable à la création d'index simples, j'espère que vous me pardonnerez si je ne détaille pas trop mes explications. 

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

Lors de la création de la table

Soit on précise **`PRIMARY KEY`** dans la description de la colonne qui doit devenir la clé primaire (pas de clé composite dans ce cas)

Code : SQL

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

Donc par exemple, pour la table *Animal* :

Code : SQL

```
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
)
ENGINE=InnoDB;
```

Ou alors, on ajoute la clé à la suite des colonnes, comme on l'avait fait d'ailleurs pour la table Animal. Cette méthode permet bien sûr la création d'une clé composite (donc avec plusieurs colonnes).

Code : SQL

```
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

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

Si vous créez une clé primaire sur une table existante, assurez-vous que la (les) colonne(s) où vous souhaitez l'ajouter ne contienne(nt) pas **NULL**.

Code : SQL

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

Suppression de la clé primaire

Code : SQL

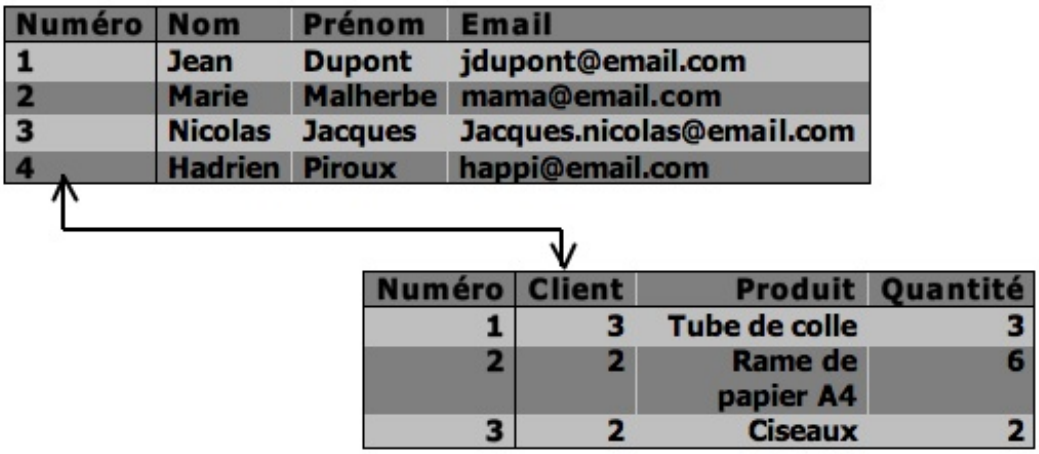
```
ALTER TABLE nom_table
DROP PRIMARY KEY
```

Pas besoin de préciser de quelle clé il s'agit, puisqu'il ne peut y en avoir qu'une seule par table !

Clés étrangères

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

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



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

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

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

Ce serait quand même sympa si à l'insertion d'une ligne dans la table *Commande*, un gentil petit lutin allait vérifier que le numéro de client indiqué correspond bien à quelque chose dans la table *Client*, non ?

Mesdames et messieurs, ce lutin existe ! C'est en fait une lutine et elle s'appelle "clé étrangère" (vous voyez bien que vos parent ont choisi un super prénom pour vous).

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

Avant d'entamer une danse de la joie, parce que quand même, MySQL, c'est beaucoup trop cool, restez concentrés cinq minutes, le temps de lire et retenir les points (**importants**) suivants :

- comme pour les index et les clés primaires, il est possible de créer des clés étrangères composites ;
- lorsque vous créez une clé étrangère sur une colonne (ou un groupe de colonnes) - la colonne *client* de *Commande* dans notre exemple -, un index est automatiquement ajouté sur celle-ci (ou sur le groupe) ;
- par contre, la colonne (le groupe de colonnes) qui sert de référence - la colonne *numéro* de *Client* - **doit** déjà posséder un index (où être clé primaire bien sûr) ;
- et enfin, la colonne (ou le groupe de colonnes) sur laquelle (lequel) la clé est créée doit être **exactement** du même type que la colonne (le groupe de colonnes) qu'elle (il) référence. Cela implique qu'en cas de clé composite, il faut bien sûr le même nombre de colonnes dans la clé et la référence. Donc si *numéro* (dans *Client*) est un `INT`, *client* (dans *Commande*) doit être de type `INT` aussi.

Création

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

- la ou les colonne(s) sur laquelle (lesquelles) on crée la clé - on utilise **`FOREIGN KEY`** ;
- la ou les colonne(s) qui va (vont) servir de référence - on utilise **`REFERENCES`**.

Lors de la création de la table

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

Code : SQL

```
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];
```

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

Code : SQL

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

Après création de la table

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

Code : SQL

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

Suppression d'une clé étrangère

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

Code : SQL

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

Modification de notre base

Maintenant que vous connaissez les clés étrangères, nous allons en profiter pour modifier notre base, et ajouter quelques tables afin de préparer le prochain chapitre, qui portera donc sur les jointures.

Jusqu'à maintenant, la seule information sur l'espèce des animaux de notre élevage était son nom courant. Nous voudrions maintenant aussi stocker son nom latin, ainsi qu'une petite description. Que faire ? Ajouter deux colonnes à notre table ? *nom_latin_espece* et *description_espece* ?
J'espère que vous n'aurez envisagé cette possibilité qu'une demi-seconde car il est assez évident que c'est une très mauvaise solution. En effet, ça obligerait à stocker la même description pour chaque chien, chaque chat, etc. Ainsi que le même nom latin. Nous le faisions déjà avec le nom courant, et ça aurait déjà pu poser problème. Imaginez que pour un animal vous fassiez une faute d'orthographe au nom de l'espèce, "chein" ou lieu de "chien" par exemple. L'animal en question n'apparaîtrait jamais lorsque vous feriez une recherche par espèce.
Il faudrait donc un système qui nous permette de ne pas répéter la même information plusieurs fois, et qui limite les erreurs que l'on pourrait faire.

La bonne solution est de créer une seconde table : la table *Espece*. Cette table aura 4 colonnes : le nom courant, le nom latin, une description, et un numéro d'identification (qui sera la clé primaire de cette table).

La table Espece

Voici donc la table que je vous propose de créer :

Grâce à la commande suivante :

Code : SQL

```
CREATE TABLE Espece (
    id SMALLINT AUTO_INCREMENT,
    nom_courant VARCHAR(40) NOT NULL,
    nom_latin VARCHAR(40) NOT NULL UNIQUE,
    description TEXT,
    PRIMARY KEY(id)
);
```

J'ai mis un index unique sur la colonne nom_latin pour être sûre que l'on ne rentrera pas deux fois la même espèce. Pourquoi sur le nom latin et pas le nom courant ? Tout simplement car le nom latin est beaucoup plus rigoureux et règlementé que le nom courant. On peut avoir plusieurs dénominations courantes pour une même espèce, ce n'est pas le cas avec le nom latin.

Bien, remplissons donc cette table avec les espèces déjà présentes dans la base :

Code : SQL

```
INSERT INTO Espece (nom_courant, nom_latin, description) VALUES
    ('Chien', 'Canis canis', 'Bestiole à quatre pattes qui aime les
caresses et tire souvent la langue'),
    ('Chat', 'Felis silvestris', 'Bestiole à quatre pattes qui saute
très haut et grimpe aux arbres'),
    ('Tortue d''Hermann', 'Testudo hermanni', 'Bestiole avec une
carapace très dure'),
    ('Perroquet amazone', 'Alipiopsitta xanthops', 'Joli oiseau
parleur vert et jaune');
```

Ce qui nous donne la table suivante :

Code : Console

```
+----+-------------------+-----------------------+--------------------------------
| id | nom_courant       | nom_latin             | description
+----+-------------------+-----------------------+--------------------------------
|  1 | Chien             | Canis canis           | Bestiole à quatre pattes qui aim
|  2 | Chat              | Felis silvestris      | Bestiole à quatre pattes qui sau
|  3 | Tortue d'Hermann  | Testudo hermanni      | Bestiole avec une carapace très
|  4 | Perroquet amazone | Alipiopsitta xanthops | Joli oiseau parleur vert et jaun
+----+-------------------+-----------------------+--------------------------------
```

Vous aurez remarqué que j'ai légèrement modifié les noms des espèces "perroquet" et "tortue". En effet, et j'espère que les biologistes pourront me pardonner, "perroquet" et "tortue" ne sont pas des espèces, mais des ordres. J'ai donc précisé un peu (si je donne juste l'ordre, c'est comme si je mettais "carnivore" au lieu de "chat" - ou "chien" d'ailleurs -).

Bien, mais cela ne suffit pas ! Il faut également modifier notre table *Animal*. Nous allons ajouter une colonne *espece_id*, qui contiendra l'*id* de l'espèce à laquelle appartient l'animal, et remplir cette colonne. Ensuite nous pourrons supprimer la colonne *espece*, qui n'aura plus de raison d'être.
La colonne *espece_id* sera une clé étrangère ayant pour référence la colonne *id* de la table *Espece*. Je rappelle donc que ça signifie qu'il ne sera pas possible d'insérer dans *espece_id* un nombre qui n'existe pas dans la colonne *id* de la table *Espece*.

La table Animal

Les commandes utilisées sont chaque fois mises dans des balises à **<secret></secret>**. Essayez donc d'écrire vous-même la requête avant de regarder comment je fais.

Ajout d'une colonne espece_id

Secret (cliquez pour afficher)

Code : SQL

```sql
ALTER TABLE Animal
ADD COLUMN espece_id SMALLINT;
```

La colonne *espece_id* doit avoir le même type que la colonne *id* de la table *Espece*, donc `SMALLINT`.

Remplissage de espece_id

Secret (cliquez pour afficher)

Code : SQL

```sql
UPDATE Animal
SET espece_id = 1
WHERE espece = 'chien';

UPDATE Animal
SET espece_id = 2
WHERE espece = 'chat';
```

```
UPDATE Animal
SET espece_id = 3
WHERE espece = 'tortue';

UPDATE Animal
SET espece_id = 4
WHERE espece = 'perroquet';
```

J'espère que dans votre enthousiasme débordant, vous n'avez pas supprimé la colonne *espece* avant de remplir *espece_id* ...

Suppression de la colonne espece

Secret (cliquez pour afficher)

Code : SQL

```
ALTER TABLE Animal
DROP COLUMN espece;
```

Rien de spécial ici.

Ajout de la clé étrangère

Secret (cliquez pour afficher)

Code : SQL

```
ALTER TABLE Animal
ADD CONSTRAINT fk_espece_id
FOREIGN KEY (espece_id)
REFERENCES Espece(id);
```

Pour tester l'efficacité de la clé étrangère, essayons d'ajouter un animal dont l'*espece_id* est 5 (et donc qui n'existe pas dans la table *Espece*) :

Code : SQL

```
INSERT INTO Animal (nom, espece_id, date_naissance)
VALUES ('Caouette', 5, '2009-02-15 12:45:00');
```

Résultat :

Code : Console

```
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fail
```

Elle est pas belle la vie ?

J'en profite également pour ajouter une contrainte **NOT NULL** sur la colonne *espece_id*. Après tout, si *espece* ne pouvait pas être **NULL**, pas de raison qu'*espece_id* puisse !

Secret (cliquez pour afficher)

Code : SQL

```sql
ALTER TABLE Animal
MODIFY espece_id SMALLINT NOT NULL;
```

Notez qu'il n'était pas possible de mettre cette contrainte à la création de la colonne, puisque tant que l'on n'avait pas rempli *espece_id*, elle contenait **NULL** pour toutes les lignes.

Voilà, nos deux tables sont maintenant prêtes. Mais avant de vous lâcher dans l'univers merveilleux des jointures et des sous-requêtes, nous allons encore compliquer un peu les choses. Parce que c'est bien joli pour un éleveur de pouvoir reconnaître un chien d'un chat, mais il serait de bon ton de reconnaître également un berger allemand d'un teckel, non ?

Par conséquent, nous allons encore ajouter deux choses à notre base. D'une part, nous allons ajouter une table Race, basée sur le même schéma que la table Espece. Il faudra donc également ajouter une colonne à la table *Animal*. Contrairement à la colonne *espece_id*, celle-ci pourra être **NULL**. Il n'est pas impossible que nous accueillons des bâtards, ou que certaines espèces que nous élevons ne soient pas classées en plusieurs races différentes.
Et ensuite, nous allons garder une trace du pedigree de nos animaux. Pour ce faire, il faut pouvoir connaître ses parents. Donc, nous ajouterons deux colonnes à la table *Animal* : *pere_id* et *mere_id*.

Ce sont toutes des commandes que vous connaissez, donc je ne détaille plus tout.

Code : SQL

```sql
-- --------------------------
-- CREATION DE LA TABLE Race
-- --------------------------
CREATE TABLE Race (
    id SMALLINT AUTO_INCREMENT,
    nom VARCHAR(40) NOT NULL,
    espece_id SMALLINT,             -- pas de nom latin, mais une
référence vers l'espèce
    description TEXT,
    PRIMARY KEY(id),
    CONSTRAINT fk_race_espece_id FOREIGN KEY (espece_id) REFERENCES
Espece(id)  -- pour assurer l'intégrité de la référence
);

-- ----------------------
-- REMPLISSAGE DE LA TABLE
-- ----------------------
INSERT INTO Race (nom, espece_id, description)
VALUES ('Berger allemand', 1, 'Chien sportif et élégant au pelage
dense, noir-marron-fauve, noir ou gris.'),
('Berger blanc suisse', 1, 'Petit chien au corps compact, avec des
pattes courtes mais bien proportionnées et au pelage tricolore ou
bicolore.'),
('Boxer', 1, 'Chien de taille moyenne, au poil ras de couleur fauve
ou bringé avec quelques marques blanches.'),
('Bleu russe', 2, 'Chat aux yeux verts et à la robe épaisse et
argentée.'),
('Maine coon', 2, 'Chat de grande taille, à poils mi-longs.'),
('Singapura', 2, 'Chat de petite taille aux grands yeux en
amandes.'),
('Sphynx', 2, 'Chat sans poils.');

-- ----------------------------------------------
```

```
-- AJOUT DE LA COLONNE race_id A LA TABLE Animal
-- ---------------------------------------------
ALTER TABLE Animal
ADD COLUMN race_id SMALLINT;

ALTER TABLE Animal
ADD CONSTRAINT fk_race_id
FOREIGN KEY (race_id)
REFERENCES Race(id);

-- -------------------------
-- REMPLISSAGE DE LA COLONNE
-- -------------------------
UPDATE Animal SET race_id = 1 WHERE id IN (1, 13, 20, 18, 22, 25,
26, 28);
UPDATE Animal SET race_id = 2 WHERE id IN (12, 14, 19, 7);
UPDATE Animal SET race_id = 3 WHERE id IN (23, 17, 21, 27);
UPDATE Animal SET race_id = 4 WHERE id IN (33, 35, 37, 41, 44, 31,
3);
UPDATE Animal SET race_id = 5 WHERE id IN (43, 40, 30, 32, 42, 34,
39, 8);
UPDATE Animal SET race_id = 6 WHERE id IN (29, 36, 38);

-- ------------------------------------------------------
-- AJOUT DES COLONNES mere_id et pere_id A LA TABLE Animal
-- ------------------------------------------------------
ALTER TABLE Animal
ADD COLUMN mere_id SMALLINT;

ALTER TABLE Animal
ADD CONSTRAINT fk_mere_id
FOREIGN KEY (mere_id)
REFERENCES Animal(id);

ALTER TABLE Animal
ADD COLUMN pere_id SMALLINT;

ALTER TABLE Animal
ADD CONSTRAINT fk_pere_id
FOREIGN KEY (pere_id)
REFERENCES Animal(id);

-- ----------------------------------------
-- REMPLISSAGE DES COLONNES mere_id ET pere_id
-- ----------------------------------------
UPDATE Animal SET mere_id = 18, pere_id = 22 WHERE id = 1;
UPDATE Animal SET mere_id = 7, pere_id = 21 WHERE id = 10;
UPDATE Animal SET mere_id = 41, pere_id = 31 WHERE id = 3;
UPDATE Animal SET mere_id = 40, pere_id = 30 WHERE id = 2;
```

A priori, la seule chose qui pourrait vous avoir surpris dans ces requêtes sont les clés étrangères sur mere_id et pere_id qui référencent toutes deux une autre colonne de la même table.

Bien, maintenant que nous avons trois tables et des données sympathiques à exploiter, nous allons passer aux choses sérieuses avec les jointures d'abord, puis les sous-requêtes.
Vous savez maintenant créer des clés et des index, et devriez être capable de les créer intelligemment. J'ai volontairement placé ce chapitre avant d'aborder les jointures, bien que ce ne soit pas indispensable, car je pense qu'il est important que ces notions soient bien claires avant de commencer à travailler avec plusieurs tables. Votre base de données n'en sera que plus propre, mieux conçue, et plus optimisée.

Jointures

Vous vous attaquez maintenant au plus important chapitre de cette partie. Le principe des jointures est plutôt simple à comprendre (quoiqu'on puisse faire des requêtes très compliquées avec), et totalement indispensable.

Les jointures vont vous permettre de jongler avec **plusieurs tables** dans la même requête.
Pour commencer, nous verrons le principe général des jointures. Puis nous ferons un petit détour par les alias, qui servent beaucoup dans les jointures (mais pas seulement). Ensuite, retour sur le sujet avec les deux types de jointures : **internes** et **externes**. Et pour finir, après un rapide tour d'horizon des syntaxes possibles pour faire une jointure, je vous propose quelques exercices pour mettre tout ça en pratique.

Principe des jointures et notion d'alias

Principe des jointures

Sans surprise, le principe des jointures est de ... joindre plusieurs tables. Pour ce faire, on utilise les informations communes des tables.

Par exemple, lorsque nous avons ajouté dans notre base les informations sur les espèces, - leur nom latin et leur description - je vous ai dit que ce serait une très mauvaise idée de tout mettre dans la table *Animal*, car il nous faudrait alors répéter la même description pour tous les chiens, la même pour toutes les tortues, etc.
Cependant, vous avez sans doute remarqué que du coup, si vous voulez afficher la description de l'espèce de Cartouche (votre petit préféré), vous avez besoin de deux requêtes.

Code : SQL

```sql
SELECT espece_id FROM Animal WHERE nom = 'Cartouche';
```

Code : Console

```
+-----------+
| espece_id |
+-----------+
|         1 |
+-----------+
```

Code : SQL

```sql
SELECT description FROM Espece WHERE id = 1;
```

Code : Console

```
+-----------------------------------------------------------------------------+
| description                                                                 |
+-----------------------------------------------------------------------------+
| Bestiole à quatre pattes qui aime les caresses et tire souvent la langue |
+-----------------------------------------------------------------------------+
```

Ne serait-ce pas merveilleux de pouvoir faire tout ça (et plus encore) en une seule requête ? Et c'est là que les jointures entrent en jeu, pour nous le permettre. Pour l'exemple ci-dessus, on va donc utiliser l'information commune entre les deux tables : l'id de l'espèce, qui est présente dans *Animal* avec la colonne *espece_id*, et dans *Espece* avec la colonne *id*.

Code : SQL

```
SELECT Espece.description
FROM Espece
INNER JOIN Animal
    ON Espece.id = Animal.espece_id
WHERE Animal.nom = 'Cartouche';
```

Tataaaaaaaam :

Code : Console

```
+-----------------------------------------------------------------------------+
| description                                                                 |
+-----------------------------------------------------------------------------+
| Bestiole à quatre pattes qui aime les caresses et tire souvent la langue |
+-----------------------------------------------------------------------------+
```

En fait, lorsque l'on fait une jointure, on crée une table virtuelle et temporaire qui reprend les colonnes des tables liées. Donc, on part de ces deux tables-ci (je ne reprends qu'une partie de la table *Animal*) :

Code : Console

```
+----+---------+---------------------+----------+-------------------------+--------
| id | sexe    | date_naissance      | nom      | commentaires            | espece_
+----+---------+---------------------+----------+-------------------------+--------
|  5 | NULL    | 2010-10-03 16:44:00 | Choupi   | Né sans oreille gauche |
|  8 | male    | 2008-09-11 15:38:00 | Bagherra | NULL                    |
|  9 | NULL    | 2010-08-23 05:18:00 | NULL     | NULL                    |
| 14 | femelle | 2009-05-26 08:56:00 | Fila     | NULL                    |
| 16 | femelle | 2009-05-26 08:50:00 | Louya    | NULL                    |
| 18 | femelle | 2007-04-24 12:59:00 | Zira     | NULL                    |
| 24 | male    | 2007-04-12 05:23:00 | Cartouche | NULL                    |
| 25 | male    | 2006-05-14 15:50:00 | Zambo    | NULL                    |
| 33 | male    | 2006-05-19 16:17:00 | Caribou  | NULL                    |
| 52 | femelle | 2006-03-15 14:26:00 | Redbul   | Insomniaque             |
| 58 | male    | 2008-02-20 02:50:00 | Gingko   | NULL                    |
+----+---------+---------------------+----------+-------------------------+--------


+----+-------------------+-----------------------+-----------------------------------
| id | nom_courant       | nom_latin             | description
+----+-------------------+-----------------------+-----------------------------------
|  1 | chien             | Canis canis           | Bestiole à quatre pattes qui aim
|  2 | chat              | Felis silvestris      | Bestiole à quatre pattes qui sau
|  3 | tortue d'Hermann  | Testudo hermanni      | Bestiole avec une carapace très
|  4 | perroquet amazone | Alipiopsitta xanthops | Joli oiseau parleur vert et jaun
+----+-------------------+-----------------------+-----------------------------------
```

Et on les joint pour arriver à ceci :

Code : Console

```
+----+---------+---------------------+----------+-------------------------+--------
| id | sexe    | date_naissance      | nom      | commentaires            | espece_
+----+---------+---------------------+----------+-------------------------+--------
|  5 | NULL    | 2010-10-03 16:44:00 | Choupi   | Né sans oreille gauche |
|  8 | male    | 2008-09-11 15:38:00 | Bagherra | NULL                    |
|  9 | NULL    | 2010-08-23 05:18:00 | NULL     | NULL                    |
```

```
| 14 | femelle | 2009-05-26 08:56:00 | Fila      | NULL                  |
| 16 | femelle | 2009-05-26 08:50:00 | Louya     | NULL                  |
| 18 | femelle | 2007-04-24 12:59:00 | Zira      | NULL                  |
| 24 | male    | 2007-04-12 05:23:00 | Cartouche | NULL                  |
| 25 | male    | 2006-05-14 15:50:00 | Zambo     | NULL                  |
| 33 | male    | 2006-05-19 16:17:00 | Caribou   | NULL                  |
| 52 | femelle | 2006-03-15 14:26:00 | Redbul    | Insomniaque           |
| 58 | male    | 2008-02-20 02:50:00 | Gingko    | NULL                  |
+----+---------+---------------------+-----------+-----------------------+--------
```

Notion d'alias

Je fais ici une petite parenthèse avant de vous expliquer en détail le fonctionnement des jointures pour vous parler d'un petit truc bien utile : les alias.

Les alias sont des noms de remplacement, que l'on donne de manière temporaire (le temps d'une requête en fait) à une colonne, une table, une donnée. Les alias sont introduits par le mot-clé **AS**. Ce mot-clé est facultatif, vous pouvez très bien définir un alias sans utiliser **AS**, mais je préfère personnellement toujours le mettre. Je trouve qu'on y voit plus clair.

Comment ça marche ?

Prenons cette requête toute simple :

Code : SQL

```sql
SELECT 5+3;
```

Code : Console

```
+-----+
| 5+3 |
+-----+
|   8 |
+-----+
```

Et imaginons que ce calcul savant représente en fait le nombre de chiots de Cartouche, qui a eu une première portée de 5 chiots, et une seconde de seulement 3 chiots. Nous voudrions donc indiquer qu'il s'agit bien de ça, et non pas d'un calcul inutile destiné simplement à illustrer une notion obscure.

Facile ! Il suffit d'utiliser les alias :

Code : SQL

```sql
SELECT 5+3 AS Chiots_Cartouche;

-- OU, sans utiliser AS

SELECT 5+3 Chiots_Cartouche;
```

Code : Console

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

```
| Chiots_Cartouche |
+------------------+
|                8 |
+------------------+
```

Bon, tout ça c'est bien joli, mais pour l'instant ça n'a pas l'air très utile...
Prenons donc un exemple plus parlant. Retounez donc voir la table créée grâce aux jointures un peu plus haut. Celle qui joint Espece et Animal. Cette table temporaire possède 15 colonnes :

- ***id***
- *sexe*
- *date_naissance*
- *nom*
- *commentaires*
- *espece_id*
- *race_id*
- *mere_id*
- *pere_id*
- ***id***
- *nom_courant*
- *nom_latin*
- *description*

Mais que vois-je ? J'ai deux colonnes *id* ! Comment faire pour les différencier ? Comment être sûrs de savoir à quoi elles se rapportent ?
Avec les alias pardi ! Il suffit de donner l'alias *espece_id* à la colonne *id* de la table *Espece*, et *animal_id* à la colonne *id* de la table *Animal*.
Tout à coup, ça vous semble plus intéressant non ??

Je vais vous laisser sur ce sentiment. Il n'y a pas grand-chose de plus à dire sur les alias, vous en comprendrez toute l'utilité à travers les nombreux exemples dans la suite de ce cours. L'important pour l'instant est que vous sachiez que ça existe et comment les définir.

Jointure interne

L'air de rien, dans l'introduction, je vous ai déjà montré comment faire une jointure. Et la première émotion passée, vous devriez vous être dit "Tiens, mais ça n'a pas l'air bien compliqué en fait, les jointures".
Et en effet, une fois que vous aurez compris comment réfléchir aux jointures, tout se fera tout seul. Personnellement, ça m'aide vraiment d'imaginer la table virtuelle créée par la jointure, et de travailler sur cette table pour tout ce qui est conditions, tris, etc.

Revoici la jointure que je vous ai fait faire, et qui est en fait une jointure **interne**.

Code : SQL

```sql
SELECT Espece.description
FROM Espece
INNER JOIN Animal
    ON Espece.id = Animal.espece_id
WHERE Animal.nom = 'Cartouche';
```

Décomposons !

- **`SELECT Espece.description`** : je sélectionne la colonne *description* de la table *Espece*.
- **`FROM Espece`** : je travaille sur la table *Espece*.
- **`INNER JOIN Animal`** : je la joins (avec une jointure interne) à la table *Animal*.
- **`ON Espece.id = Animal.espece_id`** : la jointure se fait sur les colonnes *id* de la table *Espece* et *espece_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*.

A priori, si vous avez compris ça, vous avez tout compris~~, même si vous n'avez pas Free~~ !

Syntaxe

Alors, comme d'habitude, voici donc la syntaxe à utiliser pour faire des requêtes avec jointure(s) interne(s).

Code : SQL

```
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 ...]
```

Condition de jointure

La clause **ON** sert à preciser la condition de la jointure. C'est-à-dire sur quel(s) critère(s) les deux tables doivent être jointes. Dans la plupart des cas, il s'agira d'une condition d'égalité simple, comme **ON** `Animal.espece_id = Espece.id`. Il est cependant tout à fait possible d'avoir plusieurs conditions à remplir pour lier les deux tables. On utilise alors les opérateurs logiques habituels. Par exemple, une jointure peut très bien se faire sur plusieurs colonnes :

Code : SQL

```
SELECT *
FROM nom_table1
INNER JOIN nom_table2
   ON colonne1_table1 = colonne1_table2
      AND colonne2_table1 = colonne2_table2
      [AND ...];
```

Expliciter le nom des colonnes

Il peut arriver que vous ayez dans vos deux tables des colonnes portant le même nom. C'est le cas dans notre exemple, puisque la table Animal comporte une colonne id, tout comme la table Espece. Il est donc important de préciser de quelle colonne on parle dans ce cas-là.
Vous l'aurez vu dans notre requête, on utilise pour ça l'opérateur **.** (donc *nom_table.nom_colonne*). Pour les colonnes ayant un nom non-ambigu (qui n'existe dans aucune autre table de la jointure), il n'est pas obligatoire de préciser la table.
En général, je précise systématiquement la table quand il s'agit de grosses requêtes avec plusieurs jointures. Par contre, pour les petites jointures courantes, c'est vrai que c'est moins long à écrire si on laisse tomber la table. Dans la suite de ce tutoriel, j'essayerai cependant de toujours indiquer le nom de la table, afin qu'il n'y ait aucune confusion/incompréhension possible.

Exemple :

Code : SQL

```
SELECT Espece.id,                   -- ici, pas le choix, il faut
préciser
       Espece.description,           -- ici, on pourrait mettre juste
```

```
        description
               Animal.nom                    -- idem, la précision n'est pas
        obligatoire. C'est cependant plus clair puisque les espèces ont un
        nom aussi
        FROM Espece
        INNER JOIN Animal
             ON Espece.id = Animal.espece_id
        WHERE Animal.nom LIKE 'C%';
```

Résultat :

Code : Console

```
+----+-------------------------------------------------------------------------+--
| id | description                                                             | n
+----+-------------------------------------------------------------------------+--
|  1 | Bestiole à quatre pattes qui aime les caresses et tire souvent la langue | C
|  1 | Bestiole à quatre pattes qui aime les caresses et tire souvent la langue | C
|  1 | Bestiole à quatre pattes qui aime les caresses et tire souvent la langue | C
|  1 | Bestiole à quatre pattes qui aime les caresses et tire souvent la langue | C
|  2 | Bestiole à quatre pattes qui saute très haut et grimpe aux arbres        | C
|  2 | Bestiole à quatre pattes qui saute très haut et grimpe aux arbres        | C
|  2 | Bestiole à quatre pattes qui saute très haut et grimpe aux arbres        | C
|  2 | Bestiole à quatre pattes qui saute très haut et grimpe aux arbres        | C
|  2 | Bestiole à quatre pattes qui saute très haut et grimpe aux arbres        | C
|  2 | Bestiole à quatre pattes qui saute très haut et grimpe aux arbres        | C
|  3 | Bestiole avec une carapace très dure                                     | C
|  3 | Bestiole avec une carapace très dure                                     | C
+----+-------------------------------------------------------------------------+--
```

Utiliser les alias

Les alias sont souvent fort utilisés avec les jointures. Ils permettent notamment de renommer les tables, et ainsi d'écrire moins.

Exemple :

Code : SQL

```
SELECT e.id,
       e.description,
       a.nom
FROM Espece AS e           -- On donne l'alias "e" à Espece
INNER JOIN Animal AS a     -- et l'alias "a" à Animal.
     ON e.id = a.espece_id
WHERE a.nom LIKE 'C%';
```

Comme vous le voyez, on peut facilement utiliser les alias pour écrire moins. A nouveau, c'est quelque chose que j'utilise souvent pour de petites requêtes ponctuelles. Par contre, pour de grosses requêtes, je préfère les noms explicites, c'est plus facile de s'y retrouver.

Un autre utilité, c'est de renommer les colonnes, pour que le résultat soit plus clair. En effet, regardez le résultat de la requête ci-dessus. Vous avez trois colonnes : *id*, *description* et *nom*, le nom de la table, n'est indiqué nulle part. A priori, vous savez ce que vous avez demandé, surtout qu'il n'y a pas encore trop de colonnes, mais imaginez que vous sélectionniez une vingtaine de colonnes. Ce serait quand même mieux de savoir de quel *id* on parle, s'il s'agit du nom de la bestiole, du maître, du père, du fils ou du saint-esprit.
Il est donc intéressant là aussi d'utiliser les alias. Ce qui nous donnerait :

Code : SQL

```
SELECT Espece.id AS id_espece,
       Espece.description AS description_espece,
       Animal.nom AS nom_bestiole
FROM Espece
INNER JOIN Animal
     ON Espece.id = Animal.espece_id
WHERE Animal.nom LIKE 'C%';
```

Résultat :

Code : Console

```
+-----------+-------------------------------------------------------------------
| id_espece | description_espece
+-----------+-------------------------------------------------------------------
|         1 | Bestiole à quatre pattes qui aime les caresses et tire souvent la lan
|         1 | Bestiole à quatre pattes qui aime les caresses et tire souvent la lan
|         1 | Bestiole à quatre pattes qui aime les caresses et tire souvent la lan
|         1 | Bestiole à quatre pattes qui aime les caresses et tire souvent la lan
|         2 | Bestiole à quatre pattes qui saute très haut et grimpe aux arbres
|         2 | Bestiole à quatre pattes qui saute très haut et grimpe aux arbres
|         2 | Bestiole à quatre pattes qui saute très haut et grimpe aux arbres
|         2 | Bestiole à quatre pattes qui saute très haut et grimpe aux arbres
|         2 | Bestiole à quatre pattes qui saute très haut et grimpe aux arbres
|         2 | Bestiole à quatre pattes qui saute très haut et grimpe aux arbres
|         3 | Bestiole avec une carapace très dure
|         3 | Bestiole avec une carapace très dure
+-----------+-------------------------------------------------------------------
```

Moi, je trouve ça plus clair !

Pourquoi "interne" ?

Oui parce que quand même, ça fait un bon quart-d'heure que je vous fais faire des jointures internes, et vous ne savez toujours pas ce que veut dire ce "interne".

C'est très simple ! Lorque l'on fait une jointure interne, cela veut dire qu'on exige qu'il y ait des données de part et d'autre de la jointure. Donc, si l'on fait une jointure sur la colonne *a* de la table *A* et la colonne *b* de la table *B* :

Code : SQL

```
SELECT *
FROM A
INNER JOIN B
    ON A.a = B.b
```

Ceci retournera UNIQUEMENT les lignes pour lesquelles *A.a* et *B.b* ont une correspondance.

Par exemple, si je veux connaître la race de mes animaux, je fais la requête suivante :

Code : SQL

```
SELECT Animal.nom AS nom_animal, Race.nom AS race
```

```
FROM Animal
INNER JOIN Race
    ON Animal.race_id = Race.id
ORDER BY Race.nom, Animal.nom;
```

Ce qui me donne :

Code : Console

```
+----------------+---------------------+
| nom_animal     | race                |
+----------------+---------------------+
| Balou          | Berger allemand     |
| Bouli          | Berger allemand     |
| Pilou          | Berger allemand     |
| Rouquine       | Berger allemand     |
| Rox            | Berger allemand     |
| Samba          | Berger allemand     |
| Zambo          | Berger allemand     |
| Zira           | Berger allemand     |
| Cali           | Berger blanc suisse |
| Caroline       | Berger blanc suisse |
| Fila           | Berger blanc suisse |
| Java           | Berger blanc suisse |
| Callune        | Bleu russe          |
| Caribou        | Bleu russe          |
| Cawette        | Bleu russe          |
| Feta           | Bleu russe          |
| Filou          | Bleu russe          |
| Raccou         | Bleu russe          |
| Schtroumpfette | Bleu russe          |
| Moka           | Boxer               |
| Pataud         | Boxer               |
| Welva          | Boxer               |
| Zoulou         | Boxer               |
| Bagherra       | Maine coon          |
| Bilba          | Maine coon          |
| Capou          | Maine coon          |
| Cracotte       | Maine coon          |
| Farceur        | Maine coon          |
| Milla          | Maine coon          |
| Zara           | Maine coon          |
| Zonko          | Maine coon          |
| Boucan         | Singapura           |
| Boule          | Singapura           |
| Fiero          | Singapura           |
+----------------+---------------------+
```

Et on peut voir ici que les animaux pour lesquels je n'ai pas d'informations sur la race (donc race_id est **NULL**) ne sont pas repris dans les résultats. De même, aucun des chats n'est de la race "Sphynx", celle-ci n'est donc pas reprise. Si je veux les inclure, je dois utiliser une jointure externe.

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

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.

Alors, table de gauche, table de droite, laquelle est laquelle ? C'est très simple, nous lisons de gauche à droite, donc la table de gauche est la première table mentionnée dans la requête, c'est-à-dire, en général, la table donnée dans la clause **FROM**.

Donc, si l'on veut de nouveau connaitre la race de nos animaux, mais que cette fois-ci nous voulons également afficher les animaux qui n'ont pas de race, on peut utiliser la jointure suivante (je ne prends que les animaux dont le nom commence par "C", histoire de ne pas vous taper 60 lignes de résultats, mais essayez chez vous avec les conditions que vous voulez) :

Code : SQL

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

-- OU

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;
```

Résultat :

Code : Console

```
+------------+---------------------+
| nom_animal | race                |
+------------+---------------------+
| Canaille   | NULL                |
| Cartouche  | NULL                |
| Cheli      | NULL                |
| Chicaca    | NULL                |
| Choupi     | NULL                |
| Cali       | Berger blanc suisse |
| Caroline   | Berger blanc suisse |
| Callune    | Bleu russe          |
| Caribou    | Bleu russe          |
| Cawette    | Bleu russe          |
| Capou      | Maine coon          |
| Cracotte   | Maine coon          |
+------------+---------------------+
```

On ne connait pas la race de Canaille, Cartouche, Chelli, Chicaca et Choupi, et pourtant, ils font bien partie des lignes sélectionnées. Essayez donc avec une jointure interne, vous verrez qu'ils n'apparaîtront plus.

Jointures par la droite

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.

Par exemple, toujours avec les races (on prend uniquement les chats pour ne pas surcharger) :

Code : SQL

```
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;

- OU

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

Code : Console

```
+----------------+------------+
| nom_animal     | race       |
+----------------+------------+
| Callune        | Bleu russe |
| Caribou        | Bleu russe |
| Cawette        | Bleu russe |
| Feta           | Bleu russe |
| Filou          | Bleu russe |
| Raccou         | Bleu russe |
| Schtroumpfette | Bleu russe |
| Bagherra       | Maine coon |
| Bilba          | Maine coon |
| Capou          | Maine coon |
| Cracotte       | Maine coon |
| Farceur        | Maine coon |
| Milla          | Maine coon |
| Zara           | Maine coon |
| Zonko          | Maine coon |
| Boucan         | Singapura  |
| Boule          | Singapura  |
| Fiero          | Singapura  |
| NULL           | Sphynx     |
+----------------+------------+
```

On a bien une ligne avec la race "Sphynx", bien que nous n'ayons aucun sphynx dans notre table *Animal*.

À noter que toutes les jointures par la droite peuvent être faites grâce à une jointure par la gauche (et vice-versa). L'équivalent par la gauche de la requête que nous venons de faire serait par exemple :

Code : SQL

```
SELECT Animal.nom AS nom_animal, Race.nom AS race
FROM Race                                                  -- Table de
gauche
LEFT JOIN Animal                                            -- Table
de droite
    ON Animal.race_id = Race.id
```

```
WHERE Race.espece_id = 2
ORDER BY Race.nom, Animal.nom;
```

Syntaxes alternatives

Les syntaxes que je vous ai montrées jusqu'ici, avec `[INNER] JOIN` et `LEFT|RIGHT [OUTER] JOIN`, sont les syntaxes classiques, que vous retrouverez le plus souvent. Il existe cependant d'autres manières de faire des jointures.

Jointures avec USING

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`.

Code : SQL

```
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;
```

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

Cela peut paraître absurde, mais il est tout à fait possible de faire une jointure sans utiliser le mot `JOIN`. Ce n'est cependant possible que pour les jointures internes.
Il suffit de mentionner les tables que l'on veut joindre dans la clause `FROM` (séparées par des virgules), et de mettre la condition de jointure dans la clause `WHERE` (pas de clause `ON` donc).

Code : SQL

```
SELECT *
FROM Table1, Table2
WHERE Table1.colonne1 = Table2.colonne2;

-- équivalent à

SELECT *
FROM Table1
[INNER] JOIN Table2
    ON Table1.colonne1 = Table2.colonne2;
```

Je vous déconseille cependant d'utiliser cette syntaxe. En effet, lorsque vous ferez des grosses jointures, avec plusieurs conditions dans la clause `WHERE`, vous serez bien content de pouvoir différencier au premier coup d'oeil les conditions de jointures des conditions "normales".

Exemples d'application et exercices

Maintenant que vous savez comment faire une jointure, on va un peu s'amuser. Cette partie sera en fait un mini-TP. Je vous dis à quel résultat vous devez parvenir en utilisant des jointures, vous essayez, et ensuite, vous allez voir la réponse et les explications.

Techniquement, vous avez vu toute la théorie nécessaire pour réaliser toutes les requêtes que je vous demanderai ici.
Cependant, il y aura des choses que je ne vous ai pas "montrées" explicitement, comme les jointures avec plus que deux tables, ou les auto-jointures (joindre une table avec elle-même).
C'est voulu !

Je ne m'attends pas à ce que vous réussissiez à construire toutes les requêtes sans jeter un oeil à la solution. Le but ici est de vous faire réfléchir, et surtout de vous faire prendre conscience qu'on peut faire pas mal de chose en SQL, en combinant plusieurs techniques par exemple.
Si un jour vous vous dites "Tiens, ce serait bien si en SQL on pouvait...", arrêtez de vous le dire et faites-le, simplement ! Vous serez probablement plus souvent limité par votre imagination que par SQL (bon, entendons-nous bien, vous ne ferez jamais sortir un chat vivant de votre écran grâce à MySQL hein... ).

Bien, ça c'est dit, donc allons-y !


Pour jouer le jeu jusqu'au bout, il faut bien sûr considérer que vous ne connaissez pas les id correspondants aux différentes races, et espèces. Donc quand je demande la liste des chiens par exemple, il n'est pas intéressant de sélectionner les animaux `WHERE espece_id = 1`, et bien plus utile de faire une jointure avec la table *Espece*.

A/ Commençons par du facile

Des jointures sur deux tables, avec différentes conditions à respecter.

1. Moi, j'aime bien les chiens de berger

Vous devez obtenir la liste des races de chiens qui sont des chiens de berger.


On considère (même si ce n'est pas tout à fait vrai) que les chiens de berger ont "berger" dans leur nom de race.

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT Race.nom AS Race
FROM Race
INNER JOIN Espece ON Espece.id = Race.espece_id
WHERE Espece.nom_courant = 'chien' AND Race.nom LIKE '%berger%';
```

Code : Console

```
+----------------------+
| Race                 |
+----------------------+
| Berger allemand      |
| Berger blanc suisse  |
+----------------------+
```

Bon, c'était juste un échauffement. Normalement vous ne devriez pas avoir eu de difficultés avec cette requête. Peut-être avez-vous oublié la condition `Espece.nom_courant = 'chien'` ? On ne sait jamais, une race de chat (ou autre) pourrait très bien contenir "berger", or j'ai explicitement demandé les chiens.

2. Mais de quelle couleur peut bien être son pelage ?

Vous devez obtenir la liste des animaux (leur nom, date de naissance et race) pour lesquels nous n'avons aucune information sur la couleur que devrait avoir leur pelage.

Dans la description des races, j'utilise parfois "pelage", parfois "poils", et parfois "robe".

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT Animal.nom AS nom_animal, Animal.date_naissance, Race.nom
AS race
FROM Animal
LEFT JOIN Race
    ON Animal.race_id = Race.id
WHERE (Race.description NOT LIKE '%poil%'
       AND Race.description NOT LIKE '%robe%'
       AND Race.description NOT LIKE '%pelage%'
      )
      OR Race.id IS NULL;
```

Résultat :

Code : Console

```
+------------+---------------------+-----------+
| nom_animal | date_naissance      | race      |
+------------+---------------------+-----------+
| Roucky     | 2010-03-24 02:23:00 | NULL      |
| NULL       | 2009-08-03 05:12:00 | NULL      |
| Choupi     | 2010-10-03 16:44:00 | NULL      |
| Bobosse    | 2009-06-13 08:17:00 | NULL      |
| NULL       | 2010-08-23 05:18:00 | NULL      |
| Bobo       | 2010-07-21 15:41:00 | NULL      |
| Canaille   | 2008-02-20 15:45:00 | NULL      |
| Anya       | 2008-02-20 15:47:00 | NULL      |
| Louya      | 2009-05-26 08:50:00 | NULL      |
| Cartouche  | 2007-04-12 05:23:00 | NULL      |
| Fiero      | 2009-05-14 06:30:00 | Singapura |
| Boucan     | 2009-05-14 06:42:00 | Singapura |
| Boule      | 2009-05-14 06:45:00 | Singapura |
| Nikki      | 2007-04-01 18:17:00 | NULL      |
| Tortilla   | 2009-03-24 08:23:00 | NULL      |
| Scroupy    | 2009-03-26 01:24:00 | NULL      |
| Lulla      | 2006-03-15 14:56:00 | NULL      |
| Dana       | 2008-03-15 12:02:00 | NULL      |
| Cheli      | 2009-05-25 19:57:00 | NULL      |
| Chicaca    | 2007-04-01 03:54:00 | NULL      |
| Redbul     | 2006-03-15 14:26:00 | NULL      |
| Spoutnik   | 2007-04-02 01:45:00 | NULL      |
| Bubulle    | 2008-03-16 08:20:00 | NULL      |
| Relou      | 2008-03-15 18:45:00 | NULL      |
| Bulbizard  | 2009-05-25 18:54:00 | NULL      |
| Safran     | 2007-03-04 19:36:00 | NULL      |
| Gingko     | 2008-02-20 02:50:00 | NULL      |
| Bavard     | 2009-03-26 08:28:00 | NULL      |
| Parlotte   | 2009-03-26 07:55:00 | NULL      |
```

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

Alors, il faut soit qu'on ne connaisse pas la race (on n'a alors pas d'information sur le pelage de la race, a fortiori), soit qu'on connaisse la race et que dans la description de celle-ci, il n'y ait pas d'information sur le pelage.

1/ On ne connait pas la race

Les animaux dont on ne connait pas la race ont **NULL** dans la colonne *race_id*. Mais vu que l'on fait une jointure sur cette colonne, il ne faut pas oublier de faire une jointure **externe**, sinon tous ces animaux sans race seront éliminés.
Une fois cela fait, pour les sélectionner, il suffit de mettre la condition `Animaux.race_id IS NULL` par exemple, ou simplement `Race.n'importe quelle colonne IS NULL` vu qu'il n'y a pas de correspondance avec Race.

2/ Pas d'informations sur le pelage dans la description de la race

Je vous ai donné comme indice le fait que j'utilisais les mots "pelage", "poil" ou "robe" dans les descriptions des espèces. Il fallait donc sélectionner les races pour lesquelles la description ne contient pas ces mots. D'où l'utilisation de **NOT LIKE**.

Si on met tout ça ensemble : il fallait faire une jointure externe des tables *Animal* et *Race*, et ensuite sélectionner les animaux qui n'avaient SOIT pas de race, SOIT une race dont la description ne contient ni "pelage", ni "poil", ni "robe".

B/ Compliquons un peu les choses

Jointures sur deux tables, ou plus !

1. La race ET l'espèce

Vous devez obtenir la liste des animaux de la base (tous), avec leur sexe, leur espèce (nom latin) et leur race s'ils en ont une. Regroupez les mêmes espèces ensemble, et au sein de l'espèce, les races.

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT Animal.nom as nom_animal, Animal.sexe, Especе.nom_latin as
espece, Race.nom as race
FROM Animal
INNER JOIN Especе
    ON Animal.espece_id = Especе.id
LEFT JOIN Race
    ON Animal.race_id = Race.id
ORDER BY Especе.nom_latin, Race.nom;
```

Code : Console

```
+----------------+---------+----------------------+---------------------+
| nom_animal     | sexe    | espece               | race                |
+----------------+---------+----------------------+---------------------+
| Gingko         | male    | Alipiopsitta xanthops | NULL                |
| Bavard         | male    | Alipiopsitta xanthops | NULL                |
| Parlotte       | femelle | Alipiopsitta xanthops | NULL                |
| Safran         | male    | Alipiopsitta xanthops | NULL                |
| Canaille       | femelle | Canis canis          | NULL                |
| Cartouche      | male    | Canis canis          | NULL                |
| Anya           | femelle | Canis canis          | NULL                |
| Louya          | femelle | Canis canis          | NULL                |
```

```
| Bobo            | male    | Canis canis           | NULL                |
| Rox             | male    | Canis canis           | Berger allemand     |
| Zira            | femelle | Canis canis           | Berger allemand     |
| Pilou           | male    | Canis canis           | Berger allemand     |
| Zambo           | male    | Canis canis           | Berger allemand     |
| Bouli           | male    | Canis canis           | Berger allemand     |
| Rouquine        | femelle | Canis canis           | Berger allemand     |
| Samba           | male    | Canis canis           | Berger allemand     |
| Balou           | male    | Canis canis           | Berger allemand     |
| Fila            | femelle | Canis canis           | Berger blanc suisse |
| Cali            | femelle | Canis canis           | Berger blanc suisse |
| Caroline        | femelle | Canis canis           | Berger blanc suisse |
| Java            | femelle | Canis canis           | Berger blanc suisse |
| Welva           | femelle | Canis canis           | Boxer               |
| Moka            | male    | Canis canis           | Boxer               |
| Pataud          | male    | Canis canis           | Boxer               |
| Zoulou          | male    | Canis canis           | Boxer               |
| Choupi          | NULL    | Felis silvestris      | NULL                |
| Roucky          | NULL    | Felis silvestris      | NULL                |
| Schtroumpfette  | femelle | Felis silvestris      | Bleu russe          |
| Caribou         | male    | Felis silvestris      | Bleu russe          |
| Callune         | femelle | Felis silvestris      | Bleu russe          |
| Filou           | male    | Felis silvestris      | Bleu russe          |
| Cawette         | femelle | Felis silvestris      | Bleu russe          |
| Feta            | femelle | Felis silvestris      | Bleu russe          |
| Raccou          | male    | Felis silvestris      | Bleu russe          |
| Bilba           | femelle | Felis silvestris      | Maine coon          |
| Zara            | femelle | Felis silvestris      | Maine coon          |
| Zonko           | male    | Felis silvestris      | Maine coon          |
| Cracotte        | femelle | Felis silvestris      | Maine coon          |
| Milla           | femelle | Felis silvestris      | Maine coon          |
| Capou           | male    | Felis silvestris      | Maine coon          |
| Bagherra        | male    | Felis silvestris      | Maine coon          |
| Farceur         | male    | Felis silvestris      | Maine coon          |
| Fiero           | male    | Felis silvestris      | Singapura           |
| Boucan          | male    | Felis silvestris      | Singapura           |
| Boule           | femelle | Felis silvestris      | Singapura           |
| NULL            | femelle | Testudo hermanni      | NULL                |
| Relou           | male    | Testudo hermanni      | NULL                |
| Redbul          | femelle | Testudo hermanni      | NULL                |
| Dana            | femelle | Testudo hermanni      | NULL                |
| Tortilla        | femelle | Testudo hermanni      | NULL                |
| Bobosse         | femelle | Testudo hermanni      | NULL                |
| Bulbizard       | male    | Testudo hermanni      | NULL                |
| Spoutnik        | male    | Testudo hermanni      | NULL                |
| Cheli           | femelle | Testudo hermanni      | NULL                |
| Scroupy         | femelle | Testudo hermanni      | NULL                |
| NULL            | NULL    | Testudo hermanni      | NULL                |
| Bubulle         | male    | Testudo hermanni      | NULL                |
| Chicaca         | femelle | Testudo hermanni      | NULL                |
| Lulla           | femelle | Testudo hermanni      | NULL                |
| Nikki           | femelle | Testudo hermanni      | NULL                |
+-----------------+---------+-----------------------+---------------------+
```

Comme vous voyez, c'est non seulement très simple de faire des jointures sur plus d'une table, c'est également possible de mélanger jointures internes et externes. Si ça vous pose problème, essayez vraiment de vous imaginer les étapes.
D'abord, on fait la jointure de *Animal* et *Espece*. On se retrouve alors avec une grosse table qui possède toutes les colonnes de *Animal* et toutes les colonnes de *Espece*. Et à cette grosse table (à la fois virtuelle et intermédiaire), on joint la table *Race*, grâce à la colonne *Animal.race_id*.
Notez que l'ordre dans lequel vous faites les jointures n'est pas important.

En ce qui concerne la clause **`ORDER BY`**, j'ai choisi de trier par ordre alphabétique, mais il est évident que vous pouviez également trier sur les *id* de l'espèce et la race. L'important ici était de trier d'abord sur une colonne de *Espece*, ensuite sur une colonne de *Race*.

2. Futures génitrices

Vous devez obtenir la liste des chiens femelles dont on connait la race, et qui sont en âge de procréer (c'est-à-dire nées avant juillet 2010). Affichez leur nom, date de naissance et race.

Secret (cliquez pour afficher)

Code : SQL

```
SELECT Animal.nom AS nom_chienne, Animal.date_naissance, Race.nom
AS race
FROM Animal
INNER JOIN Espece
    ON Animal.espece_id = Espece.id
INNER JOIN Race
    ON Animal.race_id = Race.id
WHERE Espece.nom_courant = 'chien'
    AND Animal.date_naissance < '2010-07-01'
    AND Animal.sexe = 'femelle';
```

Code : Console

```
+-------------+---------------------+---------------------+
| nom_chienne | date_naissance      | race                |
+-------------+---------------------+---------------------+
| Rouquine    | 2007-04-24 12:54:00 | Berger allemand     |
| Zira        | 2007-04-24 12:59:00 | Berger allemand     |
| Caroline    | 2008-12-06 05:18:00 | Berger blanc suisse |
| Cali        | 2009-05-26 08:54:00 | Berger blanc suisse |
| Fila        | 2009-05-26 08:56:00 | Berger blanc suisse |
| Java        | 2009-05-26 09:02:00 | Berger blanc suisse |
| Welva       | 2008-03-10 13:45:00 | Boxer               |
+-------------+---------------------+---------------------+
```

Cette fois, il fallait faire une jointure interne avec *Race*, puisqu'on voulait que la race soit connue. Le reste de la requête ne présentait pas de difficulté majeure.

C/ Et maintenant, le test ultime !

Jointures sur deux tables ou plus, avec éventuelles auto-jointures.

Je vous ai fait rajouter, à la fin du chapitre précédent, deux jolies petites colonnes dans la table Animal : mere_id et pere_id. Le moment est venu de les utiliser !

1. Mon père, ma mère, mes frères et mes soeurs (Wohooooo)

Vous devez obtenir la liste des chats dont on connait les parents, ainsi que leur nom.

Secret (cliquez pour afficher)

Code : SQL

```
SELECT Animal.nom, Pere.nom AS Papa, Mere.nom AS Maman
```

```
FROM Animal
INNER JOIN Animal AS Pere
    ON Animal.pere_id = Pere.id
INNER JOIN Animal AS Mere
    ON Animal.mere_id = Mere.id
INNER JOIN Espece
    ON Animal.espece_id = Espece.id
WHERE Espece.nom_courant = 'chat';
```

Code : Console

```
+----------------+-------+-------+
| nom            | Papa  | Maman |
+----------------+-------+-------+
| Roucky         | Zonko | Milla |
| Schtroumpfette | Filou | Feta  |
+----------------+-------+-------+
```

Si celle-là, vous avez trouvé tout seul, je vous félicite ! Sinon, c'est un peu normal. Vous voici face au premier cas dans lequel les alias sont **obligatoires**. En effet, vous aviez sans doute compris que vous pouviez faire `FROM Animal INNER JOIN Animal`, puisque j'avais mentionné les auto-jointures, mais vous avez probablement bloqué sur la clause `ON`. Comment différencier les colonnes de *Animal* dans `FROM` des colonnes de *Animal* dans `JOIN` ? Vous savez maintenant qu'il suffit d'utiliser des alias.
Et il faut faire une jointure sur trois tables, puisqu'au final, vous avez besoin des noms de trois animaux. Or en liant deux tables *Animal* ensemble, vous avez deux colonnes *nom*. Pour pouvoir en avoir trois, il faut lier trois tables.

Prenez le temps de bien comprendre les auto-jointures, le pourquoi du comment, et le comment du pourquoi. Faites des schémas si besoin, imaginez les tables intermédiaires.

2. Je suis ton père

Histoire de se détendre un peu, vous devez maintenant obtenir la liste des enfants de Bouli (nom, sexe et date de naissance).

Secret (cliquez pour afficher)

Code : SQL

```
SELECT Animal.nom, Animal.sexe, Animal.date_naissance
FROM Animal
INNER JOIN Animal AS Pere
    ON Animal.pere_id = Pere.id
WHERE Pere.nom = 'Bouli';
```

Code : Console

```
+------+------+---------------------+
| nom  | sexe | date_naissance      |
+------+------+---------------------+
| Rox  | male | 2010-04-05 13:43:00 |
+------+------+---------------------+
```

Après la requête précédente, celle-ci devrait vous avoir semblé plutôt facile ! Notez qu'il y a plusieurs manières de faire bien

sûr. En voici une autre :

Code : SQL

```
SELECT Enfant.nom, Enfant.sexe, Enfant.date_naissance
FROM Animal
INNER JOIN Animal AS Enfant
    ON Enfant.pere_id = Animal.id
WHERE Animal.nom = 'Bouli';
```

L'important, c'est le résultat ! Evidemment, si vous avez utilisé 45 jointures et 74 conditions, alors non, ce n'est pas bon non plus. Mais du moment que vous n'avez joint que deux tables, ça devrait être bon.

3. C'est un pure race ?

Courage, c'est la dernière (et la plus trash )!

Vous devez obtenir la liste des animaux dont on connait le père, la mère, la race, la race du père, la race de la mère. Affichez le nom et la race de l'animal et de ses parents, ainsi que l'espèce de l'animal (pas des parents).

Secret (cliquez pour afficher)

Code : SQL

```
SELECT Espece.nom_courant AS espece, Animal.nom AS nom_animal,
Race.nom AS race_animal,
    Pere.nom AS papa, Race_pere.nom AS race_papa,
    Mere.nom AS maman, Race_mere.nom AS race_maman
FROM Animal
INNER JOIN Espece
    ON Animal.espece_id = Espece.id
INNER JOIN Race
    ON Animal.race_id = Race.id
INNER JOIN Animal AS Pere
    ON Animal.pere_id = Pere.id
INNER JOIN Race AS Race_pere
    ON Pere.race_id = Race_pere.id
INNER JOIN Animal AS Mere
    ON Animal.mere_id = Mere.id
INNER JOIN Race AS Race_mere
    ON Mere.race_id = Race_mere.id;
```

Code : Console

```
+--------+----------------+-----------------+-------+-----------------+-------+-
| espece | nom_animal     | race_animal     | papa  | race_papa       | maman | 
+--------+----------------+-----------------+-------+-----------------+-------+-
| chien  | Rox            | Berger allemand | Bouli | Berger allemand | Zira  | 
| chat   | Schtroumpfette | Bleu russe      | Filou | Bleu russe      | Feta  | 
+--------+----------------+-----------------+-------+-----------------+-------+-
```

Pfiou  ! Le principe est exactement le même que pour avoir simplement le nom des parents. Il faut simplement rajouter une jointure avec *Race* pour le père, pour la mère, et pour l'enfant, en n'oubliant pas de bien utiliser les alias bien sûr.

C'est avec ce genre de requête que l'on se rend compte à quel point il est important de bien structurer et indenter sa requête, et à quel point un choix d'alias intelligent peut clarifier les choses.

Si vous avez survécu jusqu'ici, vous devriez maintenant avoir compris en profondeur le principe des jointures, et être capable de manipuler de nombreuses tables sans faire tout tomber par terre.
C'est pas top les jointures ?  Vous verrez, vous ne pourrez bientôt plus vous en passer.

Pas de questionnaire à choix multiples pour ce chapitre, vous avez probablement assez sué sur la dernière partie.

Sous-requêtes

Nous allons maintenant apprendre à imbriquer plusieurs requêtes, ce qui vous permettra de faire en une seule fois ce qui vous aurait, jusqu'ici, demandé plusieurs étapes.

Une sous-requête est une requête **à l'intérieur** d'une autre requête. Avec le SQL, vous pouvez construire des requêtes imbriquées sur autant de niveaux que vous voulez. Vous pouvez également mélanger jointures et sous-requêtes. Tant que votre requête est correctement structurée, elle peut être aussi complexe que vous voulez.

Une sous-requête peut être faite dans une requête de type **SELECT**, **INSERT**, **UPDATE** ou **DELETE** (et quelques autres que nous n'avons pas encore vues). Nous ne verrons dans ce chapitre que les requêtes de sélection. Les jointures et sous-requêtes pour la modification, l'insertion et la suppression de données étant traitées dans le prochain chapitre.

La plupart des requêtes de sélection que vous allez voir dans ce chapitre sont tout à fait faisables autrement, souvent avec une jointure. Certains préfèrent les sous-requêtes aux jointures parce que c'est légèrement plus clair comme syntaxe, et peut-être plus intuitif. Il faut cependant savoir qu'une jointure sera toujours au moins aussi rapide que la même requête faite avec une sous-requête. Par conséquent, s'il est important pour vous d'optimiser les performances de votre application, utilisez plutôt des jointures lorsque c'est possible.

Sous-requêtes dans le FROM

Lorsque l'on fait une requête de type **SELECT**, le résultat de la requête nous est envoyé sous forme de table. Et grâce aux sous-requêtes, il est tout à fait possible d'utiliser cette table, et de refaire une recherche uniquement sur les lignes de celle-ci.

Petit exemple, je sélectionne tous les perroquets et toutes les tortues.

Code : SQL

```sql
SELECT Animal.*
FROM Animal
INNER JOIN Espece
    ON Espece.id = Animal.espece_id
WHERE Espece.nom_courant IN ('Tortue d''Hermann', 'Perroquet
amazone');
```

Code : Console

```
+----+---------+---------------------+-----------+------------------+-----------+--
| id | sexe    | date_naissance      | nom       | commentaires     | espece_id | r
+----+---------+---------------------+-----------+------------------+-----------+--
|  4 | femelle | 2009-08-03 05:12:00 | NULL      | NULL             |         3 |
|  6 | femelle | 2009-06-13 08:17:00 | Bobosse   | Carapace bizarre |         3 |
|  9 | NULL    | 2010-08-23 05:18:00 | NULL      | NULL             |         3 |
| 45 | femelle | 2007-04-01 18:17:00 | Nikki     | NULL             |         3 |
| 46 | femelle | 2009-03-24 08:23:00 | Tortilla  | NULL             |         3 |
| 47 | femelle | 2009-03-26 01:24:00 | Scroupy   | NULL             |         3 |
| 48 | femelle | 2006-03-15 14:56:00 | Lulla     | NULL             |         3 |
| 49 | femelle | 2008-03-15 12:02:00 | Dana      | NULL             |         3 |
| 50 | femelle | 2009-05-25 19:57:00 | Cheli     | NULL             |         3 |
| 51 | femelle | 2007-04-01 03:54:00 | Chicaca   | NULL             |         3 |
| 52 | femelle | 2006-03-15 14:26:00 | Redbul    | Insomniaque      |         3 |
| 53 | male    | 2007-04-02 01:45:00 | Spoutnik  | NULL             |         3 |
| 54 | male    | 2008-03-16 08:20:00 | Bubulle   | NULL             |         3 |
| 55 | male    | 2008-03-15 18:45:00 | Relou     | Surpoids         |         3 |
| 56 | male    | 2009-05-25 18:54:00 | Bulbizard | NULL             |         3 |
| 57 | male    | 2007-03-04 19:36:00 | Safran    | NULL             |         4 |
| 58 | male    | 2008-02-20 02:50:00 | Gingko    | NULL             |         4 |
| 59 | male    | 2009-03-26 08:28:00 | Bavard    | NULL             |         4 |
| 60 | femelle | 2009-03-26 07:55:00 | Parlotte  | NULL             |         4 |
+----+---------+---------------------+-----------+------------------+-----------+--
```

Et parmi ces perroquets et ces tortues, je veux savoir quel âge a l'animal le plus âgé. Je vais donc faire une sélection dans la table des résultats de ma requête.

Code : SQL

```
SELECT MIN(date_naissance)
FROM (
    SELECT Animal.*
    FROM Animal
    INNER JOIN Espece
        ON Espece.id = Animal.espece_id
    WHERE Espece.nom_courant IN ('Tortue d''Hermann', 'Perroquet amazone')
) AS tortues_perroquets;
```

Code : Console

```
+---------------------+
| MIN(date_naissance) |
+---------------------+
| 2006-03-15 14:26:00 |
+---------------------+
```

MIN() est une fonction qui va chercher la valeur minimale dans une colonne, nous reparlerons plus en détail des fonctions dans une prochaine partie de ce tutoriel.

Les règles à respecter

Parenthèses

Un sous-requête doit toujours se trouver dans des parenthèses, afin de définir clairement ses limites.

Alias

Dans le cas des sous-requêtes dans le **FROM**, il est également obligatoire de préciser un alias pour la table intermédiaire (le résultat de notre sous-requête). Si vous ne le faites pas, MySQL déclenchera une erreur. Ici, je l'ai appelée tortues_perroquets. Nommer votre table intermédiaire permet de plus de vous y référer si vous faites une jointure dessus, ou si certains noms de colonnes sont ambigus et que le nom de la table doit être précisé. Attention qu'il ne s'agit pas de la table *Animal*, mais bien d'une table tirée de *Animal*.
Par conséquent, si vous voulez préciser le nom de la table dans le **SELECT** principal, vous devez mettre **SELECT MIN**(tortues_perroquets.date_naissance), et non pas **SELECT MIN**(Animal.date_naissance).

Cohérence des colonnes

Les colonnes sélectionnées dans le **SELECT** "principal" doivent bien sûr être présentes dans la table intermédiaire. La requête suivante, par exemple, ne fonctionnera pas :

Code : SQL

```
SELECT MIN(date_naissance)
FROM (
    SELECT Animal.id, Animal.nom
    FROM Animal
    INNER JOIN Espece
```

```
            ON Espece.id = Animal.espece_id
        WHERE Espece.nom_courant IN ('Tortue d''Hermann', 'Perroquet
amazone')
) AS tortues_perroquets;
```

En effet, *tortues_perroquets* n'a que deux colonnes : *id* et *nom*. Il est donc impossible de sélectionner la colonne *date_naissance* de cette table.

Noms ambigus

Pour finir, attention aux nom de colonnes ambigus. Une table, même intermédiaire, ne peut pas avoir deux colonnes ayant le même nom. C'est la raison pour laquelle j'ai sélectionné `Animal.*` dans la sous-requête, et pas simplement `*`. En effet, si j'avais sélectionné toutes les colonnes des deux tables, il y aurait eu deux colonnes *id* (puisque je fais une jointure avec *Animal* et Espece). Si deux colonnes ont le même nom, il est nécessaire de renommer explicitement au moins l'une des deux.

Donc, si je veux toutes les colonnes de *Animal* et de *Espece*, je peux par exemple procéder comme ceci :

Code : SQL

```
SELECT MIN(date_naissance)
FROM (
    SELECT Animal.*, Espece.id AS espece_espece_id,
Espece.nom_courant, Espece.nom_latin, Espece.description    -- J'ai
renommé la colonne id de Espece, donc il n'y a plus de doublons.
    FROM Animal
-- Attention de ne pas la renommer espece_id, puisqu'il existe
également une colonne espece_id dans Animal !
    INNER JOIN Espece
        ON Espece.id = Animal.espece_id
    WHERE Espece.nom_courant IN ('Tortue d''Hermann', 'Perroquet
amazone')
) AS tortues_perroquets;
```

Sous-requêtes dans les conditions

Je vous ai donc dit que lorsque vous faites une requête **SELECT**, le résultat est sous forme de table. Ces tables de résultats peuvent avoir :

- plusieurs colonnes et plusieurs lignes ;
- ou plusieurs colonnes mais une seule ligne ;
- ou plusieurs lignes mais une seule colonne ;
- ou encore une seule ligne et une seule colonne (c'est-à-dire juste **une** valeur).

Les sous-requêtes renvoyant plusieurs lignes **et** plusieurs colonnes ne sont utilisées que dans les clauses **FROM**. Nous allons ici nous intéresser aux trois autres possibilités uniquement.

Comparaisons

Pour rappel, voici un tableau des opérateurs de comparaison.

Opérateur	Signification
=	égal
<	inférieur
<=	inférieur ou égal
>	supérieur

>=	supérieur ou égal
<> ou !=	différent
<=>	égal (valable pour **NULL** aussi)

On peut utiliser des comparaisons de ce type avec des sous-requêtes qui donnent comme résultat, soit une valeur (c'est-à-dire une seule ligne et une seule colonne), soit une ligne (donc plusieurs colonnes mais une seule ligne).

Sous-requête renvoyant une valeur

Le cas le plus simple est évidemment d'utiliser une sous-requête qui renvoie une valeur.

Code : SQL

```sql
SELECT *
FROM Animal
WHERE race_id = (
    SELECT id
    FROM Race
    WHERE nom = 'Berger Allemand');
```

Code : Console

```
+----+---------+---------------------+----------+-------------------+-----------+--
| id | sexe    | date_naissance      | nom      | commentaires      | espece_id | r
+----+---------+---------------------+----------+-------------------+-----------+--
|  1 | male    | 2010-04-05 13:43:00 | Rox      | Mordille beaucoup |         1 |
| 13 | femelle | 2007-04-24 12:54:00 | Rouquine | NULL              |         1 |
| 18 | femelle | 2007-04-24 12:59:00 | Zira     | NULL              |         1 |
| 20 | male    | 2007-04-24 12:45:00 | Balou    | NULL              |         1 |
| 22 | male    | 2007-04-24 12:42:00 | Bouli    | NULL              |         1 |
| 25 | male    | 2006-05-14 15:50:00 | Zambo    | NULL              |         1 |
| 26 | male    | 2006-05-14 15:48:00 | Samba    | NULL              |         1 |
| 28 | male    | 2006-05-14 15:40:00 | Pilou    | NULL              |         1 |
+----+---------+---------------------+----------+-------------------+-----------+--
```

Ici, la sous-requête renvoyait simplement *1*.

Code : Console

```
+----+
| id |
+----+
|  1 |
+----+
```

Comme vous voyez, cette requête est également faisable avec une simple jointure.
Voici un exemple de requête avec sous-requête qu'il est impossible de faire avec une simple jointure :

Code : SQL

```sql
SELECT id, nom, espece_id
FROM Race
```

```
    WHERE espece_id = (
        SELECT MIN(id)      -- Je rappelle que MIN() permet de récupérer
    la plus petite valeur de la colonne parmi les lignes sélectionnées
        FROM Espece);
```

Code : Console

```
+----+--------------------+-----------+
| id | nom                | espece_id |
+----+--------------------+-----------+
|  1 | Berger allemand    |         1 |
|  2 | Berger blanc suisse |         1 |
|  3 | Boxer              |         1 |
+----+--------------------+-----------+
```

En ce qui concerne les autres opérateurs, le principe est bien sûr exactement le même :

Code : SQL

```
SELECT id, nom, espece_id
FROM Race
WHERE espece_id < (
    SELECT id
    FROM Espece
    WHERE nom_courant = 'Tortue d''Hermann');
```

Code : Console

```
+----+--------------------+-----------+
| id | nom                | espece_id |
+----+--------------------+-----------+
|  1 | Berger allemand    |         1 |
|  2 | Berger blanc suisse |         1 |
|  3 | Boxer              |         1 |
|  4 | Bleu russe         |         2 |
|  5 | Maine coon         |         2 |
|  6 | Singapura          |         2 |
|  7 | Sphynx             |         2 |
+----+--------------------+-----------+
```

Ici la sous-requête renvoie *3*, donc nous avons bien les races dont l'espèce a une *id* inférieure à 3 (donc 1 et 2).

Sous-requête renvoyant une ligne

Seuls les opérateurs = et != (ou <>) sont utilisables avec une sous-requête de ligne, toutes les comparaisons de type "plus grand" ou "plus petit" ne sont pas supportées.

Dans le cas d'une sous-requête dont le résultat est une ligne, la syntaxe est la suivante :

Code : SQL

```
SELECT *
```

```
FROM nom_table1
WHERE [ROW](colonne1, colonne2) = (     -- le ROW n'est pas
obligatoire
    SELECT colonneX, colonneY
    FROM nom_table2
    WHERE...);                          -- Condition qui ne retourne
qu'UNE SEULE LIGNE
```

Cette requête va donc renvoyer toutes les lignes de la table1 dont la colonne1 = la colonneX de la ligne résultat de la sous-requête ET la colonne2 = la colonneY de la ligne résultat de la sous-requête.

Vous voulez un exemple peut-être ? Allons-y !

Code : SQL

```
SELECT *
FROM Animal
WHERE (id, race_id) = (
    SELECT id, espece_id
    FROM Race
    WHERE id = 7);
```

Code : Console

```
+----+---------+---------------------+----------+--------------+-----------+-------
| id | sexe    | date_naissance      | nom      | commentaires | espece_id | race_i
+----+---------+---------------------+----------+--------------+-----------+-------
|  7 | femelle | 2008-12-06 05:18:00 | Caroline | NULL         |         1 |
+----+---------+---------------------+----------+--------------+-----------+-------
```

Décomposons calmement. Voyons d'abord ce que la sous-requête donne comme résultat.

Code : SQL

```
SELECT id, espece_id
FROM Race
WHERE id = 7;
```

Code : Console

```
+----+-----------+
| id | espece_id |
+----+-----------+
|  7 |         2 |
+----+-----------+
```

Et comme condition, on a `WHERE (id, race_id) = #le résultat de la sous-requête#`.
Donc la requête renverra les lignes de la table Animal pour lesquelles *id* vaut 7 et *race_id* vaut 2.

Attention, il est impératif que la sous-requête ne renvoie qu'une seule ligne. Dans le cas contraire, la requête échouera.

Conditions avec IN et NOT IN

IN

Vous connaissez déjà l'opérateur **IN**. Nous l'avons d'ailleurs utilisé juste au-dessus.

Code : SQL

```sql
SELECT Animal.id, Animal.nom, Animal.espece_id
FROM Animal
INNER JOIN Espece
    ON Espece.id = Animal.espece_id
WHERE Espece.nom_courant IN ('Tortue d''Hermann', 'Perroquet
amazone');
```

Cet opérateur peut également s'utiliser avec une sous-requête dont le résultat est une **colonne** ou une **valeur**. On peut donc réécrire la requête ci-dessus en utilisant une sous-requête plutôt qu'une jointure :

Code : SQL

```sql
SELECT id, nom, espece_id
FROM Animal
WHERE espece_id IN (
    SELECT id
    FROM Espece
    WHERE nom_courant IN ('Tortue d''Hermann', 'Perroquet amazone')
);
```

Le fonctionnement est plutôt facile à comprendre. La sous-requête donne les résultats suivants :

Code : SQL

```sql
SELECT id            -- On ne sélectionne bien q'UNE SEULE COLONNE.
FROM Espece
WHERE nom_courant IN ('Tortue d''Hermann', 'Perroquet amazone');
```

Code : Console

```
+----+
| id |
+----+
|  3 |
|  4 |
+----+
```

Et donc la requête principale sélectionnera les lignes qui ont une *espece_id* parmi celles renvoyées par la sous-requête, donc 3 ou 4.

NOT IN

Si l'on utilise **NOT IN**, c'est bien sûr le contraire, on exclut les lignes qui correspondent au résultat de la sous-requête. La

requête suivante nous renverra donc les animaux dont l'*espece_id* n'est **pas** 3 ou 4.

Code : SQL

```
SELECT id, nom, espece_id
FROM Animal
WHERE espece_id NOT IN (
    SELECT id
    FROM Espece
    WHERE nom_courant IN ('Tortue d''Hermann', 'Perroquet amazone')
);
```

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.).

Bien entendu, comme pour **IN**, il faut des sous-requêtes dont le résultat est soit une **valeur**, soit une **colonne**.

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

ANY (ou SOME)

La requête suivante signifie donc "Sélectionne les lignes de la table *Animal*, dont l'*espece_id* est inférieure à **au moins une** des valeurs sélectionnées dans la sous-requête". C'est-à-dire inférieur à 3 **ou** à 4. Vous aurez donc dans les résultats toutes les lignes dont l'*espece_id* vaut 1, 2 ou 3 (puisque 3 est inférieur à 4).

Code : SQL

```
SELECT *
FROM Animal
WHERE espece_id < ANY (
    SELECT id
    FROM Espece
    WHERE nom_courant IN ('Tortue d''Hermann', 'Perroquet amazone')
);
```

ALL

Par contre, si vous utilisez **ALL** plutôt que **ANY**, elle signifiera "Sélectionne les lignes de la table *Animal*, dont l'*espece_id* est inférieure à **toutes** les valeurs sélectionnées dans la sous-requête". Donc inférieur à 3 **et** à 4. Vous n'aurez donc plus que les lignes dont l'*espece_id* vaut 1 ou 2.

Code : SQL

```
SELECT *
FROM Animal
WHERE espece id < ALL (
```

```
        SELECT id
        FROM Espece
        WHERE nom_courant IN ('Tortue d''Hermann', 'Perroquet amazone')
    );
```

Remarque : lien avec IN

Remarquez que = **ANY** est l'équivalent de **IN**, tandis que <> **ALL** est l'équivalent de **NOT IN**. Attention cependant que **ANY** et **ALL** (et **SOME**) ne peuvent s'utiliser qu'avec des sous-requêtes, et non avec des valeurs comme on peut le faire avec **IN**.
On ne peut donc pas faire ceci :

Code : SQL

```
SELECT id
FROM Espece
WHERE nom_courant = ANY ('Tortue d''Hermann', 'Perroquet amazone');
```

Code : Console

```
#1064 - You have an error in your SQL syntax;
```

Sous-requêtes 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.

Vu que ce n'est pas une définition extrêmement claire de prime abord, voici un exemple de requête avec une sous-requête corrélée :

Code : SQL

```
SELECT colonne1
FROM tableA
WHERE colonne2 = ANY (
    SELECT colonne3
    FROM tableB
    WHERE tableB.colonne4 = tableA.colonne5
    );
```

Si l'on prend la sous-requête toute seule, on ne pourra pas l'exécuter :

Code : SQL

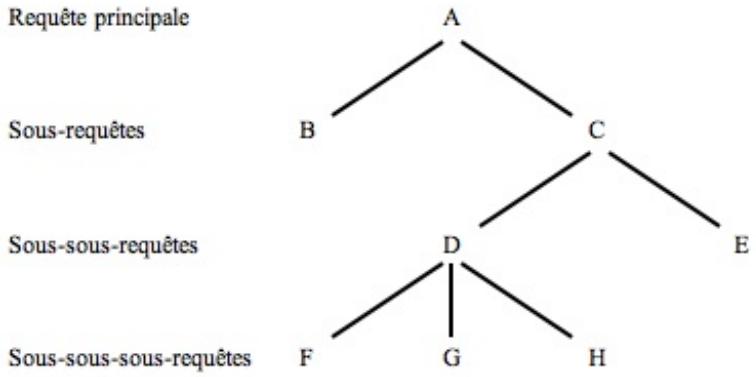
```
SELECT colonne3
FROM tableB
WHERE tableB.colonne4 = tableA.colonne5
```

En effet, seule la *tableB* est sélectionnée dans la clause **FROM**, il n'y a pas de jointure avec la *tableA*, et pourtant on utilise la *tableA* dans la condition.

Par contre, aucun problème pour l'utiliser comme sous-requête, puisque la clause **FROM** de la requête principale sélectionne la *tableA*. La sous-requête est donc **corrélée** à la requête principale.

Attention que si MySQL rencontre une table inconnue dans une sous-requête, elle va aller chercher au **niveau supérieur**

uniquement si cette table existe. Donc une sous-requête peut être corrélée à la requête, une sous-sous-requête peut être corrélée à la sous-requête dont elle dépend, ou à la requête principale, mais une sous-requête ne peut être corrélée à une autre sous-requête de la même requête principale. Si l'on prend le schéma suivant, on peut donc remonter l'arbre, mais jamais descendre d'un cran pour trouver les tables nécessaires.



Peuvent être corrélées à :

- **A** : B, C, D, E, F, G et H
- **B** : aucune
- **C** : D, E, F, G et H
- **D** : F, G et H
- **E** : aucune

Un petit exemple pratique maintenant :

Code : SQL

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

Si vous essayez d'exécuter uniquement la sous-requête, vous allez bien sûr juste déclencher une erreur.

Code : SQL

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

Code : Console

```
ERROR 1054 (42S22): Unknown column 'Espece.id' in 'where clause'
```

Par contre, la requête complète vous renvoie bien quelque chose :

Code : Console

```
+-------------+
| nom_courant |
+-------------+
| Chien       |
+-------------+
```

Mais pourquoi "Chien" uniquement ?
Prenons la sous-requête, et transformons un tout petit peu pour qu'elle soit correcte sans la requête principale (il suffit d'ajouter une jointure).

Code : SQL

```sql
SELECT Animal.id
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Especе.id
WHERE race_id IS NOT NULL
ORDER BY id;    -- Je rajoute un tri pour qu'on y voie plus clair.
```

Ce qui nous donne 35 résultats. Etant donné que cette sous-requête va servir à faire une condition sur l'*id* de l'espèce, on peut ne considérer que les résultats inférieurs à 5, puisque l'*id* la plus élevée de notre table *Espece* est 4. Il n'y a donc que deux résultats intéressants.

Code : Console

```
+----+
| id |
+----+
|  1 |
|  3 |
+----+
```

On pourrait donc penser que les espèces avec les *id* 1 et 3 (Chien et Perroquet Amazone) seront renvoyées par la requête complète, et pourtant seul "Chien" apparait dans les résultats.

Code : SQL

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

C'est en fait à cause de la corrélation. Sélectionnons l'*espece_id* en plus de l'*id* des résultats intéressants de la sous-requête. Au cas où, voici la requête à faire :

Code : SQL

```sql
SELECT Animal.id
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Espece.id
WHERE race_id IS NOT NULL
    AND Animal.id < 5
ORDER BY id;
```

Code : Console

```
+----+-----------+
| id | espece_id |
+----+-----------+
|  1 |         1 |
|  3 |         2 |
+----+-----------+
```

Or, la condition avec corrélation nous dit ceci : **WHERE** `Animal.espece_id = Espece.id`, ce qui n'arrive que pour la première ligne.
CQFD !

Conditions avec EXISTS et NOT EXISTS

Les conditions **EXISTS** et **NOT EXISTS** s'utilisent de la manière suivante :

Code : SQL

```sql
SELECT *
FROM table1
WHERE [NOT] EXISTS (sous-requête)
```

Une condition avec **EXISTS** sera vraie (et donc la requête renverra quelque chose) si la sous-requête correspondante renvoie au moins une ligne.
Une condition avec **NOT EXISTS** sera vraie si la sous-requête correspondante ne renvoie aucune ligne.

Petit exemple : on sélectionne les races s'il existe un animal qui s'appelle Balou.

Code : SQL

```sql
SELECT *
FROM Race
WHERE EXISTS (
                SELECT *
                FROM Animal
                WHERE nom = 'Balou'
              );
```

Résultats : vu qu'il existe bien un animal du nom de Balou dans notre table *Animal*, la condition est vraie, on sélectionne donc toutes les races. Si l'on avait utilisé un nom qui n'existe pas, la requête n'aurait renvoyé aucun résultat.

Code : Console

```
+----+---------------------+-----------+-----------------------------------------
| id | nom                 | espece_id | description
+----+---------------------+-----------+-----------------------------------------
|  1 | Berger allemand     |         1 | Chien sportif et élégant au pelage est den
|  2 | Berger blanc suisse |         1 | Petit chien au corps compact, avec des pat
|  3 | Boxer               |         1 | Chien de taille moyenne, au poil ras de cc
|  4 | Bleu russe          |         2 | Chat aux yeux verts et à la robe épaisse e
|  5 | Maine coon          |         2 | Chat de grande taille, à poils mi-longs.
|  6 | Singapura           |         2 | Chat de petite taille aux grands yeux en a
|  7 | Sphynx              |         2 | Chat sans poils.
|  8 | Nebelung            |         2 | Chat bleu russe, mais avec des poils longs
+----+---------------------+-----------+-----------------------------------------
```

Vous conviendrez cependant qu'une telle requête n'a pas beaucoup de sens, c'était juste pour vous faire comprendre le principe. En général, on utilise **WHERE [NOT] EXISTS** avec des sous-requêtes corrélées.

Par exemple, je veux séléctionner toutes les races dont on ne possède aucun animal.

Code : SQL

```
SELECT *
FROM Race
WHERE NOT EXISTS (
                    SELECT *
                    FROM Animal
                    WHERE Animal.race_id = Race.id
                  );
```

La sous-requête est bien corrélée à la requête principale, puisqu'elle utilise la table *Race*, qui n'est pas sélectionnée dans la sous-requête.

Et en résultat, on a bien le Sphynx, puisqu'on n'en possède aucun.

Code : Console

```
+----+--------+-----------+------------------+
| id | nom    | espece_id | description      |
+----+--------+-----------+------------------+
|  7 | Sphynx |         2 | Chat sans poils. |
+----+--------+-----------+------------------+
```

Beaucoup de nouveaux mots-clés dans cette partie ! Mais ne vous inquiètez pas, ça rentre vite une fois qu'on pratique un peu.

Jointures et sous-requêtes pour l'insertion, la modification et la suppression de données

Voici un petit chapitre bonus sur les jointures et les sous-requêtes.
Vous allez apprendre ici à utiliser ces outils non pas dans le cadre de la sélection de données, comme on l'a fait jusqu'à présent, mais pour l'insertion, la modification et la suppression de données.
A nouveau, cela devrait vous permettre de réaliser, en une seule requête, ce qui vous demande pour l'instant deux, trois, voire énormément de requêtes.

Insertion

Pour l'insertion nous n'allons nous servir que de sous-requêtes, pas de jointures. Quoique... La sous-requête pourrait très bien contenir une jointure !

Sous-requête pour l'insertion

Vous venez d'acquérir un magnifique Maine Coon chez un éléveur voisin. Vous vous apprêtez à l'insérer dans votre base de données, mais vous ne vous souvenez absolument pas de l'id de la race Maine Coon, ni de l'espèce Chat. Du coup, deux possibilités s'offrent à vous.

- Vous pouvez faire une requête pour sélectionner l'id de la race et de l'espèce, puis faire ensuite une requête d'insertion avec les données que vous venez d'afficher.
- Vous pouvez utiliser une sous-requête pour insérer directement l'id de la race et de l'espèce à partir du nom de la race.

Je ne sais pas ce que vous en pensez, mais moi je trouve la seconde option bien plus sexy !

Donc, allons-y. Qu'avons-nous comme information ?

Nom : Yoda
Date de naissance : 2010-11-09
Sexe : mâle
Espèce : chat
Race : Maine coon

Et de quoi avons nous besoin en plus pour l'insérer dans notre table ? L'id de l'espèce et de la race.
Comment récupérer ces deux id ? Ca, vous savez faire : une simple requête suffit :

Code : SQL

```sql
SELECT id AS race_id, espece_id
FROM Race
WHERE nom = 'Maine coon';
```

Code : Console

```
+---------+-----------+
| race_id | espece_id |
+---------+-----------+
|       5 |         2 |
+---------+-----------+
```

Bien, mais donc le but c'était de tout faire en une seule requête, pas d'insérer nous-mêmes 5 et 2 après les avoir récupérés.

C'est ici que les sous-requêtes interviennent. Nous allons utiliser une nouvelle syntaxe d'insertion.

INSERT INTO... SELECT

Cette syntaxe permet de sélectionner des éléments dans des tables, afin de les insérer directement dans une autre.

Code : SQL

```
INSERT INTO nom_table
   [(colonne1, colonne2, ...)]
SELECT [colonne1, colonne2, ...]
FROM nom_table2
[WHERE ...]
```

Vous n'êtes bien sûr pas obligé de préciser dans quelles colonnes se fait l'insertion, si vous sélectionnez une valeur pour toutes les colonnes de la table. Ce sera cependant rarement le cas puisque nous avons des clés primaires auto-incrémentées.

Avec cette requête, il est absolument indispensable (sinon l'insertion ne se fera pas), d'avoir le même nombre de colonnes dans l'insertion que dans la sélection, et qu'elles soient dans le même ordre.
Si vous n'avez pas le même nombre de colonnes, cela déclenchera une erreur. Par contre, si l'ordre n'est pas bon, vous aurez soit une erreur (si du coup vous essayez d'insérer un `VARCHAR` dans un `INT` par exemple), soit une insertion erronée (si par malheur vous avez échangé deux colonnes de types compatibles).

Bien, nous allons donc insérer le résultat de notre requête qui sélectionne les id de l'espèce et la race directement dans notre table animal.
Oui mais c'est bien beau tout ça, mais il faut également insérer le nom, le sexe, etc.

En effet ! Mais c'est très facile. Souvenez-vous, vous pouvez très bien faire des requêtes de ce type :

Code : SQL

```
SELECT 'Yoda' AS nom;
```

Et donc, en combinant avec notre requête précédente :

Code : SQL

```
SELECT id AS race_id, espece_id
FROM Race
WHERE nom = 'Maine coon';
```

Vous pouvez obtenir très facilement, et en une seule requête, tous les renseignements indispensables à l'insertion de notre petit Yoda !

Code : SQL

```
SELECT 'Yoda', 'male', '2010-11-09', id AS race_id, espece_id
FROM Race
WHERE nom = 'Maine coon';
```

Attention bien sûr de ne pas oublier les guillemets autour des chaînes de caractères, sinon MySQL va essayer de trouver la colonne Yoda de la table Race, et forcément, ça va moins bien fonctionner. Si tout se passe bien, cette requête devrait vous donner ceci :

Code : Console

```
+------+------+------------+---------+-----------+
| Yoda | male | 2010-11-09 | race id | espece id |
```

```
+------+------+------------+---------+-----------+
| Yoda | male | 2010-11-09 |       5 |         2 |
+------+------+------------+---------+-----------+
```

Les noms qui sont donnés aux colonnes n'ont pas d'importance, mais vous pouvez bien sûr changer ça avec des alias si cela vous perturbe.

Venons-en maintenant à notre super insertion !

Code : SQL

```sql
INSERT INTO Animal
    (nom, sexe, date_naissance, race_id, espece_id)
-- Je précise les colonne puisque je ne donne pas une valeur pour
toutes.
SELECT  'Yoda', 'male', '2010-11-09', id AS race_id, espece_id     -
- Attention à l'ordre !
FROM Race
WHERE nom = 'Maine coon';
```

Sélectionnons maintenant les Maine coon de notre base, pour vérifier que l'insertion s'est faite correctement.

Code : SQL

```sql
SELECT Animal.*, Race.nom AS race, Espece.nom_courant as espece
FROM Animal
INNER JOIN Race
    ON Animal.race_id = Race.id
INNER JOIN Espece
    ON Race.espece_id = Espece.id
WHERE Race.nom = 'Maine coon';
```

Et qui voyons-nous apparaître dans les résultats ? Notre petit Yoda !

Code : Console

```
+----+---------+---------------------+----------+---------------------------------
| id | sexe    | date_naissance      | nom      | commentaires
+----+---------+---------------------+----------+---------------------------------
|  8 | male    | 2008-09-11 15:38:00 | Bagherra | NULL
| 30 | male    | 2007-03-12 12:05:00 | Zonko    | NULL
| 32 | male    | 2007-03-12 12:07:00 | Farceur  | NULL
| 34 | male    | 2008-04-20 03:22:00 | Capou    | NULL
| 39 | femelle | 2008-04-20 03:26:00 | Zara     | NULL
| 40 | femelle | 2007-03-12 12:00:00 | Milla    | NULL
| 42 | femelle | 2008-04-20 03:20:00 | Bilba    | Sourde de l'oreille droite à 80%
| 43 | femelle | 2007-03-12 11:54:00 | Cracotte | NULL
| 61 | male    | 2010-11-09 00:00:00 | Yoda     | NULL
+----+---------+---------------------+----------+---------------------------------
```

Plutôt pratique, SQL être, quand bien le connaître, prendre la peine !

Modification

Utilisation des sous-requêtes

Pour la sélection

Imaginez que pour une raison bizarre, vous vouliez que tous les perroquets aient en commentaires "Coco veut un gâteau !".

Si vous saviez que l'id de l'espèce est 4, ce serait facile :

```sql
UPDATE Animal SET commentaires = 'Coco veut un gâteau' WHERE espece_id = 4;
```

Seulement voilà, vous ne savez évidemment pas que l'id de cette espèce est 4. Sinon, ce n'est pas drôle ! Vous allez donc utiliser une magnifique sous-requête pour modifier ces bruyants volatiles !

Code : SQL

```sql
UPDATE Animal
SET commentaires = 'Coco veut un gâteau !'
WHERE espece_id = (
                    SELECT id
                    FROM Espece
                    WHERE nom_courant LIKE 'Perroquet%'
                  );
```

Bien sûr, toutes les possibilités de conditions que l'on a vues pour la sélection sont encore valables pour les modifications. Après tout, une clause **WHERE** est une clause **WHERE** !

Pour l'élément à modifier

Ce matin, un client demande à voir vos chats Bleu Russe, car il compte en offrir un à sa fille. Ni une ni deux, vous vérifiez dans la base de données, puis allez chercher Schtroumpfette, Filou, Caribou, Raccou, Callune, Feta et Cawette.
Et là, horreur et damnation ! A l'instant où ses yeux se posent sur Cawette, le client devient vert de rage. Il prend à peine le temps de vous expliquer, outré, que Cawette n'est pas un Bleu Russe mais bien un Nebelung, à cause de ses poils longs, puis s'en va chercher un éleveur plus compétent.

Bon... L'erreur est humaine, mais autant la réparer rapidement. Vous insérez donc une nouvelle race dans la table ad-hoc.

Code : SQL

```sql
INSERT INTO Race (nom, espece_id, description)
VALUES ('Nebelung', 2, 'Chat bleu russe, mais avec des poils
longs...');
```

Une fois cela fait, il vous reste encore à modifier la race de Cawette. Ce pour quoi vous avez besoin de l'id de la race Nebelung que vous venez d'ajouter.
Et vous vous en doutez, il est tout à fait possible de le faire grâce à une sous-requête :

Code : SQL

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

Il est bien entendu indispensable que le résultat de la sous-requête soit une **valeur** !

Limitation des sous-requêtes dans un UPDATE

Une limitation importante des sous-requêtes est qu'on ne peut pas modifier un élément d'une table que l'on utilise dans une sous-requête.
Un petit exemple : vous trouver que Callune ressemble quand même fichtrement à Cawette, et ses poils sont aussi longs. Du coup, vous vous dites que vous auriez du également modifier la race de Callune. Vous essayez donc la requête suivante :

Code : SQL

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

Malheureusement :

Code : Console

```
ERROR 1093 (HY000): You can't specify target table 'Animal' for update in FROM clau
```

La sous-requête utilise la table *Animal*, or vous cherchez à modifier le contenu de celle-ci. C'est impossible !

Il vous faudra donc utiliser la même requête que pour Cawette, en changeant simplement le nom (je ne vous fais pas l'affront de vous l'écrire).

Modification avec jointure

Imaginons que vous vouliez que, pour les tortues et les perroquets, si un animal n'a pas de commentaire, on lui ajoute comme commentaire la description de l'espèce. Vous pourriez sélectionner les descriptions, les copier, retenir l'id de l'espèce, et ensuite faire un **UPDATE** pour les tortues et un autre pour les perroquets.
Ou alors, vous pourriez simplement faire un **UPDATE** avec jointure !

Voici la syntaxe que vous devriez utiliser pour le faire avec une jointure :

Code : SQL

```
UPDATE Animal
-- Classique !
INNER JOIN Espece
-- Jointure.
    ON Animal.espece_id = Espece.id
-- Condition de la jointure.
SET Animal.commentaires = Especе.description
-- Ensuite, la modification voulue.
WHERE Animal.commentaires IS NULL
-- Seulement s'il n'y a pas encore de commentaire.
AND Espece.nom_courant IN ('Perroquet amazone', 'Tortue
d''Hermann');      -- Et seulement pour les perroquets et les
tortues.
```

- Vous pouvez bien sûr mettre ce que vous voulez comme modifications. Ici, j'ai utilisé la valeur de la colonne dans l'autre table, mais vous auriez pu mettre `Animal.commentaires = 'Tralala'`, et la jointure n'aurait alors servi qu'à sélectionner les tortues et les perroquets grâce au nom courant de l'espèce.
- Toutes les jointures sont possibles. Vous n'êtes pas limités au jointures internes, ni à deux tables jointes.

Suppression

Cette partie sera relativement courte, puisque l'utilisation des sous-requêtes et des jointures pour la suppression ressemble assez à celle qu'on fait pour la modification.
Simplement, pour la suppression, les sous-requêtes et jointures ne peuvent servir qu'à sélectionner les lignes à supprimer.

Utilisation des sous-requêtes

On peut, tout simplement, utiliser une sous-requête dans la clause **WHERE**. Par exemple, imaginez que nous ayons deux animaux dont le nom est "Carabistouille", un chat et un perroquet. Vous désirez supprimer Carabistouille-le-chat, mais garder Carabistouille-le-perroquet. Vous ne pouvez donc pas utiliser la requête suivante, qui supprimera les deux :

Code : SQL

```
DELETE
FROM Animal
WHERE nom = 'Carabistouille';
```

Mais il suffit d'une sous-requête dans la clause **WHERE** pour sélectionner l'espèce, et le tour est joué !

Code : SQL

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

Limitations

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 jointure

Pour les jointures, c'est le même principe. Si je reprends le même problème que ci-dessus, voici comment supprimer la ligne voulue avec une jointure :

Code : SQL

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

Vous remarquez une petite différence avec la syntaxe "classique" de **DELETE** (sans jointure) : je précise le nom de la table dans laquelle les lignes doivent être supprimées juste après le **DELETE**. En effet, comme on utilise plusieurs tables, cette précision est obligatoire. Ici, on ne supprimera que les lignes de *Animal* correspondantes.
Ca valait la peine non ? Voilà qui conclut la partie "jointures et sous-requêtes". Je répète, il s'agit de notions extrêmement importantes. Si vous n'êtes pas sûrs d'avoir compris, il vaut mieux bien relire, et surtout pratiquer un maximum ! Du SQL sans jointure ni sous-requête, c'est un peu comme une bolognaise sans tomate...

Union de plusieurs requêtes

Toujours dans l'optique de rassembler plusieurs requêtes en une seule, voici l'**UNION**.
Faire l'union de deux requêtes, cela veut simplement dire réunir les résultats de la première requête ET les résultats de la seconde requête.
Voyons donc comment ça fonctionne !

Syntaxe

La syntaxe d'**UNION** est simplissime : vous avez donc deux requêtes **SELECT** dont vous voulez additionner les résultats, il vous suffit d'ajouter **UNION** entre ces deux requêtes.

Code : SQL

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

Le nombre de requêtes qu'il est possible d'unir est illimité. Si vous avez cinquante requêtes de sélection, placez un **UNION** entre les cinquante requêtes.

Code : SQL

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

Par exemple, vous pouvez obtenir les chiens et les tortues de la manière suivante :

Code : SQL

```
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';
```

Cette requête peut bien sûr être faite sans **UNION**, en faisant tout simplement un seul **SELECT** avec **WHERE** `Espece.nom_courant = 'Chat' OR Espece.nom_courant = 'Tortue d''Hermann'`. Il ne s'agit ici que d'un exemple destiné à illustrer la syntaxe. Nous verrons plus loins des exemples de cas où **UNION** est indispensable.

Les règles

Bien sûr, ce n'est pas si simple que ça, il faut respecter certaines contraintes.

Nombre des colonnes

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

Dans la requête que l'on a fait ci-dessus, aucun problème puisque l'on sélectionne *Animal.** dans les deux requêtes. Mais il ne serait donc pas possible de sélectionner un nombre de colonnes différent dans chaque requête intermédiaire.

Par exemple, la requête ci-dessous renverra une erreur :

Code : SQL

```
-- Pas le même nombre de colonnes --
-------------------------------------

SELECT Animal.id, Animal.nom, Espece.nom_courant
-- 3 colonnes sélectionnées
FROM Animal
INNER JOIN Espece
    ON Animal.espece_id = Espece.id
WHERE Espece.nom_courant = 'Chat'
UNION
SELECT Animal.id, Animal.nom, Espece.nom_courant, Animal.espece_id -
- 4 colonnes sélectionnées
FROM Animal
INNER JOIN Espece
    ON Animal.espece_id = Espece.id
WHERE Espece.nom_courant = 'Tortue d''Hermann';
```

Code : Console

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

Type et ordre des colonnes

En ce qui concerne le type des colonnes, je pense vous avoir déjà signalé que MySQL est très (très très) permissif. Par conséquent, si vous sélectionnez des colonnes de différents types, vous n'aurez pas d'erreurs, mais vous aurez des résultats un peu... spéciaux. 

Prenons la requête suivante :

Code : SQL

```
SELECT Animal.id, Animal.nom, Espece.nom_courant    -- 3e colonne :
nom_courant VARCHAR
FROM Animal
INNER JOIN Espece
    ON Animal.espece_id = Espece.id
WHERE Espece.nom_courant = 'Chat'
UNION
SELECT Animal.id, Animal.nom, Espece.id             -- 3e colonne :
id SMALLINT
FROM Animal
INNER JOIN Espece
    ON Animal.espece_id = Espece.id
WHERE Espece.nom_courant = 'Tortue d''Hermann';
```

Vous aurez bien un résultat :

Code : Console

```
+----+----------------+-------------+
| id | nom            | nom_courant |
+----+----------------+-------------+
|  2 | Roucky         | Chat        |
|  3 | Schtroumpfette | Chat        |
|  5 | Choupi         | Chat        |
|  8 | Bagherra       | Chat        |
| 29 | Fiero          | Chat        |
| 30 | Zonko          | Chat        |
| 31 | Filou          | Chat        |
| 32 | Farceur        | Chat        |
| 33 | Caribou        | Chat        |
| 34 | Capou          | Chat        |
| 35 | Raccou         | Chat        |
| 36 | Boucan         | Chat        |
| 37 | Callune        | Chat        |
| 38 | Boule          | Chat        |
| 39 | Zara           | Chat        |
| 40 | Milla          | Chat        |
| 41 | Feta           | Chat        |
| 42 | Bilba          | Chat        |
| 43 | Cracotte       | Chat        |
| 44 | Cawette        | Chat        |
| 61 | Yoda           | Chat        |
|  4 | NULL           | 3           |
|  6 | Bobosse        | 3           |
|  9 | NULL           | 3           |
| 45 | Nikki          | 3           |
| 46 | Tortilla       | 3           |
| 47 | Scroupy        | 3           |
| 48 | Lulla          | 3           |
| 49 | Dana           | 3           |
| 50 | Cheli          | 3           |
| 51 | Chicaca        | 3           |
| 52 | Redbul         | 3           |
| 53 | Spoutnik       | 3           |
| 54 | Bubulle        | 3           |
| 55 | Relou          | 3           |
| 56 | Bulbizard      | 3           |
+----+----------------+-------------+
```

Mais avouez que ce n'est pas un résultat très cohérent. MySQL va simplement convertir tout en chaîne de carctères pour ne pas avoir de problème. Soyez donc très prudent !

Vous pouvez constater en passant que les noms de colonnes utilisés dans les résultats sont ceux de la première requête effectuée. Vous pouvez bien sûr les renommer avec les alias.

Et enfin, pour l'ordre des colonnes, à nouveau vous n'aurez pas d'erreur tant que vous avez le même nombre de colonnes dans chaque requête, mais vous aurez des résultats bizarres. En effet, MySQL n'analyse pas les noms des colonnes pour trouver une quelconque correspondance d'une requête à l'autre, tout se fait sur base de la position de la colonne dans la requête.

Code : SQL

```
SELECT Animal.id, Animal.nom, Espece.nom_courant
FROM Animal
INNER JOIN Espece
    ON Animal.espece_id = Espece.id
WHERE Espece.nom_courant = 'Chat'
```

```
UNION
SELECT Animal.nom, Animal.id, Espece.nom_courant             -- 1e
et 2e colonne inversées par rapport à la première requête
FROM Animal
INNER JOIN Espece
    ON Animal.espece_id = Espece.id
WHERE Espece.nom_courant = 'Tortue d''Hermann';
```

Code : Console

```
+-----------+----------------+------------------+
| id        | nom            | nom_courant      |
+-----------+----------------+------------------+
| 2         | Roucky         | Chat             |
| 3         | Schtroumpfette | Chat             |
| 5         | Choupi         | Chat             |
| 8         | Bagherra       | Chat             |
| 29        | Fiero          | Chat             |
| 30        | Zonko          | Chat             |
| 31        | Filou          | Chat             |
| 32        | Farceur        | Chat             |
| 33        | Caribou        | Chat             |
| 34        | Capou          | Chat             |
| 35        | Raccou         | Chat             |
| 36        | Boucan         | Chat             |
| 37        | Callune        | Chat             |
| 38        | Boule          | Chat             |
| 39        | Zara           | Chat             |
| 40        | Milla          | Chat             |
| 41        | Feta           | Chat             |
| 42        | Bilba          | Chat             |
| 43        | Cracotte       | Chat             |
| 44        | Cawette        | Chat             |
| 61        | Yoda           | Chat             |
| NULL      | 4              | Tortue d'Hermann |
| Bobosse   | 6              | Tortue d'Hermann |
| NULL      | 9              | Tortue d'Hermann |
| Nikki     | 45             | Tortue d'Hermann |
| Tortilla  | 46             | Tortue d'Hermann |
| Scroupy   | 47             | Tortue d'Hermann |
| Lulla     | 48             | Tortue d'Hermann |
| Dana      | 49             | Tortue d'Hermann |
| Cheli     | 50             | Tortue d'Hermann |
| Chicaca   | 51             | Tortue d'Hermann |
| Redbul    | 52             | Tortue d'Hermann |
| Spoutnik  | 53             | Tortue d'Hermann |
| Bubulle   | 54             | Tortue d'Hermann |
| Relou     | 55             | Tortue d'Hermann |
| Bulbizard | 56             | Tortue d'Hermann |
+-----------+----------------+------------------+
```

UNION ALL

Exécutez la requête suivante :

Code : SQL

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

Non, non, je ne me suis pas trompée, je vous demande bien d'unir deux requêtes qui sont exactement les mêmes.

Résultat :

Code : Console

```
+----+-------------------+-----------------------+--------------------------------
| id | nom_courant       | nom_latin             | description
+----+-------------------+-----------------------+--------------------------------
|  1 | Chien             | Canis canis           | Bestiole à quatre pattes qui aim
|  2 | Chat              | Felis silvestris      | Bestiole à quatre pattes qui sau
|  3 | Tortue d'Hermann  | Testudo hermanni      | Bestiole avec une carapace très
|  4 | Perroquet amazone | Alipiopsitta xanthops | Joli oiseau parleur vert et jaun
+----+-------------------+-----------------------+--------------------------------
```

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 **UNION ALL**.

Code : SQL

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

Code : Console

```
+----+-------------------+-----------------------+--------------------------------
| id | nom_courant       | nom_latin             | description
+----+-------------------+-----------------------+--------------------------------
|  1 | Chien             | Canis canis           | Bestiole à quatre pattes qui aim
|  2 | Chat              | Felis silvestris      | Bestiole à quatre pattes qui sau
|  3 | Tortue d'Hermann  | Testudo hermanni      | Bestiole avec une carapace très
|  4 | Perroquet amazone | Alipiopsitta xanthops | Joli oiseau parleur vert et jaun
|  1 | Chien             | Canis canis           | Bestiole à quatre pattes qui aim
|  2 | Chat              | Felis silvestris      | Bestiole à quatre pattes qui sau
|  3 | Tortue d'Hermann  | Testudo hermanni      | Bestiole avec une carapace très
|  4 | Perroquet amazone | Alipiopsitta xanthops | Joli oiseau parleur vert et jaun
+----+-------------------+-----------------------+--------------------------------
```

Il n'est pas possible de mélanger **UNION DISTINCT** et **UNION ALL** dans une même requête. Si vous utilisez les deux, les **UNION ALL** seront considérés comme des **UNION DISTINCT**.

LIMIT et ORDER BY

LIMIT

Il est possible de restreindre les résultats avec **LIMIT** au niveau de la requête entière, ou au niveau des différentes requêtes unies. L'important est que ce soit clair. Par exemple, vous unissez deux requêtes et vous voulez limiter les résultats de la première. La requête suivante est parfaite pour ça :

Code : SQL

```sql
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.

Code : Console

```
+----+-----------------------+
| id | nom                   |
+----+-----------------------+
|  1 | Berger allemand       |
|  2 | Berger blanc suisse   |
|  3 | Boxer                 |
|  4 | Alipiopsitta xanthops |
|  1 | Canis canis           |
|  2 | Felis silvestris      |
|  3 | Testudo hermanni      |
+----+-----------------------+
```

En passant, voici un résultat qu'il n'est pas possible d'obtenir sans utiliser **UNION** !

Par contre, si vous voulez limiter les résultats de la seconde requête, comment faire ? Essayons la requête suivante.

Code : SQL

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

Résultat :

Code : Console

```
+----+--------------------+
| id | nom                |
+----+--------------------+
|  1 | Berger allemand    |
|  2 | Berger blanc suisse |
+----+--------------------+
```

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.

Code : SQL

```
SELECT id, nom
FROM Race
UNION
(SELECT id, nom latin
```

```
    FROM Espece
    LIMIT 2);
```

Et voilà le travail !

Code : Console

```
+----+-----------------------+
| id | nom                   |
+----+-----------------------+
|  1 | Berger allemand       |
|  2 | Berger blanc suisse   |
|  3 | Boxer                 |
|  4 | Bleu russe            |
|  5 | Maine coon            |
|  6 | Singapura             |
|  7 | Sphynx                |
|  8 | Nebelung              |
|  4 | Alipiopsitta xanthops |
|  1 | Canis canis           |
+----+-----------------------+
```

ORDER BY

Par contre, s'il est possible de trier le résultat final d'une requête avec **UNION**, on ne peut pas trier les résultats des requêtes intermédiaires. Par exemple, cette requête trie bien les résultats par ordre anti-alphabétique du nom :

Code : SQL

```
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** `Espece.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).

Code : Console

```
+----+-----------------------+
| id | nom                   |
+----+-----------------------+
|  3 | Testudo hermanni      |
|  7 | Sphynx                |
|  6 | Singapura             |
|  8 | Nebelung              |
|  5 | Maine coon            |
|  2 | Felis silvestris      |
|  1 | Canis canis           |
|  3 | Boxer                 |
|  4 | Bleu russe            |
|  2 | Berger blanc suisse   |
|  1 | Berger allemand       |
```

```
|  4 | Alipiopsitta xanthops |
+----+-----------------------+
```

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

Code : SQL

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

Code : Console

```
+----+-----------------------+
| id | nom                   |
+----+-----------------------+
|  4 | Alipiopsitta xanthops |
|  1 | Berger allemand       |
|  2 | Berger blanc suisse   |
|  4 | Bleu russe            |
|  3 | Boxer                 |
+----+-----------------------+
```

Exception pour les tris sur les requêtes intermédiaires

Je vous ai dit qu'il n'était pas possible de trier les résultats d'une requête intermédiaire. En réalité, c'est plus subtil que ça.
Dans une requête intermédiaire, il est possible d'utiliser un **ORDER BY** mais uniquement combiné à un **LIMIT**. Cela permettra de restreindre les résultats voulus (les *X* premiers dans l'ordre défini par l'**ORDER BY** par exemple).

Prenons la requête suivante :

Code : SQL

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

Cela vous renvoie bien 6 races et 3 espèces, mais imaginons que vous ne vouliez pas n'importe quelles races, mais les 6 dernières par ordre alphabétique du nom. Dans ce cas-là, vous pouvez utiliser **ORDER BY** en combinaison avec **LIMIT** dans la requête intermédiaire :

Code : SQL

```
(SELECT id, nom
```

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

Code : Console

```
+----+-----------------------+
| id | nom                   |
+----+-----------------------+
|  7 | Sphynx                |
|  6 | Singapura             |
|  8 | Nebelung              |
|  5 | Maine coon            |
|  3 | Boxer                 |
|  4 | Bleu russe            |
|  4 | Alipiopsitta xanthops |
|  1 | Canis canis           |
|  2 | Felis silvestris      |
+----+-----------------------+
```

Nous en avons maintenant fini avec les requêtes permettant d'utiliser plusieurs tables.

Options des clés étrangères

Lorsque je vous ai parlé des clés étrangères, et que je vous ai donné la syntaxe pour les créer, j'ai omis de vous parler des deux options, fort utiles :

- **ON DELETE**, qui permet de déterminer le comportement de MySQL en cas de suppression d'une référence ;
- **ON UPDATE**, qui permet de déterminer le comportement de MySQL en cas de modification d'une référence.

Nous allons donc maintenant examiner ces options.

Option sur suppression des clés étrangères

Petit rappel

La syntaxe

Voici comment on ajoute une clé étrangère à une table déjà existante :

Code : SQL

```
ALTER TABLE nom_table
ADD [CONSTRAINT fk_col_ref]          -- On donne un nom à la clé (facultatif)
    FOREIGN KEY colonne              -- La colonne sur laquelle on ajoute la clé
    REFERENCES table_ref(col_ref);   -- La table et la colonne de référence
```

Le principe expliqué ici est exactement le même si l'on crée la clé en même temps que la table. La commande **ALTER TABLE** est simplement plus courte, c'est la raison pour laquelle je l'utilise dans mes exemples plutôt que **CREATE TABLE**.

Le principe

Dans notre table *Animal*, nous avons par exemple mis une clé étrangère sur la colonne *race_id*, référençant la colonne *id* de la table *Race*. Cela implique que chaque fois qu'une valeur est insérée dans cette colonne (soit en ajoutant une ligne, soit en modifiant une ligne existante), MySQL va vérifier que cette valeur existe bien dans la colonne *id* de la table *Race*.
Aucun animal ne pourra donc avoir une *race_id* qui ne correspond à rien dans notre base.

Suppression d'une référence

Que se passe-t-il si l'on supprime la race des Boxers ? Certains animaux référencent cette espèce dans leur colonne race_id. On risque donc d'avoir des données incohérentes. Or, éviter cela est précisémment la raison d'être de notre clé étrangère.

Essayons :

Code : SQL

```
DELETE FROM Race
WHERE nom = 'Boxer';
```

Code : Console

```
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint
```

Ouf ! Visiblement, MySQL vérifie la contrainte de clé étrangère lors d'une suppression aussi, et empêche de supprimer une ligne si elle contient une référence utilisée ailleurs dans la base (ici, l'*id* de la ligne est donc utilisée par certaines lignes de la table *Animal*).

Mais ça veut donc dire que chaque fois qu'on veut supprimer des lignes de la table *Race*, il faut d'abord supprimer toutes les références à ces races. Dans notre base, ça va encore, il n'y a pas énormément de clés étrangères, mais imaginez si l'*id* de *Race* servait de référence à des clés étrangères dans ne fût-ce que cinq ou six tables. Pour supprimer une seule race, il faudrait faire jusqu'à six ou sept requêtes.

C'est donc ici qu'intervient notre option : **ON DELETE**, qui permet de changer la manière dont la clé étrangère gère la suppression d'une référence.

Syntaxe

Voici comment on ajoute cette option à la clé étrangère :

Code : SQL

```sql
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 !
```

Il y a donc quatre comportements possibles, que je m'en vais vous détailler tout de suite (bien que leurs noms soient plutôt clairs) : **RESTRICT**, **NO ACTION**, **SET NULL** et **CASCADE**

RESTRICT ou NO ACTION

RESTRICT 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** a exactement le même effet.

Cette équivalence de **RESTRICT** et **NO ACTION** est propre à MySQL. Dans d'autres SGBD, ces deux options n'auront pas le même effet (**RESTRICT** étant généralement plus strict que **NO ACTION**).

SET NULL

Si on choisit **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**;

D'ailleurs, ça me semble plutôt intéressant comme comportement dans cette situation ! Nous allons donc modifier notre clé étrangère *fk_race_id*. C'est-à-dire que nous allons supprimer la clé, puis la recréer avec le bon comportement :

Code : SQL

```sql
ALTER TABLE Animal
DROP FOREIGN KEY fk_race_id;

ALTER TABLE Animal
ADD CONSTRAINT fk_race_id
```

```
    FOREIGN KEY (race_id)
    REFERENCES Race(id)
    ON DELETE SET NULL;
```

Dorénavant, si vous supprimez une race, tous les animaux auxquels vous aurez attribué cette race auparavant auront **NULL** comme *race_id*.

Vérifions en supprimant les Boxers, depuis le temps qu'on essaye !

Code : SQL

```
-- Affichons d'abord tous les animaux, avec leur race --
--------------------------------------------------------
SELECT Animal.nom, Animal.race_id, Race.nom as race
FROM Animal
LEFT JOIN Race
    ON Animal.race_id = Race.id;

-- Supprimons ensuite la race 'Boxer' --
----------------------------------------
DELETE FROM Race
WHERE nom = 'Boxer';

-- Réaffichons les animaux --
-----------------------------
SELECT Animal.nom, Animal.race_id, Race.nom as race
FROM Animal
LEFT JOIN Race
    ON Animal.race_id = Race.id;
```

Les ex-boxers existent toujours dans la table *Animal*, mais ils n'appartiennent plus à aucune race.

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 !
Donc, si on choisit ce comportement pour la clé étrangère sur la colonne *especе_id* de la table *Animal*, vous supprimez l'espèce "Perroquet Amazone" et POUF !, quatre lignes de votre table *Animal* (les quatre perroquets) sont supprimées en même temps.
Il faut donc être bien sûr de ce que l'on fait si l'on choisit **ON DELETE CASCADE**. 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. Plutôt pratique non ?

Je répète : soyez bien sûr de ce que vous faites ! Je décline toute responsabilité en cas de perte de données causée par un **ON DELETE CASCADE** inconsidérément utilisé !

Option sur modification des clés étrangères

On peut également rencontrer des problèmes de cohérence des données en cas de modification. En effet, si l'on change par exemple l'*id* de la race "Singapura", tous les animaux qui ont l'ancienne *id* dans leur colonne *race_id* référenceront une ligne qui n'existe plus. Les modifications de références de clés étrangères sont donc soumises aux mêmes restrictions que la suppression. Voyez plutôt :

Code : SQL

```
UPDATE Race
SET id = 3              -- Anciennement, id des Boxer
WHERE nom = 'Singapura';
```

Code : Console

```
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint
```

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** et **NO** ACTION : empêche la modification si elle casse la contrainte (comportement par défaut).
- **SET NULL** : met **NULL** partout où la valeur modifiée était référencée.
- **CASCADE** : modifie également la valeur là où elle est référencée.

Petite explication à propos de CASCADE

CASCADE signifie que l'événement est répété sur les tables qui référencent la valeur. Pensez à des "réactions en cascade". Ainsi, une suppression provoquera d'autres suppressions, tandis qu'une modification provoquera d'autres ... Modifications ! 

Modifions par exemple à nouveau la clé étrangère sur *Animal.race_id*, avant de modifier l'*id* de la race "Singapura" (jetez d'abord un oeil aux données des tables *Race* et *Animal*, afin de voir les différences).

Code : SQL

```
-- Suppression de la clé --
----------------------------
ALTER TABLE Animal
DROP FOREIGN KEY fk_race_id;

-- Recréation de la clé avec les bonnes options --
--------------------------------------------------
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;

-- Modification de l'id des Singapura --
----------------------------------------
UPDATE Race
SET id = 3
WHERE nom = 'Singapura';
```

Les animaux notés comme étant des "Singapura" ont désormais leur *race_id* à 3. Parfait ! 

En règle générale (dans 99,99% des cas), c'est une très très mauvaise idée de changer la valeur d'une *id* (ou de votre clé primaire auto-incrémentée, quel que soit le nom que vous lui donnez). En effet, vous risquez des problèmes avec l'auto-incrément : si vous donnez une valeur pas encore atteinte par auto-incrémentation par exemple. Soyez bien conscient de ce que vous faites. Je ne l'ai montré ici que pour illustrer le **ON UPDATE**, parce que toutes nos clés étrangères référencent des clés primaires. Mais ce n'est pas le cas partout. Une clé étrangère pourrait parfaitement référencer un simple index, dépourvu de toute auto-incrémentation, auquel cas vous pouvez vous amuser à joyeusement en changer la valeur, autant de fois que vous le voudrez.

Utilisation de ces options dans notre base

Appliquons maintenant ce que nous avons appris à notre base de données *elevage*. Celle-ci comporte 5 clés étrangères :

Sur la table Animal

- *race_id* référence *Race.id*
- *espece_id* référence *Espece.id*
- *mere_id* référence *Animal.id*
- *pere_id* référence *Animal.id*

Sur la table Race

- *espece_id* référence *Espece.id*

Modifications

Pour les modifications, le mieux ici est de mettre `ON UPDATE CASCADE` pour toutes nos clés étrangères. Je ne vois pas de raisons d'empêcher les modifications (bien que je vous ai dit : ça ne devrait pas arriver dans le cas des clés primaires auto-incrémentées), ni de mettre à `NULL` quand la valeur de la référence change.

Suppressions

Le problème est plus délicat pour les suppressions. On a déjà défini `ON DELETE SET NULL` pour la clé sur Animal.race_id. Prenons les autres clés une à une.

Clé sur Animal.espece_id

Si l'on supprime une espèce de la base de données, c'est qu'on ne l'utilise plus dans notre élevage, donc a priori, on n'a plus besoin non plus des animaux de cette espèce. Sachant cela, on serait sans doute tenté de mettre `ON DELETE CASCADE`. Ainsi, en une seule requête, tout est fait.
Cependant, les animaux sont quand même le point central de notre base de données. Cela me parait donc un peu violent de les supprimer automatiquement de cette manière, en cas de suppression d'espèce. Par conséquent, je vous propose plutôt de laisser le `ON DELETE RESTRICT`. Supprimer une espèce n'est pas anodin, et de nombreux animaux d'un coup non plus. En empêchant la suppression des espèces tant qu'il existe des animaux de celle-ci, on oblige l'utilisateur à supprimer d'abord tous ces animaux d'abord. Pas de risque de fausse manoeuvre donc.
Attention que le `ON DELETE SET NULL` n'est bien sûr pas envisageable, puisque la colonne *espece_id* de la table *Animal* ne **peut pas** être `NULL`.

Il s'agit de mon point de vue personnel bien sûr. Si vous pensez que c'est mieux de mettre `ON DELETE CASCADE`, faites-le. On peut certainement trouver des arguments en faveur des deux possibilités.

.

Clé sur Animal.mere_id et Animal.pere_id

Ce n'est pas parce qu'on supprime un animal que tous ses enfant doivent être supprimés également. Par contre, mettre à `NULL` semble une bonne idée. `ON DELETE SET NULL` donc !

Clé sur Race.espece_id

Si une espèce est finalement supprimée, et donc que tous les animaux de cette espèce ont également été supprimés auparavant (puisqu'on a mis `ON DELETE RESTRICT` pour la clé sur *Animal.espece_id*, je rappelle), alors les races de cette espèce deviennent caduques. On peut donc utiliser un `ON DELETE CASCADE` ici.

Les requêtes

Vous avez toutes les informations nécessaires pour écrire ces requêtes, je vous encourage donc à les écrire vous-même avant de regarder mon code.

Secret (cliquez pour afficher)

Code : SQL

```sql
-- Animal.espece_id --
----------------------
ALTER TABLE Animal
DROP FOREIGN KEY fk_espece_id;

ALTER TABLE Animal
ADD CONSTRAINT fk_espece_id
    FOREIGN KEY (espece_id)
    REFERENCES Espece(id)
    ON DELETE RESTRICT   -- C'est le comportement par défaut, donc
c'est facultatif de le préciser
    ON UPDATE CASCADE;

-- Animal.mere_id --
--------------------
ALTER TABLE Animal
DROP FOREIGN KEY fk_mere_id;

ALTER TABLE Animal
ADD CONSTRAINT fk_mere_id
    FOREIGN KEY (mere_id)
    REFERENCES Individual(id)
    ON DELETE SET NULL
    ON UPDATE CASCADE;

-- Animal.pere_id --
--------------------
ALTER TABLE Animal
DROP FOREIGN KEY fk_pere_id;

ALTER TABLE Animal
ADD CONSTRAINT fk_pere_id
    FOREIGN KEY (pere_id)
    REFERENCES Individual(id)
    ON DELETE SET NULL
    ON UPDATE CASCADE;

-- Race.espece_id --
--------------------
ALTER TABLE Race
DROP FOREIGN KEY fk_race_espece_id;

ALTER TABLE Race
ADD CONSTRAINT fk_race_espece_id
    FOREIGN KEY (espece_id)
    REFERENCES Espece(id)
    ON DELETE CASCADE
    ON UPDATE CASCADE;
```

Avec ça, plus de risque de vous retrouver avec des données incohérentes en principe. Vous êtes parés !

Violation de contrainte d'unicité

Lorsque vous insérez ou modifiez une ligne dans une table, différents événements en relation avec les clés et les index peuvent se produire.

- L'insertion/la modification peut réussir, c'est évidemment le mieux
- L'insertion/la modification peut échouer car une contrainte de clé secondaire n'est pas respectée
- L'insertion/la modification peut échouer car une contrainte d'unicité (clé primaire ou index **UNIQUE**) n'est pas respectée

Nous allons ici nous intéresser à la troisième possibilité : ce qui arrive en cas de non-respect d'une contrainte d'unicité. Pour l'instant, dans ce cas là, une erreur est déclenchée. A la fin de ce chapitre, vous serez capable de modifier ce comportement :

- vous pourrez ne pas faire l'insertion/la modification mais ne pas déclencher d'erreur ;
- vous pourrez remplacer la (les) lignes qui existe(nt) déjà par la ligne que vous essayez d'insérer (uniquement insertion) ;
- enfin vous pourrez modifier la ligne qui existe déjà au lieu d'en insérer une nouvelle (uniquement insertion).

Ignorer les erreurs

Les commmandes d'insertion et modification possèdent une option : **IGNORE**, qui permet d'ignorer (tiens donc ! ) l'insertion/la modification si elle viole une contrainte d'unicité.

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*.

Code : SQL

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

Code : Console

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

Par contre, si l'on utilise le mot-clé **IGNORE** :

Code : SQL

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

Code : Console

```
Query OK, 0 rows affected (0.01 sec)
```

Plus d'erreur, la ligne n'a simplement pas été insérée.

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**.

Code : SQL

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

Code : Console

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

Code : SQL

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

Code : Console

```
Query OK, 0 rows affected (0.01 sec)
```

Et les chats sont toujours des *Felix silvestris* !

LOAD DATA INFILE

La même option est disponible avec la commande **LOAD DATA** INFILE, ce qui est plutôt pratique si vous voulez éviter de devoir chipoter à votre fichier suite à une insertion partielle due à une ligne qui ne respecte pas une contrainte d'unicité.

Syntaxe

Code : SQL

```sql
LOAD DATA [LOCAL] INFILE 'nom_fichier' IGNORE     -- IGNORE se place
juste avant INTO, comme dans INSERT
INTO TABLE nom_table
[FIELDS
    [TERMINATED BY '\t']
    [ENCLOSED BY '']
    [ESCAPED BY '\\' ]
]
[LINES
    [STARTING BY '']
    [TERMINATED BY '\n']
]
```

```
[IGNORE nombre LINES]
[(nom_colonne,...)];
```

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 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.

Mais que veut dire "remplacer l'ancienne ligne par la nouvelle" ?

Prenons par exemple Spoutnik la tortue, qui se trouve dans notre table *Animal*.

Code : SQL

```sql
SELECT *
FROM Animal
WHERE nom = 'Spoutnik';
```

Code : Console

```
+----+------+---------------------+----------+-------------------------------------
| id | sexe | date_naissance      | nom      | commentaires
+----+------+---------------------+----------+-------------------------------------
| 53 | male | 2007-04-02 01:45:00 | Spoutnik | Bestiole avec une carapace très dure
+----+------+---------------------+----------+-------------------------------------
```

Etant donné que nous avons mis un index **UNIQUE** sur (*nom, espece_id*), il est absolument impossible d'avoir une autre tortue du nom de Spoutnik dans notre table. La requête suivante va donc échouer lamentablement.

Code : SQL

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

Code : Console

```
ERROR 1062 (23000): Duplicate entry 'Spoutnik-3' for key 'uni_nom_espece'
```

Par contre, si on utilise **REPLACE** au lieu de **INSERT** :

Code : SQL

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

Code : Console

```
Query OK, 2 rows affected (0.06 sec)
```

Pas d'erreur, mais vous pouvez voir que **deux** lignes ont été affectées par la commande. En effet, Spoutnik est mort, vive Spoutnik !

Code : SQL

```
SELECT *
FROM Animal
WHERE nom = 'Spoutnik';
```

Code : Console

```
+----+---------+---------------------+----------+--------------+-----------+-------
| id | sexe    | date_naissance      | nom      | commentaires | espece_id | race_i
+----+---------+---------------------+----------+--------------+-----------+-------
| 63 | femelle | 2010-08-06 15:05:00 | Spoutnik | NULL         |         3 |    NUL
+----+---------+---------------------+----------+--------------+-----------+-------
```

Comme vous voyez, nous n'avons toujours qu'une seule tortue du nom de Spoutnik, mais il ne s'agit plus du mâle né le 2 avril 2007 que nous avions précédemment, mais bien de la femelle née le 6 août 2010 que nous venons d'insérer avec **REPLACE INTO**. L'autre Spoutnik a été purement et simplement remplacé.

Attention cependant, quand je dis que l'ancien Spoutnik a été remplacé, j'utilise le terme "remplacé" car il s'agit de la traduction de **REPLACE**. Il s'agit cependant d'un abus de langage. En réalité, la ligne de l'ancien Spoutnik a été supprimée, et ensuite seulement, le nouveau Spoutnik a été inséré. C'est d'ailleurs pour cela que les deux Spoutnik n'ont pas du tout la même *id*.

Remplacement de plusieurs lignes

Pourquoi ai-je bien précisé qu'il ne s'agissait pas vraiment d'un remplacement, mais d'une suppression suivie d'une insertion ? Parce que ce comportement a des conséquences qu'il ne faut pas négliger !

Prenons par exemple un table sur laquelle existent plusieurs contraintes d'unicité. C'est le cas de *Animal*, puisqu'on a donc cet index **UNIQUE** (*nom, espece_id*), ainsi que la clé primaire (l'*id* doit donc être unique aussi).

Nous allons insérer avec **REPLACE** une ligne qui viole les deux contraintes d'unicité :

Code : SQL

```
REPLACE INTO Animal (id, sexe, nom, date_naissance, espece_id) -- Je
donne moi-même une id, qui existe déjà !
VALUES (32, 'male', 'Spoutnik', '2009-07-26 11:52:00', 3);  -- Et
Spoutnik est mon souffre-douleur du jour.
```

Code : Console

```
Query OK, 3 rows affected (0.05 sec)
```

Cette fois-ci, trois lignes ont été affectées. Trois !
Tout simplement, les deux lignes qui empêchaient l'insertion à cause des contraintes d'unicité ont été supprimées. La ligne qui avait l'*id* 32, ainsi que l'ancien Spoutnik ont été supprimés. Le nouveau Spoutnik a ensuite été inséré.

Je l'ai fait ici pour vous donner un exemple, mais je rappelle que c'est une **très mauvaise idée** de donner soi-même une id lorsque la colonne est auto-incrémentée (ce qui sera presque toujours le cas).

LOAD DATA INFILE

REPLACE est également disponible avec **LOAD DATA** INFILE. Le comportement est exactement le même.

Bien entendu, **IGNORE** et **REPLACE** ne peuvent pas être utilisés en même temps. C'est l'un **ou** l'autre.

Syntaxe

Code : SQL

```
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 l'ancienne ligne

Donc, **REPLACE** supprime l'ancienne ligne (ou les anciennes lignes), puis insère la nouvelle. Mais parfois, ce qu'on veut, c'est bien modifier la ligne déjà existante. C'est possible grâce à la clause **ON** DUPLICATE **KEY UPDATE** de la commande **INSERT**.

Syntaxe

Voici donc la syntaxe de **INSERT INTO** avec cette fameuse clause :

Code : SQL

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

Donc, si une contrainte d'unicité est violée par la requête d'insertion, la clause **ON** DUPLICATE **KEY** va aller modifier les colonnes spécifiées dans la ligne déjà existante dans la table.

Exemple

Et revoici notre petit Spoutnik !

Code : SQL

```sql
SELECT *
FROM Animal
WHERE nom = 'Spoutnik';
```

Code : Console

```
+----+------+---------------------+----------+--------------+-----------+---------+
| id | sexe | date_naissance      | nom      | commentaires | espece_id | race_id |
+----+------+---------------------+----------+--------------+-----------+---------+
| 32 | male | 2009-07-26 11:52:00 | Spoutnik | NULL         |         3 |    NULL |
+----+------+---------------------+----------+--------------+-----------+---------+
```

Essayons d'insérer une autre tortue du nom de Spoutnik, mais cette fois-ci avec la nouvelle clause.

Code : SQL

```sql
INSERT INTO Animal (sexe, date_naissance, espece_id, nom, mere_id)
VALUES ('male', '2010-05-27 11:38:00', 3, 'Spoutnik', 52) --
date_naissance et mere_id sont différents du Spoutnik dans la table
ON DUPLICATE KEY UPDATE mere_id = 52;
```

Spoutnik est toujours là, mais il a désormais une mère ! Et son *id* n'a pas été modifié. Il s'agit donc bien d'une modification de la ligne, et non d'une suppression suivie d'une insertion. De même, sa date de naissance est restée la même puisque l'**UPDATE** ne portait que sur *mere_id*.

Code : Console

```
+----+------+---------------------+----------+--------------+-----------+---------+
| id | sexe | date_naissance      | nom      | commentaires | espece_id | race_id |
+----+------+---------------------+----------+--------------+-----------+---------+
| 32 | male | 2009-07-26 11:52:00 | Spoutnik | NULL         |         3 |    NULL |
+----+------+---------------------+----------+--------------+-----------+---------+
```

En fait, ce que nous avons fait était équivalent à la requête de modification suivante :

Code : SQL

```sql
UPDATE Animal
SET mere_id = 52
WHERE nom = 'Spoutnik';
```

Attention : plusieurs contraintes d'unicité sur la même table

Souvenez-vous, avec **REPLACE**, nous avons vu que s'il existait plusieurs contraintes d'unicité sur la même table, et que plusieurs lignes faisaient échouer l'insertion à cause de ces contraintes, **REPLACE** supprimait autant de lignes que nécessaire pour faire l'insertion.

Le comportement de `ON DUPLICATE KEY` est différent ! Dans le cas où plusieurs lignes seraient impliquées, seule une de ces lignes est modifiée (et impossible de prédire laquelle). Il faut donc **à tout prix** éviter d'utiliser cette clause quand plusieurs contraintes d'unicité pourraient être violées par l'insertion.
Hé bien voilà qui conclut cette partie ! Quelques petites fonctionnalités pratiques et souvent méconnues.

On commence à faire des choses plutôt sympathiques avec nos données, n'est-ce pas ? Cette partie a pu vous paraître un peu plus compliquée. Du coup, pour les prochaines, je vous prépare quelque chose de simple, et pourtant extrêmement utile !

Partie 3 : Fonctions : nombres, chaînes et agrégats

Maintenant que vous savez faire les opérations basiques sur vos données (insérer, lire, modifier, supprimer - les opérations du "CRUD"), les deux prochaines parties seront consacrées à la manipulation de ces données, en particulier grâce à l'utilisation des fonctions.
Cette partie-ci sera consacrée aux nombres et aux chaînes de caractères. Après une brève introduction sur les fonctions, vous découvrirez quelques fonctions bien utiles. Les fonctions d'agrégat clôtureront ensuite ce chapitre.

Rappels et introduction

Pour commencer en douceur, voici un chapitre d'introduction. On commence avec quelques **rappels** et astuces. Ensuite, on entre dans le vif du sujet avec la **définition d'une fonction**, et la différence entre une fonction **scalaire** et une fonction **d'agrégation**. Et pour finir, nous verrons quelques fonctions permettant d'obtenir des renseignements sur votre environnement, sur la dernière requête effectuée, et permettant de convertir des valeurs.

Que du bonheur, que du facile, que du super utile !

Rappels et manipulation simple de nombres

Rappels

Nous avons vu (il y a longtemps c'est vrai) qu'il était possible d'afficher des nombres ou des chaînes de caractères avec un simple `SELECT`:

Code : SQL

```sql
SELECT 3;

SELECT 'Bonjour !';
```

Code : Console

```
+---+
| 3 |
+---+
| 3 |
+---+

+-----------+
| Bonjour ! |
+-----------+
| Bonjour ! |
+-----------+
```

Vous savez également qu'il est possible de faire diverses opérations mathématiques de la même manière, et que la priorité des opérations est respectée :

Code : SQL

```sql
SELECT 3+5;

SELECT 8/3;

SELECT 10+2/2;

SELECT (10+2)/2;
```

Code : Console

```
+-----+
| 3+5 |
+-----+
|   8 |
+-----+

+--------+
| 8/3    |
+--------+
| 2.6667 |
+--------+

+---------+
| 10+2/2  |
+---------+
| 11.0000 |
+---------+

+-----------+
| (10+2)/2  |
+-----------+
|   6.0000  |
+-----------+
```

Opérateurs mathématiques

Six opérateurs mathématiques sont utilisables avec MySQL.

Symbole	Opération
+	addition
-	soustraction
*	multiplication
/	division
DIV	division entière
% (ou **MOD**)	modulo

Pour ceux qui ne le sauraient pas, le modulo de a par b est le reste de la division entière de a par b.

Code : SQL

```sql
SELECT 1+1, 4-2, 3*6, 5/2, 5 DIV 2, 5 % 2, 5 MOD 2;
```

Code : Console

```
+-----+-----+-----+--------+---------+-------+---------+
| 1+1 | 4-2 | 3*6 | 5/2    | 5 DIV 2 | 5 % 2 | 5 MOD 2 |
+-----+-----+-----+--------+---------+-------+---------+
|   2 |   2 |  18 | 2.5000 |       2 |     1 |       1 |
+-----+-----+-----+--------+---------+-------+---------+
```

Combiner les données avec des opérations mathématiques

Jusqu'ici, rien de plus simple. Cependant, si l'on veut juste faire 3+6, pas besoin de MySQL, une calculette (ou un cerveau) suffit. Après tout, dans une base de données, l'important, ce sont les données. Et, il est évidemment possible de faire des opérations mathématiques (et bien d'autres choses) sur les données.

Mais avant de rentrer dans le vif du sujet, je vais vous demander d'ajouter quelque chose à notre base de données. En effet, pour l'instant nous n'avons pas de données numériques, si ce n'est les différentes *id*. Or, il n'est pas très intéressant (et ça peut même être dangereux) de manipuler les *id*. Par conséquent, nous allons ajouter une colonne *pattes* à notre table *Espece*, qui contiendra le nombre de pattes de l'espèce. Nous sommes bien d'accord qu'en principe tout le monde sait que les chats ont 4 pattes, mais c'est l'exemple le plus facile à mettre en place qui me soit venu.
Voici donc les commandes à exécuter :

Code : SQL

```
ALTER TABLE Espece
ADD COLUMN pattes TINYINT NOT NULL DEFAULT 0; -- Par défaut à 0, on
ne sait jamais qu'on élève des serpents un de ces jours

UPDATE Espece
SET pattes = 4
WHERE nom_latin <> 'Alipiopsitta xanthops'; -- A part les
perroquets, nos espèces actuelles ont 4 pattes

UPDATE Espece
SET pattes = 2
WHERE nom_latin = 'Alipiopsitta xanthops';
```

Opérations sur données sélectionnées

Nous voilà prêts à commencer.
Imaginons que nous voulions savoir combien de pattes nous aurions en tout par espèce en prenant 13 individus de chaque. Facile, il suffit de multiplier le nombre de pattes de chaque espèce par 13 :

Code : SQL

```
SELECT nom_courant, pattes*13 AS pattes_totales
FROM Espece;
```

Code : Console

```
+-------------------+----------------+
| nom_courant       | pattes_totales |
+-------------------+----------------+
| Chien             |             52 |
| Chat              |             52 |
| Tortue d'Hermann  |             52 |
| Perroquet amazone |             26 |
+-------------------+----------------+
```

Toutes les opérations courantes peuvent bien entendu être utilisées.

Code : SQL

```
SELECT nom_courant, pattes+1 AS pattesPlusPlus, pattes/2 AS
demiPattes, pattes-1.3 AS pattesMoins, pattes%3 AS pattesModulo
FROM Espece;
```

Code : Console

```
+------------------+----------------+------------+------------+--------------+
| nom_courant      | pattesPlusPlus | demiPattes | pattesMoins| pattesModulo |
+------------------+----------------+------------+------------+--------------+
| Chien            |              5 |     2.0000 |        2.7 |            1 |
| Chat             |              5 |     2.0000 |        2.7 |            1 |
| Tortue d'Hermann |              5 |     2.0000 |        2.7 |            1 |
| Perroquet amazone|              3 |     1.0000 |        0.7 |            2 |
| Rat brun         |              5 |     2.0000 |        2.7 |            1 |
+------------------+----------------+------------+------------+--------------+
```

Modification de données grâce à des opérations mathématiques

Il est tout à fait possible, et souvent fort utile, de modifier des données grâce à des opérations. Par exemple, la commande suivante va augmenter le nombre de pattes de chaque espèce de 3 :

Code : SQL

```
UPDATE Espece
SET pattes = pattes+3;
```

C'est quand même plus élégant que de devoir faire une requête **SELECT** pour savoir combien de pattes ont vos espèces pour l'instant, calculer ces nombres +3, puis faire un **UPDATE** avec le résultat.
Et si cet exemple ne vous parle pas (ce que je peux concevoir, l'ajout de pattes étant courant à Naheulbeuk mais beaucoup moins sur notre bonne vieille planète Terre), en voici un qui devrait vous intéresser : vous êtes webmaster d'un super site internet, et vous voudriez mettre en place un système de statistiques. Vous voudriez notamment savoir combien de fois chaque page de votre site a été vue. Vous créez donc une table Pages_vues, avec 3 colonnes : id, page, nb_vues. Et chaque fois qu'un visiteur arrive sur une page, vous ajoutez 1 à nb_vues de la page en question.

Code : SQL

```
UPDATE Page_vues
SET nb_vues = nb_vues + 1
WHERE page = 'ma_page';
```

Trop facile !

Bien entendu, on peut faire ce genre de manipulation également dans une requête de type **INSERT INTO ... SELECT** par exemple. On peut faire ce genre de manipulation partout en fait. Je vous laisse faire vos propres essais.

Je vous invite, comme premier test, à soustraire trois à la colonne pattes pour chaque espèce, du moins si vous avez ajouté trois en exécutant l'exemple ci-dessus. Sauf si vous connaissez des chats à 7 pattes...

Définition d'une fonction

Vous vous souvenez peut-être que je vous ai déjà montré l'une ou l'autre fonction. Par exemple, dans le chapitre sur les sous-requêtes, je vous proposais la requête suivante :

Code : SQL

```
SELECT MIN(date_naissance)
FROM (
    SELECT Animal.*
    FROM Animal
    INNER JOIN Espece
        ON Espece.id = Animal.espece_id
    WHERE Espece.nom_courant IN ('Tortue d''Hermann', 'Perroquet
amazone')
) AS tortues_perroquets;
```

Lorsque l'on fait `MIN(date_naissance)`, on **appelle** la **fonction** `MIN()`, en lui donnant en **paramètre** la colonne *date_naissance* (ou plus précisément, les lignes de la colonne *date_naissance* sélectionnées par la requête)

Détaillons un peu tout ça !

Une fonction

Une fonction est un code qui effectue une série d'instructions bien précises (dans le cas de `MIN()`, ces instructions visent donc à chercher la valeur minimale), et renvoie le résultat de cette action (la valeur minimale en question).
Une fonction est définie par son nom (exemple : `MIN`) et ses paramètres.

Un paramètre

Un paramètre de fonction est une donnée (ou un ensemble de données) que l'on fournit à la fonction afin qu'elle puisse effectuer son action. Par exemple, pour `MIN()`, il faut passer un paramètre : les données parmi lesquelles on souhaite récupérer la valeur minimale. Une fonction peut avoir un ou plusieurs paramètre(s), ou n'en avoir aucun. Dans le cas d'une fonction ayant plusieurs paramètres, l'ordre dans lequel on donne ces paramètres est très important.

On parle aussi des **arguments** d'une fonction.

Appeler une fonction

Lorsque l'on utilise une fonction, on dit qu'on fait appel à celle-ci. Pour appeler une fonction, il suffit donc de donner son nom, suivi des paramètres éventuels entre parenthèses (lesquelles sont obligatoires, même s'il n'y a aucun paramètre).

Exemples

Code : SQL

```
-- Fonction sans paramètre
SELECT PI();    -- renvoie le nombre Pi, avec 5 décimales

-- Fonction avec un paramètre
SELECT MIN(pattes) AS minimum  -- il est bien sûr possible
d'utiliser les alias !
FROM Espece;

-- Fonction avec plusieurs paramètres
SELECT REPEAT('fort ! Trop ', 12);  -- répète une chaîne (ici :
'fort ! Trop ', répété 12 fois)

-- Même chose qu'au-dessus, mais avec les paramètres dans le mauvais
ordre
SELECT REPEAT(12, 'fort ! Trop '); -- la chaîne de caractères 'fort
! Trop ' va être convertie en entier par MySQL, ce qui donne 0. 12
va donc être répété zéro fois...
```

Code : Console

```
+----------+
| PI()     |
+----------+
| 3.141593 |
+----------+

+---------+
| minimum |
+---------+
|       2 |
+---------+

+-------------------------------------------------------------------------------
| REPEAT('fort ! Trop ', 12)
+-------------------------------------------------------------------------------
| fort ! Trop fort ! Trop fort ! Trop fort ! Trop fort ! Trop fort ! Trop fort ! Tr
+-------------------------------------------------------------------------------

+----------------------------+
| REPEAT(12, 'fort ! Trop ') |
+----------------------------+
|                            |
+----------------------------+
```

Pas d'espace entre le nom de la fonction et la parenthèse ouvrante !

Fonctions scalaires VS fonctions d'agrégation

On peut distinguer deux types de fonctions : les fonctions scalaires, et les fonctions d'agrégation (ou fonctions de groupement). Les fonctions scalaires s'appliquent à chaque ligne indépendamment, tandis que les fonctions d'agrégation regroupent les lignes (par défaut, elles regroupent toutes les lignes en une). Un petit exemple rendra cette explication lumineuse.

Fonction scalaire

La fonction `POWER(X, Y)` renvoie la valeur de `X` élévée à la puissance `Y`. Il s'agit d'une fonction scalaire.

Code : SQL

```sql
SELECT POWER(pattes, 2) -- on élève au carré
FROM Espece;
```

Code : Console

```
+------------------+
| POWER(pattes, 2) |
+------------------+
|               16 |
|               16 |
|               16 |
|                4 |
+------------------+
```

Il y a quatre espèces dans ma table, et lorsqu'on applique la fonction `POWER(X,Y)` à la colonne *pattes*, on récupère bien quatre lignes.

Fonction d'agrégation

La fonction **`MIN`**`(X)` par contre, est une fonction d'agrégation.

Code : SQL

```sql
SELECT MIN(pattes)
FROM Espece;
```

Code : Console

```
+--------------+
| MIN(pattes)  |
+--------------+
|            2 |
+--------------+
```

On ne récupère qu'une seule ligne de résultat. Les quatre espèces ont été regroupées (ça n'aurait d'ailleurs pas de sens d'avoir une ligne par espèce).
Cette particularité des fonctions d'agrégation les rend un peu plus délicates à utiliser, mais offre des possibilités vraiment intéressantes. Nous commencerons donc en douceur avec les fonctions scalaires pour consacrer ensuite plusieurs chapitres à l'utilisation des fonctions de groupement.

Quelques fonctions générales

Petite mise au point

Le but de cette partie n'est évidemment pas de référencer toutes les fonctions existantes. De même, les fonctions présentées seront décrites, avec exemples, mais nous ne verrons pas les petits cas particuliers, les exceptions, ni les éventuels comportements étranges et imprévisibles. Pour ça, la doc est et restera votre meilleur compagnon. Non, le but ici est de vous montrer un certain nombre de fonctions que, selon mon expérience, je juge utile que vous connaissiez. Par conséquent, et on ne le répètera jamais assez, n'hésitez pas à faire un tour sur la documentation officielle de MySQL si vous ne trouvez pas votre bonheur parmi les fonctions citées ici.

Informations sur l'environnement actuel

Version de MySQL

La fonction classique parmi les classiques : **`VERSION`**`()` vous permettra de savoir sous quelle version de MySQL tourne votre serveur.

Code : SQL

```sql
SELECT VERSION();
```

Code : Console

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

```
| VERSION() |
+-----------+
| 5.5.16    |
+-----------+
```

Où suis-je ? Qui suis-je ?

Vous avez créé plusieurs utilisateurs différents pour gérer votre base de données, et présentement vous ne savez plus avec lequel vous êtes connecté ? Pas de panique, il existe les fonctions **CURRENT_USER()** et **USER()**. Ces deux fonctions ne font pas exactement la même chose. Par conséquent, il n'est pas impossible qu'elles vous renvoient deux résultats différents.

- **CURRENT_USER()** : renvoie l'utilisateur (et l'hôte) qui a été utilisé lors de l'identification au serveur
- **USER()** : renvoie l'utilisateur (et l'hôte) qui a été spécifié lors de l'identification au serveur

Mais comment cela pourrait-il être différent ? Tout simplement, parce que si vous vous connectez avec un utilisateur qui n'a aucun droit (droits que l'on donne avec la commande **GRANT**, mais nous verrons ça dans une prochaine partie), vous arriverez à vous connecter, mais le serveur vous identifiera avec un "utilisateur anonyme". **USER()** vous renverra alors votre utilisateur sans droit, tandis que **CURRENT_USER()** vous donnera l'utilisateur anonyme.

Dans notre cas, l'utilisateur "sdz" (ou quel que soit le user que vous avez créé) ayant des droits, les deux fonctions renverront exactement la même chose.

Code : SQL

```sql
SELECT CURRENT_USER(), USER();
```

Code : Console

```
+----------------+---------------+
| CURRENT_USER() | USER()        |
+----------------+---------------+
| sdz@localhost  | sdz@localhost |
+----------------+---------------+
```

Informations sur la dernière requête

Dernière ID générée par auto-incrémentation

Dans une base de données relationnelle, il arrive très souvent que vous deviez insérer plusieurs lignes en une fois dans la base de données, et que certaines de ces nouvelles lignes doivent contenir une référence à d'autres nouvelles lignes. Par exemple, vous voulez ajouter Pipo le rottweiller dans votre base. Pour ce faire, vous devez insérer une nouvelle race (rottweiller), et un nouvel animal (Pipo) pour lequel vous avez besoin de l'*id* de la nouvelle race.
Plutôt que de faire un **SELECT** sur la table *Race* une fois la nouvelle race insérée, il est possible d'utiliser `LAST_INSERT_ID()`.
Cette fonction renvoie la dernière *id* créée par auto-incrémentation, pour la connexion utilisée (donc si quelqu'un se connecte au même serveur, avec un autre client, il n'influera pas sur le `LAST_INSERT_ID()` que vous recevez).

Code : SQL

```sql
INSERT INTO Race (nom, espece_id, description)
VALUES ('Rottweiller', 1, 'Chien d''apparence solide, bien musclé, à
```

```
    la robe noire avec des taches feu bien délimitées.');

    INSERT INTO Animal (sexe, date_naissance, nom, espece_id, race_id)
    SELECT 'male', '2010-11-05', 'Pipo', 1, LAST_INSERT_ID();  --
    LAST_INSERT_ID() renverra ici l'id de la race Rottweiller
```

Nombre de lignes renvoyées par la requête

La fonction `FOUND_ROWS()` vous permet d'afficher le nombre de lignes que votre dernière requête a ramenées.

Code : SQL

```
    SELECT * FROM Race;
    SELECT FOUND_ROWS();
```

Code : Console

```
+----+---------------------+-----------+-----------------------------------------
| id | nom                 | espece_id | description
+----+---------------------+-----------+-----------------------------------------
|  1 | Berger allemand     |         1 | Chien sportif et élégant au pelage dense,
|  2 | Berger blanc suisse |         1 | Petit chien au corps compact, avec des pat
|  3 | Singapura           |         2 | Chat de petite taille aux grands yeux en a
|  4 | Bleu russe          |         2 | Chat aux yeux verts et à la robe épaisse e
|  5 | Maine coon          |         2 | Chat de grande taille, à poils mi-longs.
|  7 | Sphynx              |         2 | Chat sans poils.
|  8 | Nebelung            |         2 | Chat bleu russe, mais avec des poils longs
|  9 | Rottweiller         |         1 | Chien d'apparence solide, bien musclé, à l
+----+---------------------+-----------+-----------------------------------------

+--------------+
| FOUND_ROWS() |
+--------------+
|            8 |
+--------------+
```

Jusque-là, rien de bien extraordinaire. Cependant, utilisée avec **`LIMIT`**, `FOUND_ROWS()` peut avoir un comportement très intéressant. En effet, moyennant l'ajout d'une option dans la requête **`SELECT`** d'origine, `FOUND_ROWS()` nous donnera le nombre de lignes que la requête aurait ramenées en l'absence de **`LIMIT`**. L'option à ajouter dans le **`SELECT`** est `SQL_CALC_FOUND_ROWS`, et se place juste après le mot-clé **`SELECT`**.

Code : SQL

```
    SELECT * FROM Race LIMIT 3;                         -- Sans option
    SELECT FOUND_ROWS() AS sans_option;

    SELECT SQL_CALC_FOUND_ROWS * FROM Race LIMIT 3;  -- Avec option
    SELECT FOUND_ROWS() AS avec_option;
```

Code : Console

```
+-------------+
| sans_option |
+-------------+
```

```
|           3 |
+-------------+

+-------------+
| avec_option |
+-------------+
|           8 |
+-------------+
```

Convertir le type de données

Dans certaines situations, vous allez vouloir convertir le type de votre donnée (une chaîne de caractères `'45'` en entier `45` par exemple). Il faut savoir que dans la majorité de ces situations, MySQL est assez souple et permissif pour faire la conversion lui-même, automatiquement, et sans que vous ne vous en rendiez vraiment compte (attention, ce n'est pas le cas de la plupart des SGBDR).

Exemple :

Code : SQL

```sql
SELECT *
FROM Espece
WHERE id = '3';

INSERT INTO Espece (nom_latin, nom_courant, description, pattes)
VALUES ('Rattus norvegicus', 'Rat brun', 'Petite bestiole avec de
longues moustaches et une longue queue sans poils', '4');
```

La colonne *id* est de type `INT`, pourtant la comparaison avec une chaîne de caractères renvoie bien un résultat. De même, la colonne *pattes* est de type `INT`, mais l'insertion d'une valeur sous forme de chaîne de caractères n'a posé aucun problème. MySQL a converti automatiquement.

Dans les cas où la conversion automatique n'est pas possible, vous pouvez utiliser la fonction **`CAST`**`(expr` **`AS type`**`)`. *expr* représente la donnée que vous voulez convertir, et *type* est bien sûr le type vers lequel vous voulez convertir votre donnée.

Ce type peut être : `BINARY`, `CHAR`, `DATE`, `DATETIME`, `TIME`, `UNSIGNED` (sous-entendu `INT`), `SIGNED` (sous-entendu `INT`), `DECIMAL`.

Exemple :

Code : SQL

```sql
SELECT CAST('870303' AS DATE);
```

Code : Console

```
+-------------------------+
| CAST('870303' AS DATE)  |
+-------------------------+
| 1987-03-03              |
+-------------------------+
```

Une utilisation intéressante de ce **`CAST`**`()` est le tri des données basé sur une colonne de type `ENUM`. En principe, l'ordre est basé sur l'ordre dans lequel les différentes valeurs de l'`ENUM` ont été définies. Mais si vous voulez que le tri se fasse par ordre alphabétique, il suffit de convertir la colonne en `CHAR`. Par exemple, vous pouvez comparer les résultats de ces deux requêtes :

Code : SQL

```sql
SELECT *
FROM Animal
WHERE Espece_id = 2
ORDER BY sexe;

SELECT *
FROM Animal
WHERE Espece_id = 2
ORDER BY CAST(sexe AS CHAR);
```

Voici donc juste de quoi vous mettre l'eau à la bouche en principe. Plongez donc rapidement dans le chapitre suivant, le fun ne fait que commencer !

Fonctions scalaires

Comme prévu, ce chapitre sera consacré aux **fonctions scalaires**, de manipulation de nombres et de chaînes de caractères.
Nous verrons entre autres comment arrondir un nombre ou tirer un nombre au hasard, et comment connaître la longueur d'un texte ou en extraire une partie.
La fin de ce chapitre est constituée d'exercices afin de pouvoir mettre les fonctions en pratique.

Il existe aussi des fonctions permettant de manipuler les dates en SQL, mais une partie entière leur sera consacrée.

A nouveau, si vous ne trouvez pas votre bonheur parmi les fonctions présentées ici, n'hésitez surtout pas à faire un tour sur la documentation officielle.
N'essayez pas de retenir par coeur toutes ces fonctions bien sûr. Si vous savez qu'elles existent, il vous sera facile de retrouver leur syntaxe exacte en cas de besoin, que ce soit ici ou sur la documentation officielle.

Manipulation de nombres

Toutes ces fonctions retournent **`NULL`** en cas d'erreur !

Arrondis

Voici quatre fonctions permettant d'arrondir les nombres, selon différents critères.
Pour les paramètres, *n* représente le nombre à arrondir, *d* le nombre de décimales désirées.

CEIL()

`CEIL(n)` ou `CEILING(n)` arrondit au nombre entier supérieur.

Code : SQL

```
SELECT CEIL(3.2), CEIL(3.7);
```

Code : Console

```
+-----------+-----------+
| CEIL(3.2) | CEIL(3.7) |
+-----------+-----------+
|         4 |         4 |
+-----------+-----------+
```

FLOOR()

`FLOOR(n)` arrondit au nombre entier inférieur.

Code : SQL

```
SELECT FLOOR(3.2), FLOOR(3.7);
```

Code : Console

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

```
| FLOOR(3.2) | FLOOR(3.7) |
+------------+------------+
|          3 |          3 |
+------------+------------+
```

ROUND()

`ROUND(n, d)` arrondit au nombre à *d* décimales le plus proche. `ROUND(n)` équivaut à écrire `ROUND(n, 0)`, donc arrondit à l'entier le plus proche.

Code : SQL

```sql
SELECT ROUND(3.22, 1), ROUND(3.55, 1), ROUND(3.77, 1);

SELECT ROUND(3.2), ROUND(3.5), ROUND(3.7);
```

Code : Console

```
+----------------+----------------+----------------+
| ROUND(3.22, 1) | ROUND(3.55, 1) | ROUND(3.77, 1) |
+----------------+----------------+----------------+
|            3.2 |            3.6 |            3.8 |
+----------------+----------------+----------------+

+------------+------------+------------+
| ROUND(3.2) | ROUND(3.5) | ROUND(3.7) |
+------------+------------+------------+
|          3 |          4 |          4 |
+------------+------------+------------+
```

Si le nombre se trouve juste entre l'arrondi supérieur et l'arrondi inférieur (par exemple si on arrondit 3.5 à un nombre entier), certaines implémentations vont arrondir vers le haut, d'autres vers le bas. Testez donc pour savoir dans quel cas est votre serveur (comme vous voyez, chez moi, c'est vers le haut).

TRUNCATE()

`TRUNCATE(n, d)` arrondit en enlevant purement et simplement les décimales en trop.

Code : SQL

```sql
SELECT TRUNCATE(3.2, 0), TRUNCATE(3.5, 0), TRUNCATE(3.7, 0);

SELECT TRUNCATE(3.22, 1), TRUNCATE(3.55, 1), TRUNCATE(3.77, 1);
```

Code : Console

```
+------------------+------------------+------------------+
| TRUNCATE(3.2, 0) | TRUNCATE(3.5, 0) | TRUNCATE(3.7, 0) |
+------------------+------------------+------------------+
|                3 |                3 |                3 |
+------------------+------------------+------------------+

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

```
| TRUNCATE(3.22, 1) | TRUNCATE(3.55, 1) | TRUNCATE(3.77, 1) |
+-------------------+-------------------+-------------------+
|               3.2 |               3.5 |               3.7 |
+-------------------+-------------------+-------------------+
```

Exposants et racines

Exposants

`POWER(n, e)` (ou `POW(n, e)`) retourne le résultat de *n* exposant *e* (n^e)

Code : SQL

```
SELECT POW(2, 5), POWER(5, 2);
```

Code : Console

```
+-----------+-------------+
| POW(2, 5) | POWER(5, 2) |
+-----------+-------------+
|        32 |          25 |
+-----------+-------------+
```

Racines

`SQRT(n)` donne la racine carrée de *n*.

Code : SQL

```
SELECT SQRT(4);
```

Code : Console

```
+---------+
| SQRT(4) |
+---------+
|       2 |
+---------+
```

Il n'existe pas de fonction particulière pour obtenir la racine n-ième d'un nombre pour $n > 2$. Cependant, pour ceux qui ne le sauraient pas : $\sqrt[n]{x} = x^{\frac{1}{n}}$. Donc, pour obtenir par exemple la racine cinquième de 32, il suffit de faire :

Code : SQL

```
SELECT POW(32, 1/5);
```

Code : Console

```
+--------------+
| POW(32, 1/5) |
+--------------+
|            2 |
+--------------+
```

Hasard

Le "vrai" hasard n'existe pas en informatique. Il est cependant possible de simuler le hasard, avec par exemple un générateur de nombres aléatoires. MySQL implémente un générateur de nombres aléatoires auquel vous pouvez faire appel en utilisant la fonction `RAND()`, qui retourne un nombre "aléatoire" entre 0 et 1.

Code : SQL

```sql
SELECT RAND();
```

Code : Console

```
+---------------------+
| rand()              |
+---------------------+
| 0.08678611469155748 |
+---------------------+
```

Cette fonction peut par exemple être utile pour trier des résultats de manière aléatoire :

Code : SQL

```sql
SELECT *
FROM Race
ORDER BY RAND();
```

Divers

SIGN()

`SIGN(n)` renvoie le signe du nombre *n*. Ou plus exactement, `SIGN(n)` renvoie -1 si *n* est négatif, 0 si *n* vaut 0, et 1 si *n* est positif.

Code : SQL

```sql
SELECT SIGN(-43), SIGN(0), SIGN(37);
```

Code : Console

```
+-----------+---------+----------+
| SIGN(-43) | SIGN(0) | SIGN(37) |
+-----------+---------+----------+
|        -1 |       0 |        1 |
+-----------+---------+----------+
```

ABS()

ABS(n) retourne la valeur absolue de *n*, donc sa valeur sans le signe.

Code : SQL

```sql
SELECT ABS(-43), ABS(0), ABS(37);
```

Code : Console

```
+----------+--------+---------+
| ABS(-43) | ABS(0) | ABS(37) |
+----------+--------+---------+
|       43 |      0 |      37 |
+----------+--------+---------+
```

MOD()

La fonction **MOD**(n, div) retourne le reste de la division entière de *n* par *div* (comme l'opérateur modulo)

Code : SQL

```sql
SELECT MOD(56, 10);
```

Code : Console

```
+-------------+
| MOD(56, 10) |
+-------------+
|           6 |
+-------------+
```

Manipulation de chaînes de caractères

Nous allons maintenant manipuler, triturer, examiner des chaînes de caractères. A toutes fins utiles, je rappelle en passant qu'en SQL, les chaînes de caractères sont entourées de guillemets (simples, mais les guillemets doubles fonctionnent avec MySQL également).

Longueur et comparaison

Connaître la longueur d'une chaîne

Trois fonctions permettent d'avoir des informations sur la longueur d'une chaîne de caractères. Chaque fonction calcule la longueur d'une manière particulière.

- **BIT_LENGTH**(chaine) : retourne le nombre de bits de la chaîne. Chaque caractère est encodé sur un ou plusieurs octets, chaque octet est représenté par huit bits.
- **CHAR_LENGTH**(chaine) : (ou **CHARACTER_LENGTH**()) retourne le nombre de caractères de la chaîne.
- **LENGTH**(chaine) : retourne le nombre d'octets de la chaîne. Chaque caractère étant représenté par un ou plusieurs octets (rappel : les caractères que vous pouvez utiliser dépendent de l'encodage choisi)

Code : SQL

```
SELECT BIT_LENGTH('texte'),
       CHAR_LENGTH('texte'),
       LENGTH('texte');  -- ici, ces caractères étant tous encodés
sur 1 octet, le résultat ne diffèrera pas de CHAR_LENGTH()
```

Code : Console

```
+--------------------+----------------------+-----------------+
| BIT_LENGTH('texte') | CHAR_LENGTH('texte') | LENGTH('texte') |
+--------------------+----------------------+-----------------+
|                 40 |                    5 |               5 |
+--------------------+----------------------+-----------------+
```

A priori, dans neuf cas sur dix au moins, la fonction qui vous sera utile est donc **CHAR_LENGTH**().

Comparer deux chaînes

La fonction STRCMP(chaine1, chaine2) compare les deux chaînes passées en paramètres et retourne 0 si les chaînes sont les mêmes, -1 si la première chaîne est classée avant dans l'ordre alphabétique et 1 dans le cas contraire.

Code : SQL

```
SELECT STRCMP('texte', 'texte'),
       STRCMP('texte','texte2'),
       STRCMP('chaine','texte'),
       STRCMP('texte', 'chaine'),
       STRCMP('texte3','texte24'); -- 3 est après 24 dans l'ordre
alphabétique
```

Code : Console

```
+--------------------------+--------------------------+--------------------------+-
| STRCMP('texte', 'texte') | STRCMP('texte','texte2') | STRCMP('chaine','texte') |
+--------------------------+--------------------------+--------------------------+-
|                        0 |                       -1 |                       -1 |
+--------------------------+--------------------------+--------------------------+-
```

Retrait et ajout de caractères

Répéter une chaîne

`REPEAT(c, n)` retourne le texte *c*, *n* fois.

Code : SQL

```
SELECT REPEAT('Ok ', 3);
```

Code : Console

```
+------------------+
| REPEAT('Ok ', 3) |
+------------------+
| Ok Ok Ok         |
+------------------+
```

Compléter/réduire une chaîne

Les fonctions `LPAD()` et `RPAD()` appliquées à une chaîne de caractères retournent cette chaîne en lui donnant une longueur particulière, donnée en paramètre. Si la chaîne de départ est trop longue, elle sera raccourcie, si elle est trop courte, des caractères seront ajoutés, à gauche de la chaîne pour `LPAD()`, à droite pour `RPAD()`.
Ces fonctions nécessitent trois paramètres : la chaîne à transformer (*texte*), la longueur désirée (*long*), et le caractère à ajouter si la chaîne est trop courte (*caract*).

Code : SQL

```
LPAD(texte, long, caract)
RPAD(texte, long, caract)
```

Code : SQL

```
SELECT LPAD('texte', 3, '@'),
       LPAD('texte', 7, '$'),
       RPAD('texte', 5, 'u'),
       RPAD('texte', 7, '*'),
       RPAD('texte', 3, '-');
```

Code : Console

```
+----------------------+----------------------+----------------------+----------
| LPAD('texte', 3, '@') | LPAD('texte', 7, '$') | RPAD('texte', 5, 'u') | RPAD('tex
+----------------------+----------------------+----------------------+----------
| tex                  | $$texte              | texte                | texte**
+----------------------+----------------------+----------------------+----------
```

Oter les caractères inutiles

Il peut arriver que certaines de vos données aient des caractères inutiles ajoutés avant et/ou après le texte intéressant. Dans ce

cas, il vous est possible d'utiliser la fonction **TRIM()**, qui va supprimer tous ces caractères. Cette fonction a une syntaxe un peu particulière que voici :

Code : SQL

```
TRIM([[BOTH | LEADING | TRAILING] [caract] FROM] texte);
```

Je rappelle que ce qui est entre crochet est facultatif.
On peut donc choisir entre trois options : **BOTH**, **LEADING** et **TRAILING**.

- **BOTH** : les caractères seront éliminés à l'avant, et à l'arrière du texte
- **LEADING** : seuls les caractères à l'avant de la chaîne seront supprimés
- **TRAILING** : seuls les caractères à l'arrière de la chaîne seront supprimés

Si aucune option n'est précisée, c'est **BOTH** qui est utilisé par défaut.

caract est la chaîne de caractères (ou le caractère unique) à éliminer en début et/ou fin de chaîne. Ce paramètre est facultatif : par défaut les espaces blancs seront supprimées. Et *texte* est bien sûr la chaîne de caractères à traiter.

Code : SQL

```
SELECT TRIM(' Tralala '),
       TRIM(LEADING FROM ' Tralala '),
       TRIM(TRAILING FROM ' Tralala '),

       TRIM('e' FROM 'eeeBouHeee'),
       TRIM(LEADING 'e' FROM 'eeeBouHeee'),
       TRIM(BOTH 'e' FROM 'eeeBouHeee'),

       TRIM('123' FROM '1234ABCD4321');
```

Code : Console

```
+---------------------+-----------------------------------+-----------------------
| TRIM('   Tralala  ') | TRIM(LEADING FROM '   Tralala  ') | TRIM(TRAILING FROM '
+---------------------+-----------------------------------+-----------------------
| Tralala             | Tralala                           |    Tralala
+---------------------+-----------------------------------+-----------------------
```

Récupérer une sous-chaîne

La fonction **SUBSTRING()** retourne une partie d'une chaîne de caractères. Cette partie est définie par un ou deux paramètres : *pos* (obligatoire), qui donne la position de début de la sous-chaîne, et *long* (facultatif) qui donne la longueur de la sous-chaîne désirée (si ce paramètre n'est pas précisé, toute la fin de la chaîne est prise).
Quatre syntaxes sont possibles :

- **SUBSTRING**(chaine, pos)
- **SUBSTRING**(chaine **FROM** pos)
- **SUBSTRING**(chaine, pos, long)
- **SUBSTRING**(chaine **FROM** pos **FOR** long)

Code : SQL

```
SELECT SUBSTRING('texte', 2), SUBSTRING('texte' FROM 3),
SUBSTRING('texte', 2, 3), SUBSTRING('texte' FROM 3 FOR 1);
```

Code : Console

```
+-----------------------+---------------------------+---------------------------+---
| SUBSTRING('texte', 2) | SUBSTRING('texte' FROM 3) | SUBSTRING('texte', 2, 3) | SU
+-----------------------+---------------------------+---------------------------+---
| exte                  | xte                       | ext                       | x
+-----------------------+---------------------------+---------------------------+---
```

Recherche et remplacement

Rechercher une chaîne de caractères

`INSTR()`, `LOCATE()` et **`POSITION()`** retournent la position de la première occurrence d'une chaîne de caractères *rech* dans une chaîne de caractères *chaine*. Ces trois fonctions ont chacune une syntaxe particulière :

- `INSTR(chaine, rech)`
- `LOCATE(rech, chaine)` - les paramètres sont inversés par rapport à `INSTR()`
- **`POSITION`**`(rech` **`IN`** `chaine)`

Si la chaîne de caractères rech n'est pas trouvée dans *chaine*, ces fonctions retournent 0. Par conséquent, la première lettre d'une chaîne de caractères est à la position 1 (alors que dans beaucoup de langages de programmation, on commence toujours à la position 0).

`LOCATE()` peut aussi accepter un paramètre supplémentaire : *pos*, qui définit la position dans la chaîne à partir de laquelle il faut rechercher *rech* : `LOCATE(rech, chaine, pos)`.

Code : SQL

```
SELECT INSTR('tralala', 'la'),
       POSITION('la' IN 'tralala'),
       LOCATE('la', 'tralala'),
       LOCATE('la', 'tralala', 5);
```

Code : Console

```
+-----------------------+-----------------------------+--------------------------+-
| INSTR('tralala', 'la') | POSITION('la' IN 'tralala') | LOCATE('la', 'tralala') |
+-----------------------+-----------------------------+--------------------------+-
|                      4 |                           4 |                        4 |
+-----------------------+-----------------------------+--------------------------+-
```

Changer la casse des chaînes

Les fonctions **LOWER**(chaine) et LCASE(chaine) mettent tous les lettres de *chaine* en minuscules, tandis que **UPPER**(chaine) et UCASE(chaine) mettent toutes les lettres en majuscules.

Code : SQL

```
SELECT LOWER('AhAh'), LCASE('AhAh'), UPPER('AhAh'), UCASE('AhAh');
```

Code : Console

```
+---------------+---------------+---------------+---------------+
| LOWER('AhAh') | LCASE('AhAh') | UPPER('AhAh') | UCASE('AhAh') |
+---------------+---------------+---------------+---------------+
| ahah          | ahah          | AHAH          | AHAH          |
+---------------+---------------+---------------+---------------+
```

Récupérer la partie gauche ou droite

LEFT(chaine, long) retourne les *long* premiers caractères de *chaine* en partant de la gauche, et **RIGHT**(chaine, long) fait la même chose en partant de la droite.

Code : SQL

```
SELECT LEFT('123456789', 5), RIGHT('123456789', 5);
```

Code : Console

```
+----------------------+-----------------------+
| LEFT('123456789', 5) | RIGHT('123456789', 5) |
+----------------------+-----------------------+
| 12345                | 56789                 |
+----------------------+-----------------------+
```

Inverser une chaîne

REVERSE(chaine) renvoie *chaine* en inversant les caractères.

Code : SQL

```
SELECT REVERSE('abcde');
```

Code : Console

```
+------------------+
| REVERSE('abcde') |
+------------------+
| edcba            |
+------------------+
```

Remplacer une partie par autre chose

Deux fonctions permettent de remplacer une partie d'une chaîne de caractères : **INSERT()** et **REPLACE()**.

- **INSERT**(chaine, pos, long, nouvCaract) : permet de sélectionner la partie à remplacer sur la base la position des caractères à remplacer. Les *long* caractères à partir de la position *pos* de la chaîne *chaine* seront remplacés par *nouvCaract*.
- **REPLACE**(chaine, oldCaract, nouvCaract) : permet de sélectionner la partie à remplacer sur base des caractères. Tous les caractères (ou sous-chaînes) *oldCaract* seront remplacés par *nouvCaract*

Code : SQL

```
SELECT INSERT('texte', 3, 2, 'blabla'), REPLACE('texte', 'e', 'a'),
REPLACE('texte', 'ex', 'ou');
```

Code : Console

```
+---------------------------------+-----------------------------+-------------------
| INSERT('texte', 3, 2, 'blabla') | REPLACE('texte', 'e', 'a') | REPLACE('texte', '
+---------------------------------+-----------------------------+-------------------
| teblablae                       | taxta                       | toute
+---------------------------------+-----------------------------+-------------------
```

Concaténation

Concaténer deux chaînes de caractères signifie les mettre bout à bout pour n'en faire qu'une seule. Deux fonctions scalaires permettent la concaténation : CONCAT() et CONCAT_WS(). Ces deux fonctions permettent de concaténer autant de chaînes que vous voulez, il suffit de toutes les passer en paramètres. Par conséquent, ces deux fonctions n'ont pas un nombre de paramètres défini.

- CONCAT(chaine1, chaine2, ...) : renvoie simplement une chaîne de caractères, résultat de la concaténation de toutes les chaînes passées en paramètre.
- CONCAT_WS(separateur, chaine1, chaine2) : même chose que CONCAT(), sauf que la première chaîne passée sera utilisée comme séparateur, donc placée entre chacune des autres chaînes passées en paramètres.

Code : SQL

```
SELECT CONCAT('My', 'SQL', '!'), CONCAT_WS('-', 'My', 'SQL', '!');
```

Code : Console

```
+-------------------------+---------------------------------+
| CONCAT('My', 'SQL', '!') | CONCAT_WS('-', 'My', 'SQL', '!') |
+-------------------------+---------------------------------+
```

```
| MySQL!                    | My-SQL-!                          |
+---------------------------+-----------------------------------+
```

FIELD(), une fonction bien utile pour le tri

La fonction `FIELD(rech, chaine1, chaine2, chaine3, ...)` recherche le premier argument (*rech*) parmi les arguments suivants (*chaine1, chaine2, chaine3, ...*) et retourne l'index auquel *rech* est trouvée (1 si *rech* = *chaine1*, 2 si *rech* = *chaine2*, ...). Si *rech* n'est pas trouvéd parmi les arguments, 0 est renvoyé.

Code : SQL

```sql
SELECT FIELD('Bonjour', 'Bonjour !', 'Au revoir', 'Bonjour', 'Au
revoir !');
```

Code : Console

```
+-----------------------------------------------------------------------+
| FIELD('Bonjour', 'Bonjour !', 'Au revoir', 'Bonjour', 'Au revoir !') |
+-----------------------------------------------------------------------+
|                                                                     3 |
+-----------------------------------------------------------------------+
```

Par conséquent, `FIELD` peut être utilisé pour définir un ordre arbitraire dans une clause **`ORDER BY`** de la manière suivante :

Code : SQL

```sql
SELECT *
FROM Espece
ORDER BY FIELD(nom_courant, 'Rat brun', 'Chat', 'Tortue d''Hermann',
'Chien', 'Perroquet amazone');
```

Code : Console

```
+----+------------------+----------------------+--------------------------------
| id | nom_courant      | nom_latin            | description
+----+------------------+----------------------+--------------------------------
|  5 | Rat brun         | Rattus norvegicus    | Petite bestiole avec de longues
|  2 | Chat             | Felis silvestris     | Bestiole à quatre pattes qui sau
|  3 | Tortue d'Hermann | Testudo hermanni     | Bestiole avec une carapace très
|  1 | Chien            | Canis canis          | Bestiole à quatre pattes qui aim
|  4 | Perroquet amazone| Alipiopsitta xanthops| Joli oiseau parleur vert et jaun
+----+------------------+----------------------+--------------------------------
```

Si vous ne mettez pas toutes les valeurs existantes de la colonne en argument de `FIELD()`, les lignes ayant les valeurs non mentionnées seront classées en premier (puisque `FIELD()` reverra 0).

Code ASCII

Les deux dernières fonctions que nous allons voir sont `ASCII()` et `CHAR()`, qui sont complémentaires. `ASCII(chaine)` renvoie le code ASCII du premier caractère de la chaîne passée en paramètre, tandis que `CHAR(ascii1, ascii2, ...)` retourne les caractères correspondant aux codes ASCII passés en paramètres (autant de paramètres qu'on veut). Les arguments passés à `CHAR()` seront convertis en entiers par MySQL.

Code : SQL

```
SELECT ASCII('T'), CHAR(84), CHAR('84', 84+32, 84.2);
```

Code : Console

```
+------------+----------+-------------------------+
| ASCII('T') | CHAR(84) | CHAR('84', 84+32, 84.2) |
+------------+----------+-------------------------+
|         84 | T        | TtT                     |
+------------+----------+-------------------------+
```

Exemples d'application et exercices

Ce chapitre fut fort théorique jusqu'à maintenant. Donc pour changer un peu, et vous réveiller, je vous propose de passer à la pratique, en utilisant les données de notre base *elevage*. Ces quelques exercices sont faisables en utilisant uniquement les fonctions et opérateurs mathématiques que je vous ai décrits dans ce chapitre.

On commence par du facile

1. Afficher une phrase donnant le nombre de pattes de l'espèce, pour chaque espèce

Par exemple, afficher "Un chat a 4 pattes.", ou autre phrase du genre, et ce pour les cinq espèces enregistrées.

Secret (cliquez pour afficher)

Code : SQL

```
SELECT CONCAT('Un(e) ', nom_courant, ' a ', pattes, ' pattes.') AS
Solution
FROM Espece;

-- OU

SELECT CONCAT_WS(' ','Un(e)', nom_courant, 'a', pattes, 'pattes.')
AS Solution
FROM Espece;
```

2. Afficher les chats dont la deuxième lettre du nom est un "a"

Secret (cliquez pour afficher)

Code : SQL

```
SELECT Animal.nom, Espece.nom_courant
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Espece.id
WHERE Espece.nom_courant = 'Chat'
AND SUBSTRING(nom, 2, 1) = 'a';
```

Puis on corse un peu

1. Afficher les noms des perroquets en remplaçant les "a" par "@" et les "e" par "3" pour en faire des Perroquet Kikoolol

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT REPLACE(REPLACE(nom, 'a', '@'), 'e', '3') AS Solution
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Espece.id
WHERE Espece.nom_courant LIKE 'Perroquet%';
```

Une petite explication s'impose avant de vous laisser continuer. Comme vous voyez, il est tout à fait possible d'imbriquer plusieurs fonctions. Le tout est de le faire correctement, et pour cela, il faut procéder par étape. Ici, vous voulez faire deux remplacements successifs dans une chaîne de caractères (en l'occurence, le nom des perroquets). Donc, vous effectuez un premier remplacement, en changeant les "a" par les "@" : `REPLACE(nom, 'a', '@')`. Ensuite, **sur la chaîne résultant de ce premier remplacement**, vous effectuez le second : `REPLACE(REPLACE(nom, 'a', '@'), 'e', '3')`. Logique non ?

2. Afficher les chiens dont le nom a un nombre pair de lettres

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT nom, nom_courant
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Espece.id
WHERE nom_courant = 'Chien'
AND CHAR_LENGTH(nom)%2 = 0;

-- OU

SELECT nom, nom_courant
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Espece.id
WHERE nom_courant = 'Chien'
AND CHAR_LENGTH(nom) MOD 2 = 0;

-- OU

SELECT nom, nom_courant
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Espece.id
WHERE nom_courant = 'Chien'
AND MOD(CHAR_LENGTH(nom),2) = 0;
```

Le nombre de lettres, c'était facile, il suffisait d'utiliser `CHAR_LENGTH()` (en l'occurence, `LENGHT()` aurait fonctionné aussi, mais ce ne sera pas toujours le cas partout. Prenez donc l'habitude d'utiliser la fonction correspondant à ce dont vous avez besoin).
Par contre, pour savoir si ce nombre est pair, je suppose que certains n'ont pas trouvé tout de suite. Tout simplement,

lorsqu'un nombre est pair, alors le reste d'une division entière de ce nombre par 2 est 0. Il fallait donc utiliser l'opérateur ou la fonction **modulo**.

Voilà pour les fonctions scalaires. A priori, rien de bien compliqué. Ca tombe bien, cela vous aura fait un échauffement avant d'attaquer les fonctions de groupement et leurs nombreuses possibilités.
Pas de questionnaire ici, vous avez assez travaillé avec les exercices !

Fonctions d'agrégation

Les fonctions d'agrégation, ou de groupement, sont donc des fonctions qui vont **regrouper les lignes**. Elles agissent sur une colonne, et renvoient un résultat unique pour toutes les lignes sélectionnées (ou pour chaque groupe de lignes, mais nous verrons cela plus tard).

Elles servent majoritairement à faire des **statistiques**, comme nous allons le voir dans la première partie de ce chapitre (compter des lignes, connaître une moyenne, trouver la valeur maximale d'une colonne,...).
Nous verrons ensuite la fonction `GROUP_CONCAT()`, qui comme son nom l'indique, est une fonction de groupement qui sert à **concaténer** des valeurs.

Fonctions statistiques

La majeure partie des fonctions d'agrégation va vous permettre de faire des statistiques sur vos données.

Nombre de lignes

La fonction **`COUNT()`** permet de savoir combien de lignes sont sélectionnée par la requête.

Code : SQL

```sql
-- Combien de races avons-nous ? --
-- ------------------------------
SELECT COUNT(*) AS nb_races
FROM Race;

-- Combien de chiens avons-nous ? --
-- ------------------------------
SELECT COUNT(*) AS nb_chiens
FROM Animal
INNER JOIN Espece ON Espece.id = Animal.espece_id
WHERE Espece.nom_courant = 'Chien';
```

Code : Console

```
+----------+
| nb_races |
+----------+
|        8 |
+----------+

+------------+
| nb_chiens |
+------------+
|         21 |
+------------+
```

COUNT(*) ou COUNT(colonne)

Vous l'avez vu, j'ai utilisé **`COUNT(*)`** dans les exemples ci-dessus. Cela signifie que l'on compte tout simplement les lignes, sans se soucier de ce qu'elles contiennent.

Par contre, si on utilise **`COUNT(colonne)`**, seules les lignes dont la valeur de *colonne* n'est pas **`NULL`** seront prises en compte.

Code : SQL

```sql
SELECT COUNT(race_id), COUNT(*)
FROM Animal;
```

Code : Console

```
+----------------+----------+
| COUNT(race_id) | COUNT(*) |
+----------------+----------+
|             31 |       60 |
+----------------+----------+
```

Il n'y a donc que 31 animaux sur nos 60 pour lesquels la race est définie.

Doublons

Comme dans une requête **SELECT** tout à fait banale, il est possible d'utiliser le mot-clé **DISTINCT** pour ne pas prendre en compte les doublons.

Code : SQL

```
SELECT COUNT(DISTINCT race_id)
FROM Animal;
```

Code : Console

```
+-------------------------+
| COUNT(DISTINCT race_id) |
+-------------------------+
|                       7 |
+-------------------------+
```

Parmi nos 31 animaux dont la race est définie, on trouve donc 7 races différentes.

Minimum et maximum

Nous avons déjà eu l'occasion de rencontrer la fonction **MIN**(x), qui retourne la plus petite valeur de *x*. Il existe également une fonction **MAX**(x), qui renvoie la plus grande valeur de *x*.

Code : SQL

```
SELECT MIN(pattes), MAX(pattes)
FROM Espece;
```

Code : Console

```
+-------------+-------------+
| MIN(pattes) | MAX(pattes) |
+-------------+-------------+
|           2 |           4 |
+-------------+-------------+
```

Notez que `MIN()` et `MAX()` ne s'utilisent pas que sur des données numériques. Si vous lui passez des chaînes de caractères, `MIN()` récupèrera la première chaîne dans l'ordre alphabétique, `MAX()` la dernière ; et avec des dates, `MIN()` renverra la plus vieille, `MAX()` la plus récente.

Code : SQL

```
SELECT MIN(nom), MAX(nom)
FROM Animal;

SELECT MIN(date_naisance), MAX(date_naissance)
FROM Animal;
```

Code : Console

```
+----------+----------+
| MIN(nom) | MAX(nom) |
+----------+----------+
| Anya     | Zonko    |
+----------+----------+

+---------------------+---------------------+
| MIN(date_naissance) | MAX(date_naissance) |
+---------------------+---------------------+
| 2006-03-15 14:26:00 | 2010-11-09 00:00:00 |
+---------------------+---------------------+
```

Somme et moyenne

Somme

La fonction `SUM(x)` renvoie la somme de *x*.

Code : SQL

```
SELECT SUM(pattes)
FROM Espece;
```

Code : Console

```
+-------------+
| SUM(pattes) |
+-------------+
|          18 |
+-------------+
```

Moyenne

La fonction `AVG(x)` (du mot anglais *average*) renvoie la valeur moyenne de *x*.

Code : SQL

```sql
SELECT AVG(pattes)
FROM Espece;
```

Code : Console

```
+--------------+
| AVG(pattes)  |
+--------------+
|       3.6000 |
+--------------+
```

Concaténation

Principe

Avec les fonctions d'agrégation, on regroupe plusieurs lignes. Les fonctions statistiques nous permettent d'avoir des informations fort utiles sur le résultat d'une requête, mais parfois, il est intéressant d'avoir également les valeurs concernées. Par exemple, on récupère la somme des pattes de chaque espèce, et on affiche les espèces concernées par la même occasion.
Ceci est faisable avec `GROUP_CONCAT(nomColonne)`. Cette fonction concatène les valeurs de *nomColonne* pour chaque groupement réalisé.

Exemple

Code : SQL

```sql
SELECT SUM(pattes), GROUP_CONCAT(nom_courant)
FROM Espece;
```

Code : Console

```
+-------------+-------------------------------------------------------+
| SUM(pattes) | GROUP_CONCAT(nom_courant)                             |
+-------------+-------------------------------------------------------+
|          18 | Chien,Chat,Tortue d'Hermann,Perroquet amazone,Rat brun |
+-------------+-------------------------------------------------------+
```

Syntaxe

Voici la syntaxe de cette fonction :

Code : SQL

```sql
GROUP_CONCAT([DISTINCT] col1 [, col2, ...]
             [ORDER BY col [ASC | DESC]]
             [SEPARATOR sep])
```

- **DISTINCT** : sert comme d'habitude à éliminer les doublons.
- `col1` : est le nom de la colonne dont les valeurs doivent être concaténées. C'est le **seul argument obligatoire**.
- `col2, ...` : sont les éventuelles autres colonnes (ou chaînes de caractères) à concaténer.
- **ORDER BY** : permet de déterminer dans quel ordre les valeurs seront concaténées.
- `SEPARATOR` : permet de spécifier une chaîne de caractères à utiliser pour séparer les différentes valeurs. Par défaut, c'est

une virgule.

Exemples

Code : SQL

```sql
--------------------
-- LE PLUS SIMPLE --
--------------------
SELECT SUM(pattes), GROUP_CONCAT(nom_latin)
FROM Espece;

---------------------------------------
-- CONCATENATION DE PLUSIEURS COLONNES --
---------------------------------------
SELECT SUM(pattes), GROUP_CONCAT(nom_latin, nom_courant)
FROM Espece;

-----------------------------------------------------
-- CONCATENATION DE PLUSIEURS COLONNES EN PLUS JOLI --
-----------------------------------------------------
SELECT SUM(pattes), GROUP_CONCAT(nom_latin, ' (', nom_courant, ')')
FROM Espece;

-----------------------------
-- ELIMINATION DES DOUBLONS --
-----------------------------
SELECT SUM(pattes), GROUP_CONCAT(DISTINCT Espece.nom_courant) --
Essayez sans le DISTINCT pour voir
FROM Espece
INNER JOIN Race ON Race.espece_id = Espece.id;

----------------------------
-- UTILISATION DE ORDER BY --
----------------------------
SELECT SUM(pattes), GROUP_CONCAT(nom_latin, ' (', nom_courant, ')'
ORDER BY nom_courant DESC)
FROM Espece;

-------------------------------
-- CHANGEMENT DE SEPARATEUR --
-------------------------------
SELECT SUM(pattes), GROUP_CONCAT(nom_latin, ' (', nom_courant, ')'
SEPARATOR ' - ')
FROM Espece;
```

Maintenant que vous connaissez les principales fonctions d'agrégation, nous allons pouvoir passer aux choses sérieuses dans cette partie. Soyez prêts, la puissance des groupements va bientôt vous être révélée.

Regroupement

Vous savez donc que les fonctions d'agrégation groupent plusieurs lignes ensemble. Jusqu'à maintenant, toutes les lignes étaient à chaque fois regroupées en une seule. Mais ce qui est particulièrement intéressant avec ces fonctions, c'est qu'il est possible de **regrouper les lignes en fonction d'un critère**, et d'avoir ainsi plusieurs groupes distincts.
Par exemple, avec la fonction `COUNT(*)`, vous pouvez compter le nombre de lignes que vous avez dans la table *Animal*. Mais que diriez-vous de faire des groupes par espèces, et donc de savoir en une seule requête combien vous avez de chats, chiens, etc ? Un simple groupement, et c'est fait !

Au programme de ce chapitre :

- les règles et la syntaxe à appliquer pour regrouper des lignes ;
- le groupement sur plusieurs critères ;
- les "super-agrégats" ;
- la sélection de certains groupes sur base de critères.

Regroupement sur un critère

Les mots-clés permettant de grouper les lignes selon un critères sont `GROUP BY`, et sont extrêmement faciles à utiliser. Il suffit de les placer après l'éventuelle clause `WHERE`, suivis du nom de la (ou des) colonne(s) à utiliser comme critère(s) de regroupement.

Exemples

Code : SQL

```sql
-- On compte les animaux, groupés par espèce
SELECT COUNT(*) AS nb_animaux
FROM Animal
GROUP BY espece_id;

-- On compte les mâles, groupés par espèce
SELECT COUNT(*) AS nb_males
FROM Animal
WHERE sexe = 'male'
GROUP BY espece_id;
```

Code : Console

```
+-------------+
| nb_animaux |
+-------------+
|          21 |
|          20 |
|          15 |
|           4 |
+-------------+

+-----------+
| nb_males |
+-----------+
|        10 |
|         9 |
|         4 |
|         3 |
+-----------+
```

C'est déjà bien sympathique, mais nous n'allons pas en rester là. Parce qu'il serait quand même intéressant de savoir à quelle espèce correspond quel nombre !

Voir d'autres colonnes

Pour savoir à quoi correspond chaque nombre, il suffit d'afficher également le critère qui a permis de regrouper les lignes. Dans notre cas, espece_id.

Code : SQL

```sql
SELECT espece_id, COUNT(*) AS nb_animaux
FROM Animal
GROUP BY espece_id;
```

Code : Console

```
+-----------+------------+
| espece_id | nb_animaux |
+-----------+------------+
|         1 |         21 |
|         2 |         20 |
|         3 |         15 |
|         4 |          4 |
+-----------+------------+
```

C'est déjà mieux, mais l'idéal serait d'avoir le nom des espèces directement. Qu'à cela ne tienne, il suffit de faire une jointure ! Sans oublier de changer le critère et de mettre *nom_courant* à la place de *espece_id*.

Code : SQL

```sql
SELECT nom_courant, COUNT(*) AS nb_animaux
FROM Animal
INNER JOIN Especе ON Animal.espece_id = Especе.id
GROUP BY nom_courant;
```

Code : Console

```
+-------------------+------------+
| nom_courant       | nb_animaux |
+-------------------+------------+
| Chat              |         20 |
| Chien             |         21 |
| Perroquet amazone |          4 |
| Tortue d'Hermann  |         15 |
+-------------------+------------+
```

La règle SQL

Colonnes sélectionnées

Lorsque l'on fait un groupement dans une requête, avec **GROUP BY**, on ne peut sélectionner que deux types d'éléments dans la clause **SELECT** :

- Une ou des colonne(s) ayant servi de critère pour le regroupement
- Une fonction d'agrégation (agissant sur n'importe quelle colonne)

Cette règle est d'ailleurs logique. Imaginez la requête suivante :

Code : SQL

```
SELECT nom_courant, COUNT(*) AS nb_animaux, date_naissance
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Espece.id
GROUP BY nom_courant;
```

Que vient faire la date de naissance dans cette histoire ? Et surtout, quelle date de naissance espère-t-on sélectionner ? Chaque ligne représente une espèce puisque l'on a regroupé les lignes sur base de *Espece.nom_courant*. Donc *date_naissance* n'a aucun sens par rapport aux groupes formés, une espèce n'ayant pas de date de naissance. Il en est de même pour les colonnes *sexe* ou *commentaires* par exemple.

Mais qu'en est-il des colonnes *Espece.id*, *Animal.espece_id*, ou *Espece.nom_latin* ? En groupant sur le nom courant de l'espèce, ces différentes colonnes ont un sens, et pourraient donc être utiles. Il a cependant été décidé que par sécurité, la sélection de colonnes n'étant pas dans les critères de groupement serait interdite. Cela afin d'éviter les situations comme au-dessus, où les colonnes sélectionnées n'ont aucun sens par rapport au groupement fait. Par conséquent, si vous voulez par exemple afficher également l'id de l'espèce et son nom latin, il vous faudra grouper sur trois critères : *Espece.nom_latin*, *Espece.nom_courant* et *Espece.id*. Les groupes créés seront les mêmes qu'en groupant uniquement sur *Espece.nom_courant*, mais votre requête respectera les standards SQL.

Code : SQL

```
SELECT Espece.id, nom_courant, nom_latin, COUNT(*) AS nb_animaux
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Espece.id
GROUP BY nom_courant, Espece.id, nom_latin;
```

Code : Console

```
+----+------------------+----------------------+------------+
| id | nom_courant      | nom_latin            | nb_animaux |
+----+------------------+----------------------+------------+
|  2 | Chat             | Felis silvestris     |         20 |
|  1 | Chien            | Canis canis          |         21 |
|  4 | Perroquet amazone | Alipiopsitta xanthops |          4 |
|  3 | Tortue d'Hermann | Testudo hermanni     |         15 |
+----+------------------+----------------------+------------+
```

Le cas MySQL

On ne le répètera jamais assez : MySQL est un SGBD extrêmement permissif. Dans certains cas, c'est bien pratique, mais c'est toujours dangereux.
Et notamment en ce qui concerne **GROUP BY**, MySQL ne sera pas perturbé pour un sou si vous sélectionnez une colonne qui n'est pas dans les critères de regroupement. Reprenons la requête qui sélectionne la colonne *date_naissance* alors que le regroupement se fait sur base de l'*espece_id*. J'insiste, cette requête ne respecte pas la norme SQL, et n'a aucun sens. La plupart des SGBD vous renverront une erreur si vous tentez de l'exécuter.

Code : SQL

```
SELECT espece_id, COUNT(*) AS nb_animaux, date_naissance
FROM Animal
GROUP BY espece_id;
```

Pourtant, loin de rouspéter, MySQL donne le résultat suivant :

Code : Console

```
+-----------+------------+---------------------+
| espece_id | nb_animaux | date_naissance      |
+-----------+------------+---------------------+
|         1 |         21 | 2010-04-05 13:43:00 |
|         2 |         20 | 2010-03-24 02:23:00 |
|         3 |         15 | 2009-08-03 05:12:00 |
|         4 |          4 | 2007-03-04 19:36:00 |
+-----------+------------+---------------------+
```

MySQL a tout simplement pris n'importe quelle valeur parmi celles du groupe pour la date de naissance. D'ailleurs, il est tout à fait possible que vous ayez obtenu des valeurs différentes des miennes.
Soyez donc très prudents lorsque vous utilisez **GROUP BY**. Vous faites peut-être des requêtes qui n'ont aucun sens, et MySQL ne vous en avertira pas !

Tri des données

Par défaut dans MySQL, les données sont triées sur base du critère de regroupement. C'est la raison pour laquelle, dans la requête précédente, la colonne *nom_courant* est triée par ordre alphabétique : c'est le premier critère de regroupement. MySQL permet d'utiliser les mots-clés **ASC** et **DESC** dans une clause **GROUP BY** pour choisir un tri ascendant (par défaut) ou descendant.

Code : SQL

```
SELECT Espece.id, nom_courant, nom_latin, COUNT(*) AS nb_animaux
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Espece.id
GROUP BY nom_courant DESC, Espece.id, nom_latin;
```

Code : Console

```
+----+------------------+----------------------+------------+
| id | nom_courant      | nom_latin            | nb_animaux |
+----+------------------+----------------------+------------+
|  3 | Tortue d'Hermann | Testudo hermanni     |         15 |
|  4 | Perroquet amazone | Alipiopsitta xanthops |          4 |
|  1 | Chien            | Canis canis          |         21 |
|  2 | Chat             | Felis silvestris     |         20 |
+----+------------------+----------------------+------------+
```

Mais rien n'empêche d'utiliser une clause **ORDER BY** classique, après la clause **GROUP BY**. L'**ORDER BY** sera prioritaire sur l'ordre défini par la clause de regroupement.

Dans le même ordre d'idées que pour les colonnes sélectionnées, il n'est possible de faire un tri des données qu'à partir d'une colonne étant parmi les critères de regroupement, ou à partir d'une fonction d'agrégation. Ça n'a pas plus de sens de trier les espèces par date de naissance que de sélectionner une date de naissance par espèce.

Vous pouvez par contre parfaitement faire ceci :

Code : SQL

```sql
SELECT Espece.id, nom_courant, nom_latin, COUNT(*) AS nb_animaux
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Espece.id
GROUP BY nom_courant, Espece.id, nom_latin
ORDER BY nb_animaux;
```

Code : Console

```
+----+------------------+----------------------+------------+
| id | nom_courant      | nom_latin            | nb_animaux |
+----+------------------+----------------------+------------+
|  4 | Perroquet amazone | Alipiopsitta xanthops |          4 |
|  3 | Tortue d'Hermann  | Testudo hermanni      |         15 |
|  2 | Chat              | Felis silvestris      |         20 |
|  1 | Chien             | Canis canis           |         21 |
+----+------------------+----------------------+------------+
```

Notez que la norme SQL veut que l'on n'utilise pas d'expressions dans `GROUP BY` ou `ORDER BY`. C'est la raison pour laquelle j'ai mis `ORDER BY nb_animaux` et non pas `ORDER BY COUNT(*)`, bien qu'avec MySQL, les deux fonctionnent. Pensez donc à utiliser les alias pour ces situations.

Et les autres espèces ?

La colonne suivante nous donne le nombre d'animaux qu'on possède pour chaque espèce **dont on possède au moins un animal**. Comment peut-on faire pour afficher également les autres espèces ?

Code : SQL

```sql
SELECT nom_courant, COUNT(*) AS nb_animaux
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Espece.id
GROUP BY nom_courant;
```

Essayons donc avec une jointure externe, puisqu'il faut tenir compte de toutes les espèces, même celles qui n'ont pas de correspondance dans la table Animal.

Code : SQL

```sql
SELECT nom_courant, COUNT(*) AS nb_animaux
FROM Animal
RIGHT JOIN Espece ON Animal.espece_id = Espece.id -- RIGHT puisque
la table Espece est à droite.
GROUP BY nom_courant;
```

Code : Console

```
+------------------+------------+
| nom_courant      | nb_animaux |
+------------------+------------+
```

```
| Chat               |          20 |
| Chien              |          21 |
| Perroquet amazone  |           4 |
| Rat brun           |           1 |
| Tortue d'Hermann   |          15 |
+--------------------+-------------+
```

Les rats bruns apparaissent bien, mais ce n'est pas 1 qu'on attend, mais 0, puisqu'on n'a pas de rats dans notre élevage. Cela dit, ce résultat est logique : avec la requête externe, on aura une ligne correspondant aux rats bruns, avec **NULL** dans toutes les colonnes de la table Animal. Donc ce qu'il faudrait, c'est avoir les cinq espèces, mais ne compter que lorsqu'il y a un animal correspondant. Pour ce faire, il suffit de faire **COUNT**(Animal.espece_id) par exemple.

Code : SQL

```sql
SELECT nom_courant, COUNT(espece_id) AS nb_animaux
FROM Animal
RIGHT JOIN Espece ON Animal.espece_id = Espece.id
GROUP BY nom_courant;
```

Code : Console

```
+--------------------+-------------+
| nom_courant        | nb_animaux  |
+--------------------+-------------+
| Chat               |          20 |
| Chien              |          21 |
| Perroquet amazone  |           4 |
| Rat brun           |           0 |
| Tortue d'Hermann   |          15 |
+--------------------+-------------+
```

C'est pas magique ça ? 

Regroupement sur plusieurs critères

J'ai mentionné le fait qu'il était possible d'utiliser plusieurs critères de sélection, mais jusqu'à maintenant, les critères multiples n'ont servi qu'à pouvoir afficher correctement les colonnes voulues, sans que ça influe sur les groupes. Voyons maintenant un exemple avec deux critères différents (dans le sens qu'ils ne créent pas les mêmes groupes).

Les deux requêtes suivantes permettent de savoir combien d'animaux de chaque espèce vous avez dans la table Animal, ainsi que combien de mâles et de femelles, toutes espèces confondues.

Code : SQL

```sql
SELECT nom_courant, COUNT(*) as nb_animaux
FROM Animal
INNER JOIN Espece ON Espece.id = Animal.espece_id
GROUP BY nom_courant;

SELECT sexe, COUNT(*) as nb_animaux
FROM Animal
GROUP BY sexe;
```

Code : Console

```
+--------------------+-------------+
| nom_courant        | nb_animaux  |
```

```
+-------------------+------------+
| Chat              |         20 |
| Chien             |         21 |
| Perroquet amazone |          4 |
| Tortue d'Hermann  |         15 |
+-------------------+------------+

+---------+------------+
| sexe    | nb_animaux |
+---------+------------+
| NULL    |          3 |
| male    |         26 |
| femelle |         31 |
+---------+------------+
```

En faisant un regroupement multi-critères, il est possible de savoir facilement combien de mâles et de femelles **par espèce**, il y a dans la table Animal. Notez que l'ordre des critères a son importance.

Code : SQL

```sql
-- On regroupe d'abord sur l'espèce, puis sur le sexe
SELECT nom_courant, sexe, COUNT(*) as nb_animaux
FROM Animal
INNER JOIN Espece ON Espece.id = Animal.espece_id
GROUP BY nom_courant, sexe;
```

Code : Console

```
+-------------------+---------+------------+
| nom_courant       | sexe    | nb_animaux |
+-------------------+---------+------------+
| Chat              | NULL    |          2 |
| Chat              | male    |          9 |
| Chat              | femelle |          9 |
| Chien             | male    |         10 |
| Chien             | femelle |         11 |
| Perroquet amazone | male    |          3 |
| Perroquet amazone | femelle |          1 |
| Tortue d'Hermann  | NULL    |          1 |
| Tortue d'Hermann  | male    |          4 |
| Tortue d'Hermann  | femelle |         10 |
+-------------------+---------+------------+
```

Code : SQL

```sql
-- On regroupe d'abord sur le sexe, puis sur l'espèce
SELECT nom_courant, sexe, COUNT(*) as nb_animaux
FROM Animal
INNER JOIN Espece ON Espece.id = Animal.espece_id
GROUP BY sexe,nom_courant;
```

Code : Console

```
+-------------------+---------+------------+
| nom_courant       | sexe    | nb_animaux |
+-------------------+---------+------------+
```

```
| Chat               | NULL    |          2 |
| Tortue d'Hermann   | NULL    |          1 |
| Chat               | male    |          9 |
| Chien              | male    |         10 |
| Perroquet amazone  | male    |          3 |
| Tortue d'Hermann   | male    |          4 |
| Chat               | femelle |          9 |
| Chien              | femelle |         11 |
| Perroquet amazone  | femelle |          1 |
| Tortue d'Hermann   | femelle |         10 |
+--------------------+---------+------------+
```

Etant donné que le regroupement par sexe donnait trois groupes différents, et le regroupement par espèce donnait quatre groupes différents, il peut y avoir jusqu'à douze (3 x 4) groupes lorsque l'on regroupe en se basant sur les deux critères. Ici, il y en aura moins puisque le sexe de tous les chiens et tous les perroquets est défini (pas de **NULL**)

Super-agrégats

Parlons maintenant de l'option **WITH ROLLUP** de **GROUP BY**. Cette option va afficher des lignes supplémentaires dans la table de résultats. Ces lignes représenteront des "super-groupes" (ou super-agrégats), donc des "groupes de groupes". Deux petits exemples, et vous aurez compris !

Exemple avec un critère de regroupement

Code : SQL

```sql
SELECT nom_courant, COUNT(*) as nb_animaux
FROM Animal
INNER JOIN Espece ON Espece.id = Animal.espece_id
GROUP BY nom_courant WITH ROLLUP;
```

Code : Console

```
+--------------------+------------+
| nom_courant        | nb_animaux |
+--------------------+------------+
| Chat               |         20 |
| Chien              |         21 |
| Perroquet amazone  |          4 |
| Tortue d'Hermann   |         15 |
| NULL               |         60 |
+--------------------+------------+
```

Nous avons donc 20 chats, 21 chiens, 4 perroquets et 15 tortues. Et combien font 20 + 21 + 4 + 15 ? 60 ! Exactement. La ligne supplémentaire représente donc le regroupement de nos quatre groupes basé sur le critère **GROUP BY** `nom_courant`.

Exemple avec deux critères de regroupement

Code : SQL

```sql
SELECT nom_courant, sexe, COUNT(*) as nb_animaux
FROM Animal
INNER JOIN Espece ON Espece.id = Animal.espece_id
WHERE sexe IS NOT NULL
GROUP BY nom_courant, sexe WITH ROLLUP;
```

Code : Console

```
+-------------------+---------+------------+
| nom_courant       | sexe    | nb_animaux |
+-------------------+---------+------------+
| Chat              | male    |          9 |
| Chat              | femelle |          9 |
| Chat              | NULL    |         18 |
| Chien             | male    |         10 |
| Chien             | femelle |         11 |
| Chien             | NULL    |         21 |
| Perroquet amazone | male    |          3 |
| Perroquet amazone | femelle |          1 |
| Perroquet amazone | NULL    |          4 |
| Tortue d'Hermann  | male    |          4 |
| Tortue d'Hermann  | femelle |         10 |
| Tortue d'Hermann  | NULL    |         14 |
| NULL              | NULL    |         57 |
+-------------------+---------+------------+
```

Les deux premières lignes correspondent aux nombres de chats males et femelles. Jusque-là, rien de nouveau. Par contre, la troisième ligne est une ligne insérée par **WITH ROLLUP**, et contient le nombre de chats (males et femelles). Nous avons fait des groupes en séparant les espèces et les sexes, et **WITH ROLLUP** a créé des "super-groupes" en regroupant les sexes mais gardant les espèces séparées.
Nous avons donc également le nombre de chiens à la sixième ligne, de perroquets à la neuvième, et de tortues à la douzième. Quant à la toute dernière ligne, c'est un "super-super-groupe" qui réunit tous les groupes ensembles.

Et c'est en utilisant **WITH ROLLUP** qu'on se rend compte que l'ordre des critères est vraiment important. En effet, voyons ce qu'il se passe si on échange les deux critères *nom_courant* et *sexe*.

Code : SQL

```
SELECT nom_courant, sexe, COUNT(*) as nb_animaux
FROM Animal
INNER JOIN Espece ON Espece.id = Animal.espece_id
WHERE sexe IS NOT NULL
GROUP BY sexe, nom_courant WITH ROLLUP
```

Code : Console

```
+-------------------+---------+------------+
| nom_courant       | sexe    | nb_animaux |
+-------------------+---------+------------+
| Chat              | male    |          9 |
| Chien             | male    |         10 |
| Perroquet amazone | male    |          3 |
| Tortue d'Hermann  | male    |          4 |
| NULL              | male    |         26 |
| Chat              | femelle |          9 |
| Chien             | femelle |         11 |
| Perroquet amazone | femelle |          1 |
| Tortue d'Hermann  | femelle |         10 |
| NULL              | femelle |         31 |
| NULL              | NULL    |         57 |
+-------------------+---------+------------+
```

Cette fois-ci, les super-groupes ne correspondent pas aux espèces, mais aux sexes, c'est-à-dire au premier critère. Le regroupement se fait bien dans l'ordre donné par les critères.

J'ai ajouté la condition **WHERE** `sexe` **IS NOT NULL** pour des raisons de lisibilité uniquement, étant donné que sans cette condition, d'autres **NULL** seraient apparus, rendant plus compliquée l'explication des super-agrégats.

Conditions sur les fonctions d'agrégation

Il n'est pas possible d'utiliser la clause **WHERE** pour faire des conditions sur une fonction d'aggrégations. Donc, si l'on veut afficher les espèces dont on possède plus de 15 individus, la requête suivante ne fonctionnera pas.

Code : SQL

```sql
SELECT nom_courant, COUNT(*)
FROM Animal
INNER JOIN Espece ON Espece.id = Animal.espece_id
WHERE COUNT(*) > 15
GROUP BY nom_courant;
```

Code : Console

```
ERROR 1111 (HY000): Invalid use of group function
```

Il faut utiliser une clause spéciale : **HAVING**. Cette clause se place juste après le **GROUP BY**.

Code : SQL

```sql
SELECT nom_courant, COUNT(*)
FROM Animal
INNER JOIN Espece ON Espece.id = Animal.espece_id
GROUP BY nom_courant
HAVING COUNT(*) > 15;
```

Code : Console

```
+-------------+----------+
| nom_courant | COUNT(*) |
+-------------+----------+
| Chat        |       20 |
| Chien       |       21 |
+-------------+----------+
```

Il est également possible d'utiliser un alias dans une condition **HAVING** :

Code : SQL

```sql
SELECT nom_courant, COUNT(*) as nombre
FROM Animal
INNER JOIN Espece ON Espece.id = Animal.espece_id
GROUP BY nom_courant
HAVING nombre > 15;
```

Code : Console

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

```
| nom_courant | nombre |
+-------------+--------+
| Chat        |     20 |
| Chien       |     21 |
+-------------+--------+
```

Optimisation

Les conditions données dans la clause **HAVING** ne doivent pas nécessairement comporter une fonction d'agrégation. Les deux requêtes suivantes donneront par exemples des résultats équivalents :

Code : SQL

```sql
SELECT nom_courant, COUNT(*) as nombre
FROM Animal
INNER JOIN Espece ON Espece.id = Animal.espece_id
GROUP BY nom_courant
HAVING nombre > 6 AND SUBSTRING(nom_courant, 1, 1) = 'C';

SELECT nom_courant, COUNT(*) as nombre
FROM Animal
INNER JOIN Espece ON Espece.id = Animal.espece_id
WHERE SUBSTRING(nom_courant, 1, 1) = 'C'
GROUP BY nom_courant
HAVING nombre > 6;
```

Il est cependant préférable, et de loin, d'utiliser la clause **WHERE** autant que possible, c'est-à-dire pour toutes les conditions, sauf celles utilisant une fonction d'agrégation. En effet, les conditions **HAVING** ne sont absolument pas optimisées, au contraire des conditions **WHERE**.
Voilà pour les fonctions d'agrégation et le regroupement. Étant donné la grande utilité de ces fonctions, je vous propose de faire quelques exercices pratiques pour clore cette partie.

Exercices sur les agrégats

Jusqu'à maintenant, ce fut très théorique. Or, la meilleur façon d'apprendre, c'est la pratique. Voici donc quelques exercices que je vous conseille de faire.
S'il vaut mieux que vous essayiez par vous-même avant de regarder la solution, ne restez cependant pas bloqué trop longtemps sur un exercice. Mais prenez toujours le temps de bien comprendre la solution.

Du simple...

1. Combien de races avons-nous dans la table Race ?

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT COUNT(*)
FROM Race;
```

Simple échauffement.

2. De combien de chiens connaissons-nous le père ?

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT COUNT(pere_id)
FROM Animal
INNER JOIN Especе ON Espece.id = Animal.espece_id
WHERE Espece.nom_courant = 'Chien';
```

L'astuce ici était de ne pas oublier de donner la colonne *pere_id* en paramètre à `COUNT()`, pour ne compter que les lignes ou *pere_id* est non `NULL`. Si vous n'avez pas utilisé de jointure mais avez fait directement *espece_id = 1*, je ne suis pas fâchée.

3. Quelle est la date de naissance de notre plus jeune femelle ?

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT MAX(date_naissance)
FROM Animal
WHERE sexe = 'femelle';
```

4. En moyenne, combien de pattes ont nos animaux nés avant le premier janvier 2010 ?

Secret (cliquez pour afficher)

Code : SQL

```
SELECT AVG(pattes)
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Espece.id
WHERE date_naissance < '2010-01-01';
```

5. Combien avons-nous de perroquets mâles et femelles, et quels sont leur nom (en une seule requête bien sûr) ?

Secret (cliquez pour afficher)

Code : SQL

```
SELECT sexe, COUNT(*), GROUP_CONCAT(nom SEPARATOR ', ')
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Espece.id
WHERE nom_courant = 'Perroquet amazone'
GROUP BY sexe;
```

Il suffisait de se rappeler de la méthode `GROUP_CONCAT()` pour pouvoir réaliser simplement cette requête. Peut-être avez-vous groupé sur l'espèce aussi (avec *nom_courant* ou autre). Ce n'était pas nécessaire puisqu'on avait restreint à une seule espèce avec la clause **WHERE**. Cependant, cela n'influe pas sur le résultat (mais bien la rapidité de la requête).

...Vers le complexe

1. Quelles sont les races dont nous ne possédons aucun individu ?

Secret (cliquez pour afficher)

Code : SQL

```
SELECT Race.nom, COUNT(race_id) AS nombre
FROM Race
LEFT JOIN Animal ON Animal.race_id = Race.id
GROUP BY Race.nom
HAVING nombre = 0;
```

Il fallait ici ne pas oublier la jointure externe (**LEFT** ou **RIGHT**, selon votre requête), ainsi que de mettre la colonne *race_id* (ou *Animal.id*, ou *Animal.espece_id* mais c'est moins intuitif) en paramètre de la fonction **COUNT()**.

2. Quelles sont les espèces (triées par ordre alphabétique du nom latin) dont nous possédons moins de cinq mâles ?

Secret (cliquez pour afficher)

Code : SQL

```
SELECT Espece.nom_latin, COUNT(espece_id) AS nombre
FROM Espece
```

```
LEFT JOIN Animal ON Animal.espece_id = Espece.id
WHERE sexe = 'male' OR Animal.id IS NULL
GROUP BY Espece.nom_latin
HAVING nombre < 5;
```

A nouveau, une jointure externe et *espece_id* en argument de **COUNT()**, mais il y avait ici une petite subtilité en plus. Puisqu'on demandait des informations sur les mâles uniquement, il fallait une condition **WHERE** `sexe = 'male'`. Mais cette condition fait que les lignes de la jointure provenant de la table *Espece* n'ayant aucune correspondance dans la table *Animal* sont éliminées également (puisque forcément, toutes les colonnes de la table *Animal*, dont *sexe*, seront à **NULL** pour ces lignes). Par conséquent, il fallait ajouter une condition permettant de garder ces fameuses lignes (les espèces pour lesquelles on n'a aucun individu, donc aucun mâle). Il fallait donc ajouter **OR** `Animal.id` **IS NULL**, ou faire cette condition sur toute autre colonne de *Animal* ayant la contrainte **NOT NULL**, et qui donc ne sera **NULL** que lors d'une jointure externe, en cas de non correspondance avec l'autre table.
Il n'y a plus alors qu'à ajouter la clause **HAVING** pour sélectionner les espèces ayant moins de cinq mâles.

3. Combien de mâles et de femelles avons-nous de chaque race, avec un compte total intermédiaire pour les races (mâles et femelles confondues) et pour les espèces ?

Secret (cliquez pour afficher)

Code : SQL

```
SELECT sexe, Race.nom, Espece.nom_courant, COUNT(*) AS nombre
FROM Animal
INNER JOIN Espece ON Animal.espece_id = Espece.id
RIGHT JOIN Race ON Espece.id = Race.espece_id
GROUP BY Espece.nom_courant, Race.nom, sexe WITH ROLLUP;
```

À nouveau, une jointure externe sur *Race* est nécessaire. Pour joindre *Espece* et *Animal*, une jointure interne est suffisante puisque l'espèce est définie pour chaque animal. Il suffit alors de ne pas oublier l'option **WITH ROLLUP** et de mettre les critères de regroupement dans le bon ordre pour avoir les super-agrégats voulus.

4. Combien de pattes, par espèce et au total, aurons-nous si nous prenons Parlotte, Spoutnik, Caribou, Cartouche, Cali, Canaille, Yoda, Zambo et Lulla ?

Secret (cliquez pour afficher)

Code : SQL

```
SELECT Espece.nom_courant, SUM(pattes) AS somme
FROM Animal
INNER JOIN Espece ON Espece.id = Animal.espece_id
WHERE Animal.nom IN ('Parlotte', 'Spoutnik', 'Caribou',
'Cartouche', 'Cali', 'Canaille', 'Yoda', 'Zambo', 'Lulla')
GROUP BY Espece.nom_courant WITH ROLLUP;
```

C'est ici la fonction **SUM()** qu'il fallait utiliser, puisqu'on veut le nombre total de pattes par groupe. Sans oublier le **WITH ROLLUP** pour avoir également le nombre total tous groupes confondus.

Vous devriez maintenant pouvoir vous débrouiller avec les fonctions d'agrégation, ainsi que les regroupements avec **GROUP BY**. Vous voici donc à la fin de la troisième partie. J'espère qu'elle vous aura plu. En tout cas, je suis sûre qu'elle vous sera utile !

Et voilà pour les nombres et les chaînes de caractères ! Notez que les fonctions d'agrégat sont parmi les plus utilisées donc soyez bien sûr d'avoir compris comment elles fonctionnaient, couplées à **GROUP BY** ou non.

Partie 4 : Fonctions : manipuler les dates

Cette partie sera entièrement consacrée aux données temporelles, et plus particulièrement à la manipulation de celles-ci à l'aide de fonctions. Il en existe beaucoup, et leur utilisation n'est pas bien compliquée. Et pourtant, c'est quelque chose qui semble rebuter les débutants, à tel point que beaucoup préfèrent représenter leurs données temporelles sous forme de chaînes de caractères, perdant ainsi l'opportunité d'utiliser les outils spécifiques existants.

C'est la raison pour laquelle j'ai décidé de consacrer toute une partie à l'utilisation de ces fonctions, même si 90% de ce que je vais écrire ici se trouve dans la documentation officielle.
J'espère qu'une présentation plus structurée, dotée de nombreux exemples et terminée par une série d'exercices rendra ces fonctions plus attractives aux nouveaux venus dans le monde de MySQL.

Obtenir la date/l'heure actuelle

Avant de plonger tête la première dans les fonctions temporelles, il convient de faire un bref rappel sur les différents types de données temporelles, qui sont au nombre de cinq : `DATE`, `TIME`, `DATETIME`, **`TIMESTAMP`** et **`YEAR`**.
Ensuite, nous verrons avec quelles fonctions il est possible d'obtenir la date actuelle, l'heure actuelle, ou les deux.

Rappels

Nous allons rapidement revoir les cinq types de données temporelles disponibles pour MySQL. Pour plus de détails, je vous renvoie au chapitre consacré à ceux-ci, ou à la documentation officielle.

Date

On peut manipuler une date (jour, mois, année) avec le type `DATE`. Ce type représente la date sous forme de chaîne de caractères `'AAAA-MM-JJ'` (A = année, M = mois, J = jour). Par exemple : le 21 octobre 2011 sera représenté `'2011-10-21'`.

Lorsque que l'on crée une donnée de type `DATE`, on peut le faire avec une multitude de formats différents, que MySQL convertira automatiquement. Il suffit de donner l'année (en deux ou quatre chiffres), suivie du mois (deux chiffres) puis du jour (deux chiffres). Avec une chaîne de caractères, n'importe quel caractère de ponctuation, ou aucun caractère, peut être utilisé pour séparer l'année du mois et le mois du jour. On peut aussi utiliser un nombre entier pour initialiser une date (pour autant qu'il ait du sens en tant que date bien sûr)

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

Heure

Pour une heure, ou une durée, on utilise le type `TIME`, qui utilise également une chaîne de caractères pour représenter l'heure : `'[H]HH:MM:SS'` (H = heures, M = minutes, S = secondes).

MySQL supporte des `TIME` allant de `'-838:59:59'` à `'838:59:59'`. Ce n'est en effet pas limité à 24h, puisqu'il est possible de stocker des durées.

Pour créer un `TIME`, on donne d'abord les heures, puis les minutes, puis les secondes, avec : entre chaque donnée. On peut éventuellement aussi spécifier un nombre de jours avant les heures (suivi cette fois d'un espace, et non d'un :) : `'J HH:MM:SS'`.

Date et heure

Sans surprise, `DATETIME` est le type de données représentant une date et une heure, toujours stockées sous forme de chaîne de caractères : `'AAAA-MM-JJ HH:MM:SS'`. Les heures doivent ici être comprises entre 00 et 23, puisqu'il ne peut plus s'agir d'une durée.

Comme pour `DATE`, l'important dans `DATETIME` est l'ordre des données : année, mois, jour, heures, minutes, secondes ; chacune avec deux chiffres, sauf l'année pour laquelle on peut aussi donner quatre chiffres. Cela peut être un nombre entier, ou une chaîne de caractères, auquel cas les signes de ponctuation entre chaque partie du `DATETIME` importent peu.

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

Timestamp

Le timestamp d'une date est le nombre de secondes écoulées depuis le 1er janvier 1970, 0h0min0s (TUC) et la date en question. Mais attention, ce qui est stocké par MySQL dans une donnée de type **TIMESTAMP** n'est pas ce nombre de seconde, mais bien la date, sous format numérique : AAAAMMJJHHMMSS (contrairement à DATE, TIME et DATETIME qui utilisent des chaînes de caractères).

Un timestamp est limité aux dates allant du premier janvier 1970 00h00min00s au 19 janvier 2038 03h14min07s.

Année

Le dernier type temporel est **YEAR**, qui stocke une année sous forme d'entier. Nous n'en parlerons pas beaucoup dans cette partie.

YEAR peut contenir des années comprises entre 1901 et 2155.

Date actuelle

Il existe deux fonctions permettant d'avoir la date actuelle (juste la date, sans l'heure, donc format DATE).

CURDATE()

Code : SQL

```sql
SELECT CURDATE();
```

Code : Console

```
+------------+
| CURDATE()  |
+------------+
| 2011-10-25 |
+------------+
```

CURRENT_DATE()

Cette fonction est un peu particulière. Elle peut en effet s'utiliser de manière tout à fait classique :

Code : SQL

```sql
SELECT CURRENT_DATE();
```

Code : Console

```
+-----------------+
| CURRENT_DATE()  |
+-----------------+
| 2011-10-25      |
+-----------------+
```

Mais elle peut aussi s'utiliser sans les parenthèses :

Code : SQL

```
SELECT CURRENT_DATE;
```

Code : Console

```
+---------------+
| CURRENT_DATE  |
+---------------+
| 2011-10-25    |
+---------------+
```

C'est le cas de plusieurs autres fonctions temporelles.

Heure actuelle

À nouveau, deux fonctions existent, extrêmement similaires aux fonctions permettant d'avoir la date actuelle. Il suffit en effet de remplacer `DATE` par `TIME` dans le nom de la fonction.

Code : SQL

```
SELECT CURTIME(), CURRENT_TIME(), CURRENT_TIME;
```

Code : Console

```
+-----------+----------------+--------------+
| CURTIME() | CURRENT_TIME() | CURRENT_TIME |
+-----------+----------------+--------------+
| 18:04:20  | 18:04:20       | 18:04:20     |
+-----------+----------------+--------------+
```

Date et heure actuelles

Les fonctions

Pour obtenir la date et l'heure actuelles (format `DATETIME`), c'est Byzance : vous avez le choix entre cinq fonctions différentes !

NOW() et SYSDATE()

`NOW()` est sans doute la fonction MySQL la plus utilisée pour obtenir la date du jour. C'est aussi la plus facile à retenir (bien que les noms des fonctions soient souvent explicites en SQL), puisque "now" veut dire "maintenant" en anglais. `SYSDATE()` ("system date") est aussi pas mal utilisée.

Code : SQL

```
SELECT NOW(), SYSDATE();
```

Code : Console

```
+---------------------+---------------------+
| NOW()               | SYSDATE()           |
```

```
+---------------------+---------------------+
| 2011-10-26 09:40:18 | 2011-10-26 09:40:18 |
+---------------------+---------------------+
```

Et les autres

Les trois autres fonctions peuvent s'utiliser avec ou sans parenthèses.

- **CURRENT_TIMESTAMP** ou **CURRENT_TIMESTAMP()**
- **LOCALTIME** ou **LOCALTIME()**
- **LOCALTIMESTAMP** ou **LOCALTIMESTAMP()**

Code : SQL

```sql
SELECT LOCALTIME, CURRENT_TIMESTAMP(), LOCALTIMESTAMP;
```

Code : Console

```
+---------------------+---------------------+---------------------+
| LOCALTIME           | CURRENT_TIMESTAMP() | LOCALTIMESTAMP      |
+---------------------+---------------------+---------------------+
| 2011-10-26 10:02:31 | 2011-10-26 10:02:31 | 2011-10-26 10:02:31 |
+---------------------+---------------------+---------------------+
```

Qui peut le plus, peut le moins

Il est tout à fait possible d'utiliser une des fonctions donnant l'heure et la date pour remplir une colonne de type `DATE`, ou de type `TIME`. MySQL convertira simplement le `DATETIME` en `DATE`, ou en `TIME`, en supprimant la partie inutile.

Exemple

Créons une table de test simple, avec juste 3 colonnes. Une de type `DATE`, une de type `TIME`, une de type `DATETIME`. On peut voir que l'insertion d'une ligne en utilisant `NOW()` pour les trois colonnes donne le résultat attendu.

Code : SQL

```sql
-- Création d'une table de test toute simple
CREATE TABLE testDate (
    dateActu DATE,
    timeActu TIME,
    datetimeActu DATETIME
);

INSERT INTO testDate VALUES (NOW(), NOW(), NOW());

SELECT *
FROM testDate;
```

Code : Console

```
+------------+----------+---------------------+
| dateActu   | timeActu | datetimeActu        |
+------------+----------+---------------------+
| 2011-10-26 | 11:22:10 | 2011-10-26 11:22:10 |
+------------+----------+---------------------+
```

Timestamp unix

Il existe encore une fonction qui peut donner des informations sur la date et l'heure actuelle, sous forme de timestamp unix. Ce qui est donc le nombre de secondes écoulées depuis le premier janvier 1970, à 00:00:00. Il s'agit de `UNIX_TIMESTAMP()`.

Je vous la donne au cas où, mais j'espère que vous ne vous en servirez pas pour stocker vos dates sous forme d'`INT` avec un timestamp unix !

Code : SQL

```sql
SELECT UNIX_TIMESTAMP();
```

Code : Console

```
+------------------+
| UNIX_TIMESTAMP() |
+------------------+
|       1319621754 |
+------------------+
```

Rien de bien compliqué ni d'extraordinaire dans ce chapitre. Le meilleur reste à venir !

Formater une donnée temporelle

Lorsque l'on tombe sur quelqu'un qui a fait le (mauvais) choix de stocker ses dates sous forme de chaînes de caractères, et qu'on lui demande les raisons de son choix, celle qui revient le plus souvent est qu'il ne veut pas afficher ses dates sous la forme `'AAAA-MM-JJ'`. Donc il les stocke sous forme de `CHAR` ou `VARCHAR` `'JJ/MM/AAAA'` par exemple, ou n'importe quel format de son choix.
Malheureusement, en faisant ça, il se prive des nombreux avantages des formats temporels SQL (en particulier toutes les fonctions que nous avons vues, voyons, verrons dans cette partie), et cela pour rien, car SQL dispose de puissantes fonctions permettant de formater une donnée temporelle.
C'est donc ce que nous allons voir dans ce chapitre.

Extraire une information précise

Commençons en douceur avec des fonctions permettant d'extraire une information d'une donnée temporelle. Par exemple, le jour de la semaine, le nom du mois, l'année, etc.

Informations sur la date

*Extraire la partie **DATE***

La fonction `DATE(datetime)` permet d'extraire la partie `DATE` d'une donnée de type `DATETIME` (ou `DATE` mais c'est moins utile...).

Code : SQL

```sql
SELECT nom, date_naissance,
       DATE(date_naissance) AS uniquementDate
FROM Animal
WHERE espece_id = 4;
```

Code : Console

```
+----------+---------------------+----------------+
| nom      | date_naissance      | uniquementDate |
+----------+---------------------+----------------+
| Safran   | 2007-03-04 19:36:00 | 2007-03-04     |
| Gingko   | 2008-02-20 02:50:00 | 2008-02-20     |
| Bavard   | 2009-03-26 08:28:00 | 2009-03-26     |
| Parlotte | 2009-03-26 07:55:00 | 2009-03-26     |
+----------+---------------------+----------------+
```

Le jour

Les fonctions suivantes donnent des informations sur le jour :

- **`DAY`**`(date)` ou `DAYOFMONTH(date)` : donne le jour du mois (sous forme de nombre entier de 1 à 31) ;
- `DAYOFWEEK(date)` : donne l'index du jour de la semaine (nombre de 1 à 7 avec 1 = dimanche, 2 = lundi, ... 7 = samedi) ;
- `WEEKDAY(date)` : donne aussi l'index du jour de la semaine, de manière un peu différente (nombre de 0 à 6 avec 0 = lundi, 1 = mardi, ... 6 = dimanche) ;
- `DAYNAME(date)` : donne le nom du jour de la semaine ;
- `DAYOFYEAR(date)` : retourne le numéro du jour par rapport à l'année (de 1 à 366 donc).

Exemples :

Code : SQL

```sql
SELECT nom, date_naissance,
        DAY(date_naissance) AS jour,
        DAYOFMONTH(date_naissance) AS jour,
        DAYOFWEEK(date_naissance) AS jour_semaine_nb,
        WEEKDAY(date_naissance) AS jour_semaine_nb2,
        DAYNAME(date_naissance) AS jour_semaine,
        DAYOFYEAR(date_naissance) AS jour_annee
FROM Animal
WHERE espece_id = 4;
```

Code : Console

```
+----------+---------------------+------+------+-----------------+-----------------
| nom      | date_naissance      | jour | jour | jour_semaine_nb | jour_semaine_nb2
+----------+---------------------+------+------+-----------------+-----------------
| Safran   | 2007-03-04 19:36:00 |    4 |    4 |               1 |                6
| Gingko   | 2008-02-20 02:50:00 |   20 |   20 |               4 |                2
| Bavard   | 2009-03-26 08:28:00 |   26 |   26 |               5 |                3
| Parlotte | 2009-03-26 07:55:00 |   26 |   26 |               5 |                3
+----------+---------------------+------+------+-----------------+-----------------
```

Tout ça fonctionne très bien, mais ce serait encore mieux si l'on pouvait avoir le nom des jours en français plutôt qu'en anglais. Aucun problème, il suffit de le demander, en exécutant la requête suivante :

Code : SQL

```sql
SET lc_time_names = 'fr_FR';
```

Et voilà le travail :

Code : SQL

```sql
SELECT nom, date_naissance,
        DAYNAME(date_naissance) AS jour_semaine
FROM Animal
WHERE espece_id = 4;
```

Code : Console

```
+----------+---------------------+--------------+
| nom      | date_naissance      | jour_semaine |
+----------+---------------------+--------------+
| Safran   | 2007-03-04 19:36:00 | dimanche     |
| Gingko   | 2008-02-20 02:50:00 | mercredi     |
| Bavard   | 2009-03-26 08:28:00 | jeudi        |
| Parlotte | 2009-03-26 07:55:00 | jeudi        |
+----------+---------------------+--------------+
```

La semaine

La seule information qu'on puisse obtenir sur la semaine d'une date, c'est le numéro de la semaine par rapport à l'année. Et celui-ci

peut être obtenu grâce à trois fonctions : `WEEK(date)`, `WEEKOFYEAR(date)` et `YEARWEEK(date)`.

- `WEEK(date)` : donne uniquement le numéro de la semaine (un nombre entre 0 et 53, puisque 7x52 = 364, donc en un an, il y a 52 semaine et 1 ou 2 jour d'une 53e semaine).
- `WEEKOFYEAR(date)` : donne uniquement le numéro de la semaine (un nombre entre 1 et 53).
- `YEARWEEK(date)` : donne également l'année.

Code : SQL

```
SELECT nom, date_naissance, WEEK(date_naissance),
WEEKOFYEAR(date_naissance), YEARWEEK(date_naissance)
FROM Animal
WHERE espece_id = 4;
```

Code : Console

```
+----------+---------------------+----------------------+--------------------------
| nom      | date_naissance      | WEEK(date_naissance) | WEEKOFYEAR(date_naissance
+----------+---------------------+----------------------+--------------------------
| Safran   | 2007-03-04 19:36:00 |                    9 |
| Gingko   | 2008-02-20 02:50:00 |                    7 |
| Bavard   | 2009-03-26 08:28:00 |                   12 |                        1
| Parlotte | 2009-03-26 07:55:00 |                   12 |                        1
+----------+---------------------+----------------------+--------------------------
```

`WEEK()` et `YEARWEEK()` peuvent également accepter un deuxième argument, qui sert notamment à spécifier si la semaine doit commencer le lundi ou le dimanche, et ce qu'on considère comme la première semaine de l'année. Selon l'option utilisée par `WEEK()`, le résultat de cette fonction peut donc différer de celui de `WEEKOFYEAR()`. Si ces options vous intéressent, je vous invite à aller vous renseigner dans la documentation officielle.

Le mois

Pour le mois, il existe deux fonctions : **`MONTH`**`(date)` qui donne le numéro du mois (nombre de 1 à 12) et `MONTHNAME(date)` qui donne le nom du mois.

Code : SQL

```
SELECT nom, date_naissance, MONTH(date_naissance),
MONTHNAME(date_naissance)
FROM Animal
WHERE espece_id = 4;
```

Code : Console

```
+----------+---------------------+----------------------+--------------------------
| nom      | date_naissance      | MONTH(date_naissance) | MONTHNAME(date_naissance
+----------+---------------------+----------------------+--------------------------
| Safran   | 2007-03-04 19:36:00 |                    3 | mars
| Gingko   | 2008-02-20 02:50:00 |                    2 | février
| Bavard   | 2009-03-26 08:28:00 |                    3 | mars
| Parlotte | 2009-03-26 07:55:00 |                    3 | mars
+----------+---------------------+----------------------+--------------------------
```

L'année

Enfin, la fonction `YEAR(date)` extrait l'année.

Code : SQL

```
SELECT nom, date_naissance, YEAR(date_naissance)
FROM Animal
WHERE espece_id = 4;
```

Code : Console

```
+----------+---------------------+-----------------------+
| nom      | date_naissance      | YEAR(date_naissance)  |
+----------+---------------------+-----------------------+
| Safran   | 2007-03-04 19:36:00 |                  2007 |
| Gingko   | 2008-02-20 02:50:00 |                  2008 |
| Bavard   | 2009-03-26 08:28:00 |                  2009 |
| Parlotte | 2009-03-26 07:55:00 |                  2009 |
+----------+---------------------+-----------------------+
```

Informations sur l'heure

En ce qui concerne l'heure, voici quatre fonctions intéressantes (et faciles à retenir) :

- `TIME(datetime)` : qui extrait l'heure complète (le `TIME`) d'une donnée de type `DATETIME` ou `TIME` ;
- `HOUR(heure)` : qui extrait l'heure ;
- `MINUTE(heure)` : qui extrait les minutes ;
- `SECOND(heure)` : qui extrait les secondes.

Code : SQL

```
SELECT nom, date_naissance, TIME(date_naissance),
HOUR(date_naissance), MINUTE(date_naissance), SECOND(date_naissance)
FROM Animal
WHERE espece_id = 4;
```

Code : Console

```
+----------+---------------------+----------------------+----------------------+---
| nom      | date_naissance      | TIME(date_naissance) | HOUR(date_naissance) | MI
+----------+---------------------+----------------------+----------------------+---
| Safran   | 2007-03-04 19:36:00 | 19:36:00             |                   19 |
| Gingko   | 2008-02-20 02:50:00 | 02:50:00             |                    2 |
| Bavard   | 2009-03-26 08:28:00 | 08:28:00             |                    8 |
| Parlotte | 2009-03-26 07:55:00 | 07:55:00             |                    7 |
+----------+---------------------+----------------------+----------------------+---
```

Formater une date facilement

Avec les fonctions que nous venons de voir, vous êtes maintenant capable d'afficher une date dans un joli format, par exemple "le lundi 8 novembre 1987".

Code : SQL

```sql
SELECT nom, date_naissance, CONCAT_WS(' ', 'le',
DAYNAME(date_naissance), DAY(date_naissance),
MONTHNAME(date_naissance), YEAR(date_naissance)) AS jolie_date
FROM Animal
WHERE espece_id = 4;
```

Code : Console

```
+----------+---------------------+------------------------------+
| nom      | date_naissance      | jolie_date                   |
+----------+---------------------+------------------------------+
| Safran   | 2007-03-04 19:36:00 | le dimanche 4 mars 2007      |
| Gingko   | 2008-02-20 02:50:00 | le mercredi 20 février 2008  |
| Bavard   | 2009-03-26 08:28:00 | le jeudi 26 mars 2009        |
| Parlotte | 2009-03-26 07:55:00 | le jeudi 26 mars 2009        |
+----------+---------------------+------------------------------+
```

Cependant, il faut bien avouer que c'est un peu long à écrire. Heureusement, il existe une fonction qui va nous permettre de faire la même chose, en bien plus court : `DATE_FORMAT(date, format)`.

Cette fonction `DATE_FORMAT()` a donc deux paramètres :

- date : la date à formater (`DATE`, `TIME` ou `DATETIME`) ;
- format : le format voulu.

Format

Le format à utiliser doit être donné sous forme de chaîne de caractères. Cette chaîne peut contenir un ou plusieurs spécificateurs dont les plus courants sont listés dans le tableau ci-dessous.

Spécificateur	Description
%d	Jour du mois (nombre à deux chiffres, de 00 à 31)
e%	Jour du mois (nombre à un ou deux chiffres, de 0 à 31)
%D	Jour du mois, avec suffixe (1rst, 2nd, ..., 31th) **en anglais**
%w	Numéro du jour de la semaine (dimanche = 0, ..., samedi = 6)
%W	Nom du jour de la semaine
%a	Nom du jour de la semaine en abrégé
%m	Mois (nombre de deux chiffres, de 00 à 12)
%c	Mois (nombre de un ou deux chiffres, de 0 à 12)
%M	Nom du mois
%b	Nom du mois en abrégé

%y	Année, sur deux chiffres
%Y	Année, sur quatre chiffres
%r	Heure complète, format 12h (hh:mm:ss AM/PM)
%T	Heure complète, format 24h (hh:mm:ss)
%h	Heure sur deux chiffres et sur 12 heures (de 00 à 12)
%H	Heure sur deux chiffres et sur 24 heures (de 00 à 23)
%l	Heure sur un ou deux chiffres et sur 12 heures (de 0 à 12)
%k	Heure sur un ou deux chiffres et sur 24 heures (de 0 à 23)
%i	Minutes (de 00 à 59)
%s ou %S	Secondes (de 00 à 59)
%p	AM/PM

Tous les caractères ne faisant pas partie d'un spécificateur sont simplement recopiés tels quels.

Exemples

Même résultat que précédemment...

...Avec une requête bien plus courte :

Code : SQL

```sql
SELECT nom, date_naissance, DATE_FORMAT(date_naissance, 'le %W %e %M %Y') AS jolie_date
FROM Animal
WHERE espece_id = 4;
```

Code : Console

```
+----------+---------------------+-----------------------------+
| nom      | date_naissance      | jolie_date                  |
+----------+---------------------+-----------------------------+
| Safran   | 2007-03-04 19:36:00 | le dimanche 4 mars 2007     |
| Gingko   | 2008-02-20 02:50:00 | le mercredi 20 février 2008 |
| Bavard   | 2009-03-26 08:28:00 | le jeudi 26 mars 2009       |
| Parlotte | 2009-03-26 07:55:00 | le jeudi 26 mars 2009       |
+----------+---------------------+-----------------------------+
```

Autres exemples

Attention à bien échapper les guillemets éventuels dans la chaîne de caractères du format.

Code : SQL

```
SELECT DATE_FORMAT(NOW(), 'Nous sommes aujourd''hui le %d %M de
l''année %Y. Il est actuellement %l heures et %i minutes.') AS
Top_date_longue;

SELECT DATE_FORMAT(NOW(), '%d %b. %y - %r') AS Top_date_courte;
```

Code : Console

```
+-------------------------------------------------------------------------------
| Top_date_longue
+-------------------------------------------------------------------------------
| Nous sommes aujourd'hui le 27 octobre de l'année 2011. Il est actuellement 3 heur
+-------------------------------------------------------------------------------

+--------------------------+
| Top_date_courte          |
+--------------------------+
| 27 oct. 11 - 03:34:40 PM |
+--------------------------+
```

Fonction supplémentaire pour l'heure

`DATE_FORMAT()` peut s'utiliser sur des données de type `DATE`, `TIME` ou `DATETIME`. Mais il existe également une fonction `TIME_FORMAT(heure, format)`, qui elle ne sert qu'à formater les heures (et ne peut donc pas s'utiliser sur une `DATE`). Elle s'utilise exactement de la même manière, simplement il faut y utiliser des spécificateurs ayant du sens pour une donnée `TIME`, sinon **`NULL`** ou `0` est renvoyé.

Exemples

Code : SQL

```
-- Sur une DATETIME
SELECT TIME_FORMAT(NOW(), '%r') AS jolie_heure;

-- Sur un TIME
SELECT TIME_FORMAT(CURTIME(), '%r') AS jolie_heure;

-- Sur une DATETIME mais avec mauvais spécificateur
SELECT TIME_FORMAT(NOW(), '%M %r') AS jolie_heure;

-- Sur une DATE
SELECT TIME_FORMAT(CURDATE(), '%M %r') AS jolie_heure;
```

Code : Console

```
+-------------+
| jolie_heure |
+-------------+
| 04:39:37 PM |
+-------------+

+-------------+
| jolie_heure |
+-------------+
| 04:39:37 PM |
```

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

+-------------+
| jolie_heure |
+-------------+
| NULL        |
+-------------+

+-------------+
| jolie_heure |
+-------------+
| NULL        |
+-------------+
```

Formats standards

Il existe un certain nombre de formats de date et d'heure standards, prédéfinis, que l'on peut utiliser dans la fonction `DATE_FORMAT()`. Pour obtenir ces formats, il faut appeler la fonction `GET_FORMAT(type, standard)`.

Le paramètre type doit être choisi entre les trois types de données `DATE`, `TIME` et `DATETIME`.

Et il existe cinq formats standards :

- `'EUR'`
- `'USA'`
- `'ISO'`
- `'JIS'`
- `'INTERNAL'`

Et voici un tableau reprenant les différentes possibilités :

Fonction	Format	Exemple
`GET_FORMAT(DATE,'USA')`	'%m.%d.%Y'	10.30.1988
`GET_FORMAT(DATE,'JIS')`	'%Y-%m-%d'	1988-10-30
`GET_FORMAT(DATE,'ISO')`	'%Y-%m-%d'	1988-10-30
`GET_FORMAT(DATE,'EUR')`	'%d.%m.%Y'	30.10.1988
`GET_FORMAT(DATE,'INTERNAL')`	'%Y%m%d'	19881031
`GET_FORMAT(DATETIME,'USA')`	'%Y-%m-%d-%H.%i.%s'	1988-10-30-12.44.33
`GET_FORMAT(DATETIME,'JIS')`	'%Y-%m-%d %H:%i:%s'	1988-10-30 12:44:33
`GET_FORMAT(DATETIME,'ISO')`	'%Y-%m-%d %H:%i:%s'	1988-10-30 12:44:33
`GET_FORMAT(DATETIME,'EUR')`	'%Y-%m-%d-%H.%i.%s'	1988-10-30-12.44.33
`GET_FORMAT(DATETIME,'INTERNAL')`	'%Y%m%d%H%i%s'	19881030124433
`GET_FORMAT(TIME,'USA')`	'%h:%i:%s %p'	12:44:33 PM
`GET_FORMAT(TIME,'JIS')`	'%H:%i:%s'	12:44:33
`GET_FORMAT(TIME,'ISO')`	'%H:%i:%s'	12:44:33
`GET_FORMAT(TIME,'EUR')`	'%H.%i.%S'	12.44.33
`GET_FORMAT(TIME,'INTERNAL')`	'%H%i%s'	124433

Exemples

Code : SQL

```sql
SELECT DATE_FORMAT(NOW(), GET_FORMAT(DATE, 'EUR')) AS date_eur;

SELECT DATE_FORMAT(NOW(), GET_FORMAT(TIME, 'JIS')) AS heure_jis;

SELECT DATE_FORMAT(NOW(), GET_FORMAT(DATETIME, 'USA')) AS
date_heure_usa;
```

Code : Console

```
+------------+
| date_eur   |
+------------+
| 27.10.2011 |
+------------+

+-----------+
| heure_jis |
+-----------+
| 17:37:28  |
+-----------+

+---------------------+
| date_heure_usa      |
+---------------------+
| 2011-10-27 17.37.29 |
+---------------------+
```

Créer une date à partir d'une chaîne de caractères

Voici une dernière fonction ayant trait au format des dates : `STR_TO_DATE(date, format)`. Cette fonction est l'exact contraire de `DATE_FORMAT()` : elle prend une chaîne de caractères représentant une date suivant le format donné, et renvoie la `DATETIME` correspondante.

Exemples

Code : SQL

```sql
SELECT STR_TO_DATE('03/04/2011 à 09h17', '%d/%m/%Y à %Hh%i') AS
StrDate;

SELECT STR_TO_DATE('15blabla', '%Hblabla') StrTime;
```

Code : Console

```
+---------------------+
| StrDate             |
+---------------------+
| 2011-04-03 09:17:00 |
+---------------------+

+----------+
| StrTime  |
+----------+
| 15:00:00 |
+----------+
```

Il est bien sûr possible d'utiliser `GET_FORMAT()` aussi avec `STR_TO_DATE()`.

Code : SQL

```
SELECT STR_TO_DATE('11.21.2011', GET_FORMAT(DATE, 'USA')) AS
date_usa;

SELECT STR_TO_DATE('12.34.45', GET_FORMAT(TIME, 'EUR')) AS
heure_eur;

SELECT STR_TO_DATE('20111027133056', GET_FORMAT(TIMESTAMP,
'INTERNAL')) AS date_heure_int;
```

Code : Console

```
+------------+
| date_usa   |
+------------+
| 2011-11-21 |
+------------+

+-----------+
| heure_eur |
+-----------+
| 12:34:45  |
+-----------+

+---------------------+
| date_heure_int      |
+---------------------+
| 2011-10-27 13:30:56 |
+---------------------+
```

J'espère que vous êtes dorénavant convaincus que l'argument "je n'aime pas le format `'AAAA-MM-JJ'` des dates SQL" est un très mauvais argument !

Calculs sur les données temporelles

Il est fréquent de vouloir faire des calculs sur des données temporelles. Par exemple, pour calculer le nombre de jours ou d'heures entre deux dates, pour ajouter une certaine durée à une donnée en cas de calcul d'échéance, etc.
Pour ce faire, on peut soit se lancer dans des calculs compliqués en convertissant des jours en heures, des heures en jours, des minutes en secondes, etc ; soit utiliser les fonctions SQL existantes, faites pour ça. Je vous laisse deviner quelle solution est la meilleure...

Nous allons donc voir dans ce chapitre comment :

- calculer la différence entre deux données temporelles ;
- ajouter un intervalle de temps à une donnée temporelle ;
- convertir une donnée horaire en un nombre de secondes ;
- et d'autre petites choses bien utiles.

Différence entre deux dates/heures

Trois fonctions permettent de calculer le temps écoulé entre deux données temporelles :

- `DATEDIFF()` : qui donne un résultat en nombre de jours ;
- `TIMEDIFF()` : qui donne un résultat sous forme de TIME ;
- `TIMESTAMPDIFF()` : qui donne le résultat dans l'unité de temps souhaitée (heure, secondes, mois, ...).

DATEDIFF()

`DATEDIFF(date1, date2)` peut s'utiliser avec des données de type `DATE` ou `DATETIME` (dans ce dernier cas, seule la partie date est utilisée).

Les trois requêtes suivantes donnent donc le même résultat.

Code : SQL

```
SELECT DATEDIFF('2011-12-25','2011-11-10') AS nb_jours;
SELECT DATEDIFF('2011-12-25 22:12:18','2011-11-10 12:15:41') AS
nb_jours;
SELECT DATEDIFF('2011-12-25 22:12:18','2011-11-10') AS nb_jours;
```

Code : Console

```
+----------+
| nb_jours |
+----------+
|       45 |
+----------+
```

TIMEDIFF()

La fonction `TIMEDIFF(expr1, expr2)` calcule la durée entre *expr1* et *expr2*. Les deux arguments doivent être de même type, soit `TIME`, soit `DATETIME`.

Code : SQL

```
-- Avec des DATETIME
```

```
SELECT '2011-10-08 12:35:45' AS datetime1, '2011-10-07 16:00:25' AS
datetime2, TIMEDIFF('2011-10-08 12:35:45', '2011-10-07 16:00:25') as
difference;

-- Avec des TIME
SELECT '12:35:45' AS time1, '00:00:25' AS time2,
TIMEDIFF('12:35:45', '00:00:25') as difference;
```

Code : Console

```
+---------------------+---------------------+------------+
| datetime1           | datetime2           | difference |
+---------------------+---------------------+------------+
| 2011-10-08 12:35:45 | 2011-10-07 16:00:25 | 20:35:20   |
+---------------------+---------------------+------------+

+----------+----------+------------+
| time1    | time2    | difference |
+----------+----------+------------+
| 12:35:45 | 00:00:25 | 12:35:20   |
+----------+----------+------------+
```

TIMESTAMPDIFF()

La fonction `TIMESTAMPDIFF()` prend quant à elle un paramètre supplémentaire : l'unité de temps désirée pour le résultat. Les unités autorisées comprennent : **`SECOND`** (secondes), **`MINUTE`** (minutes), `HOUR` (heures), **`DAY`** (jours), `WEEK` (semaines), **`MONTH`** (mois), `QUARTER` (trimestres) et **`YEAR`** (années).

`TIMESTAMPDIFF(unite, date1, date2)` s'utilise également avec des données de type `DATE` ou `DATETIME`. Si vous demandez un résultat comprenant une unité inférieure au jour (heure ou minute par exemple) et que l'une de vos données est de type `DATE`, MySQL complètera cette date par l'heure par défaut `'00:00:00'`.

Code : SQL

```
SELECT TIMESTAMPDIFF(DAY, '2011-11-10', '2011-12-25') AS nb_jours,
       TIMESTAMPDIFF(HOUR,'2011-11-10', '2011-12-25 22:00:00') AS
nb_heures_def,
       TIMESTAMPDIFF(HOUR,'2011-11-10 14:00:00', '2011-12-25
22:00:00') AS nb_heures,
       TIMESTAMPDIFF(QUARTER,'2011-11-10 14:00:00', '2012-08-25
22:00:00') AS nb_trimestres;
```

Code : Console

```
+----------+---------------+-----------+---------------+
| nb_jours | nb_heures_def | nb_heures | nb_trimestres |
+----------+---------------+-----------+---------------+
|       45 |          1102 |      1088 |             3 |
+----------+---------------+-----------+---------------+
```

Ajout et retrait d'un intervalle de temps

Intervalle

Certaines des fonctions et opérations suivantes utilisent le mot-clé `INTERVAL`, permettant de définir un intervalle de temps à

ajouter ou soustraire d'une date.

Un intervalle de temps est défini par une quantité et une unité ("3 jours" par exemple, "3" étant la quantité, "jour" l'unité). En MySQL, il existe une vingtaine d'unités possibles pour un `INTERVAL`, dont une partie est listée dans le tableau ci-dessous.

Unité	Format
SECOND	-
MINUTE	-
HOUR	-
DAY	-
WEEK	-
MONTH	-
YEAR	-
MINUTE_SECOND	'm:S'
HOUR_SECOND	'HH:mm:SS'
HOUR_MINUTE	'HH:mm'
DAY_SECOND	'J HH:mm:SS'
DAY_MINUTE	'J HH:mm'
DAY_HOUR	'J HH'
YEAR_MONTH	'A-M'

Notez que, comme pour `DATE`, `DATETIME` et `TIME`, les signes de ponctuation séparant les différentes parties d'un intervalle ne doivent pas nécessairement être `'-'` pour la partie date et `':'` pour la partie heure. Il ne s'agit que de suggestions. N'importe quel signe de ponctuation (ou aucun) sera accepté.

Ajout d'un intervalle de temps

Trois fonctions permettent d'ajouter un intervalle de temps à une date (de type `DATE` ou `DATETIME`) :

- `ADDDATE()` : qui s'utilise avec un `INTERVAL` ou un nombre de jours.
- `DATE_ADD()` : qui s'utilise avec un `INTERVAL`.
- `TIMESTAMPADD()` : qui n'utilise pas d'`INTERVAL` mais un nombre plus limité d'unités de temps.

ADDDATE()

Cette fonction peut s'utiliser de deux manières : soit en précisant un intervalle de temps avec le mot-clé `INTERVAL`, soit en donnant un nombre de jours à ajouter à la date.

- `ADDDATE(date, INTERVAL quantite unite)`
- `ADDDATE(date, nombreJours)`

Exemples

Code : SQL

```sql
-- Avec DATE et INTERVAL
SELECT ADDDATE('2011-05-21', INTERVAL 3 MONTH);

-- Avec DATETIME et INTERVAL
SELECT ADDDATE('2011-05-21 12:15:56', INTERVAL '3 02:10:32'
DAY_SECOND);

-- Avec DATE et nombre de jours
SELECT ADDDATE('2011-05-21', 12);

-- Avec DATETIME et nombre de jours
SELECT ADDDATE('2011-05-21 12:15:56', 42);
```

Code : Console

```
+-----------------------------------------+
| ADDDATE('2011-05-21', INTERVAL 3 MONTH) |
+-----------------------------------------+
| 2011-08-21                              |
+-----------------------------------------+

+---------------------------------------------------------------------+
| ADDDATE('2011-05-21 12:15:56', INTERVAL '3 02:10:32' DAY_SECOND) |
+---------------------------------------------------------------------+
| 2011-05-24 14:26:28                                                 |
+---------------------------------------------------------------------+

+---------------------------+
| ADDDATE('2011-05-21', 12) |
+---------------------------+
| 2011-06-02                |
+---------------------------+

+------------------------------------+
| ADDDATE('2011-05-21 12:15:56', 42) |
+------------------------------------+
| 2011-07-02 12:15:56                |
+------------------------------------+
```

DATE_ADD()

`DATE_ADD(date, INTERVAL quantite unite)` s'utilise exactement de la même manière que `ADDDATE(date, INTERVAL quantite unite)`.

Exemples

Code : SQL

```sql
-- Avec DATE
SELECT DATE_ADD('2011-05-21', INTERVAL 3 MONTH);

-- Avec DATETIME
SELECT DATE_ADD('2011-05-21 12:15:56', INTERVAL '3 02:10:32'
DAY_SECOND);
```

Code : Console

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

```
| DATE_ADD('2011-05-21', INTERVAL 3 MONTH) |
+------------------------------------------+
| 2011-08-21                               |
+------------------------------------------+

+---------------------------------------------------------------------+
| DATE_ADD('2011-05-21 12:15:56', INTERVAL '3 02:10:32' DAY_SECOND) |
+---------------------------------------------------------------------+
| 2011-05-24 14:26:28                                                 |
+---------------------------------------------------------------------+
```

Opérateur +

Il est également possible d'ajouter un intervalle de temps à une date en utilisant simplement l'opérateur + et un INTERVAL. L'intervalle peut se trouver à droite ou à gauche du signe +.

Exemples

Code : SQL

```sql
-- Avec DATE et intervalle à droite
SELECT '2011-05-21' + INTERVAL 5 DAY;

-- Avec DATETIME et intervalle à gauche
SELECT INTERVAL '3 12' DAY_HOUR + '2011-05-21 12:15:56';
```

Code : Console

```
+--------------------------------+
| '2011-05-21' + INTERVAL 5 DAY |
+--------------------------------+
| 2011-05-26                     |
+--------------------------------+

+-------------------------------------------------+
| INTERVAL '3 12' DAY_HOUR + '2011-05-21 12:15:56' |
+-------------------------------------------------+
| 2011-05-25 00:15:56                             |
+-------------------------------------------------+
```

TIMESTAMPADD()

TIMESTAMPADD(unite, quantite, date) est un peu plus restreint que DATE_ADD() et ADDDATE(). En effet, cette fonction n'utilise pas d'INTERVAL. Il faut cependant définir une unité parmi les suivantes : FRAC_SECOND, **SECOND**, **MINUTE**, HOUR, **DAY**, WEEK, **MONTH**, QUARTER, et **YEAR**.

Exemple

Code : SQL

```sql
-- Avec DATE
SELECT TIMESTAMPADD(DAY, 5, '2011-05-21');

-- Avec DATETIME
SELECT TIMESTAMPADD(MINUTE, 34, '2011-05-21 12:15:56');
```

Code : Console

```
+-----------------------------------+
| TIMESTAMPADD(DAY, 5, '2011-05-21') |
+-----------------------------------+
| 2011-05-26                        |
+-----------------------------------+

+-------------------------------------------------+
| TIMESTAMPADD(MINUTE, 34, '2011-05-21 12:15:56') |
+-------------------------------------------------+
| 2011-05-21 12:49:56                             |
+-------------------------------------------------+
```

ADDTIME()

La fonction `ADDTIME(expr1, expr2)` permet d'ajouter *expr2* (de type `TIME`) à *expr1* (de type `DATETIME` ou `TIME`). Le résultat sera du même type que *expr1*.

Exemples

Code : SQL

```sql
-- Avec un DATETIME
SELECT NOW() AS Maintenant, ADDTIME(NOW(), '01:00:00') AS
DansUneHeure;

-- Avec un TIME
SELECT CURRENT_TIME() AS HeureCourante, ADDTIME(CURRENT_TIME(),
'03:20:02') AS PlusTard;
```

Code : Console

```
+---------------------+---------------------+
| Maintenant          | DansUneHeure        |
+---------------------+---------------------+
| 2011-11-07 15:32:47 | 2011-11-07 16:32:47 |
+---------------------+---------------------+

+---------------+----------+
| HeureCourante | PlusTard |
+---------------+----------+
| 15:32:47      | 18:52:49 |
+---------------+----------+
```

Soustraction d'un intervalle de temps

SUBDATE(), DATE_SUB() et SUBTIME()

`SUBDATE()`,`DATE_SUB()` et `SUBTIME()` sont les équivalents de `ADDDATE()`, `DATE_ADD()` et `ADDTIME()` pour la soustraction. Ils s'utilisent exactement de la même manière.

Exemples

Code : SQL

```
SELECT SUBDATE('2011-05-21 12:15:56', INTERVAL '3 02:10:32'
DAY_SECOND) AS SUBDATE_INTERVAL,
        SUBDATE('2011-05-21', 12) AS SUBDATE_nbJours,
        DATE_SUB('2011-05-21', INTERVAL 3 MONTH) AS DATE_SUB,
        SUBTIME('2011-05-21 12:15:56', '18:35:15') AS
SUBTIME_DATETIME,
        SUBTIME('12:15:56', '8:35:15') AS SUBTIME_TIME;
```

Code : Console

```
+---------------------+-----------------+------------+---------------------+-------
| SUBDATE_INTERVAL    | SUBDATE_nbJours | DATE_SUB   | SUBTIME_DATETIME    | SUBTIM
+---------------------+-----------------+------------+---------------------+-------
| 2011-05-18 10:05:24 | 2011-05-09      | 2011-02-21 | 2011-05-20 17:40:41 | 03:40:
+---------------------+-----------------+------------+---------------------+-------
```

Opérateur -

Tout comme l'opérateur + peut s'utiliser pour ajouter un intervalle de temps, il est possible d'utiliser l'opérateur − pour en soustraire un. Cependant, pour la soustraction, la date doit impérativement se trouver à gauche du signe −, et l'intervalle à droite. Il n'est en effet pas possible de soustraire une date d'un intervalle.

Exemple

Code : SQL

```
SELECT '2011-05-21' - INTERVAL 5 DAY;
```

Code : Console

```
+-------------------------------+
| '2011-05-21' - INTERVAL 5 DAY |
+-------------------------------+
| 2011-05-16                    |
+-------------------------------+
```

Soustraire, c'est ajouter un négatif

Un INTERVAL peut être défini avec une quantité négative, et ajouter un intervalle négatif, c'est soustraire un intervalle positif. De même soustraire un intervalle négatif revient à ajouter un intervalle positif.

Par conséquent, dans les requêtes suivantes, les deux parties du **SELECT** sont équivalentes.

Code : SQL

```
SELECT ADDDATE(NOW(), INTERVAL -3 MONTH), SUBDATE(NOW(), INTERVAL 3
MONTH);
SELECT DATE_ADD(NOW(), INTERVAL 4 HOUR), DATE_SUB(NOW(), INTERVAL -4
```

```
    HOUR);
    SELECT NOW() + INTERVAL -15 MINUTE, NOW() - INTERVAL 15 MINUTE;
```

Code : Console

```
+-----------------------------------+-----------------------------------+
| ADDDATE(NOW(), INTERVAL -3 MONTH) | SUBDATE(NOW(), INTERVAL 3 MONTH)  |
+-----------------------------------+-----------------------------------+
| 2011-09-01 16:04:26               | 2011-09-01 16:04:26               |
+-----------------------------------+-----------------------------------+

+-----------------------------------+-----------------------------------+
| DATE_ADD(NOW(), INTERVAL 4 HOUR)  | DATE_SUB(NOW(), INTERVAL -4 HOUR) |
+-----------------------------------+-----------------------------------+
| 2011-12-01 20:04:26               | 2011-12-01 20:04:26               |
+-----------------------------------+-----------------------------------+

+-----------------------------+-----------------------------+
| NOW() + INTERVAL -15 MINUTE | NOW() - INTERVAL 15 MINUTE  |
+-----------------------------+-----------------------------+
| 2011-12-01 15:49:28         | 2011-12-01 15:49:28         |
+-----------------------------+-----------------------------+
```

Divers

Créer une date/heure à partir d'autres informations

A partir d'un timestap unix

La fonction `FROM_UNIXTIME(ts)` renvoie un `DATETIME` à partir du timestamp unix *ts*.

Code : SQL

```
    SELECT FROM_UNIXTIME(1325595287);
```

Code : Console

```
+---------------------------+
| FROM_UNIXTIME(1325595287) |
+---------------------------+
| 2012-01-03 13:54:47       |
+---------------------------+
```

Notez que la fonction `UNIX_TIMESTAMP()`, que nous avons vue lors d'un chapitre précédent et qui donne le timestamp actuel, peut également s'utiliser avec un `DATETIME` en paramètre, auquel cas, elle fait l'inverse de la fonction `FROM_UNIXTIME(ts)` : elle renvoie le timestamp unix du `DATETIME` passé en paramètre.

Code : SQL

```
    SELECT UNIX_TIMESTAMP('2012-01-03 13:54:47');
```

Code : Console

```
+---------------------------------------+
| UNIX_TIMESTAMP('2012-01-03 13:54:47') |
+---------------------------------------+
|                            1325595287 |
+---------------------------------------+
```

A partir de différents éléments d'une date/heure

La fonction `MAKEDATE()` crée une `DATE` à partir d'une année et d'un numéro de jour (1 étant le premier janvier, 32 le premier février, etc.). Quant à la fonction `MAKETIME()`, elle crée un `TIME` à partir d'une heure et d'un nombre de minutes et de secondes.

Code : SQL

```sql
SELECT MAKEDATE(2012, 60) AS 60eJour2012, MAKETIME(3, 45, 34) AS
heureCree;
```

Code : Console

```
+-------------+-----------+
| 60eJour2012 | heureCree |
+-------------+-----------+
| 2012-02-29  | 03:45:34  |
+-------------+-----------+
```

Convertir un `TIME` en secondes, et vice versa

Il est toujours utile de connaître la fonction `SEC_TO_TIME()`, qui convertit un nombre de secondes en une donnée de type `TIME`, et son opposé `TIME_TO_SEC()`, qui convertit un `TIME` en un nombre de secondes.

Exemples

Code : SQL

```sql
SELECT SEC_TO_TIME(102569);
SELECT TIME_TO_SEC('01:00:30') as nb_secondes;
```

Code : Console

```
+----------------------+
| SEC_TO_TIME(102569)  |
+----------------------+
| 28:29:29             |
+----------------------+

+-------------+
| nb_secondes |
+-------------+
|        3630 |
+-------------+
```

Je rappelle qu'il est normal d'avoir un nombre d'heures supérieur à 24 dans un `TIME`, puisque `TIME` sert à stocker une heure ou une durée, et peut par conséquent aller de `'-838:59:59'` à `'838:59:59'`.

Dernier jour du mois

Enfin, voici la dernière fonction que nous verrons dans ce chapitre : `LAST_DAY(date)`. Cette fonction donne le dernier jour du mois de la date passée en paramètre. Cela permet par exemple de voir que 2012 est une année bissextile, contrairement à l'an 2100.

Code : SQL

```
SELECT LAST_DAY('2012-02-03') AS fevrier2012, LAST_DAY('2100-02-03')
AS fevrier2100;
```

Code : Console

```
+-------------+-------------+
| fevrier2012 | fevrier2100 |
+-------------+-------------+
| 2012-02-29  | 2100-02-28  |
+-------------+-------------+
```

Et voilà des calculs bien compliqués, rendus tout simples par les fonctions SQL !

Exercices

Nous avons maintenant vu une bonne partie des fonctions MySQL relatives aux données temporelles. N'oubliez pas de consulter la documentation officielle au besoin, car celle-ci contient plus de détails et d'autres fonctions.
Je vous propose maintenant quelques exercices pour passer de la théorie à la pratique. Cela va vous permettre de retenir déjà une partie des fonctions, mais surtout, cela va replacer ces fonctions dans un véritable contexte puisque nous allons bien sûr travailler sur notre table *Animal*.
Bien entendu, certaines questions ont plusieurs réponses possibles, mais je ne donnerai qu'une seule solution. Ne soyez donc pas étonné d'avoir trouver une requête faisant ce qui est demandé mais différente de la réponse indiquée.

Commençons par le format

1. Sélectionner tous les chiens nés en juin.

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT *
FROM Animal
WHERE MONTH(date_naissance) = 6;
```

Code : Console

```
+----+---------+---------------------+---------+-----------------+-----------+-
| id | sexe    | date_naissance      | nom     | commentaires    | espece_id | 
+----+---------+---------------------+---------+-----------------+-----------+-
|  6 | femelle | 2009-06-13 08:17:00 | Bobosse | Carapace bizarre |         3 | 
+----+---------+---------------------+---------+-----------------+-----------+-
```

2. Sélectionner tous les animaux nés dans les huit premières semaines d'une année.

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT *
FROM Animal
WHERE WEEKOFYEAR(date_naissance) < 9;
```

Code : Console

```
+----+---------+---------------------+----------+----------------------+--------
| id | sexe    | date_naissance      | nom      | commentaires         | espece_
+----+---------+---------------------+----------+----------------------+--------
| 11 | femelle | 2008-02-20 15:45:00 | Canaille | NULL                 |
| 15 | femelle | 2008-02-20 15:47:00 | Anya     | NULL                 |
| 31 | male    | 2008-02-20 15:45:00 | Filou    | NULL                 |
| 58 | male    | 2008-02-20 02:50:00 | Gingko   | Coco veut un gâteau ! |
+----+---------+---------------------+----------+----------------------+--------
```

3. Afficher le jour (en chiffres) et le mois de naissance (en toutes lettres) des tortues et chats nés avant 2008 (en deux colonnes).

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT DAY(date_naissance), MONTHNAME(date_naissance)
FROM Animal
WHERE YEAR(date_naissance) < 2008;
```

Code : Console

```
+---------------------+----------------------------+
| DAY(date_naissance) | MONTHNAME(date_naissance)  |
+---------------------+----------------------------+
|                  24 | avril                      |
|                  24 | avril                      |
|                  24 | avril                      |
|                  24 | avril                      |
|                  12 | avril                      |
|                  14 | mai                        |
|                  14 | mai                        |
|                  14 | mai                        |
|                  12 | mars                       |
|                  19 | mai                        |
|                  19 | mai                        |
|                  19 | mai                        |
|                  12 | mars                       |
|                  19 | mai                        |
|                  12 | mars                       |
|                  19 | mai                        |
|                   1 | avril                      |
|                  15 | mars                       |
|                   1 | avril                      |
|                  15 | mars                       |
|                   4 | mars                       |
+---------------------+----------------------------+
```

4. Même chose qu'à la question précédente, mais en une seule colonne.

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT DATE_FORMAT(date_naissance, '%e %M')
FROM Animal
WHERE YEAR(date_naissance) < 2008;
```

Code : Console

```
+--------------------------------------+
| DATE_FORMAT(date_naissance, '%e %M') |
+--------------------------------------+
```

```
| 24 avril                              |
| 24 avril                              |
| 24 avril                              |
| 24 avril                              |
| 12 avril                              |
| 14 mai                                |
| 14 mai                                |
| 14 mai                                |
| 12 March                              |
| 19 mai                                |
| 19 mai                                |
| 19 mai                                |
| 12 mars                               |
| 19 mai                                |
| 12 mars                               |
| 19 mai                                |
| 1 avril                               |
| 15 mars                               |
| 1 avril                               |
| 15 mars                               |
| 4 mars                                |
+---------------------------------------+
```

5. Sélectionner tous les animaux nés en mai, mais pas un 25 mai, triés par heure de naissance décroissante (heure dans le sens commun du terme donc heure, minutes, secondes) et afficher leur date de naissance suivant le même format que l'exemple ci-dessous.

Format : 8 janvier, à 6h30PM, en l'an 2010 après J.C.

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT DATE_FORMAT(date_naissance, '%e %M, à %lh%i%p, en l''an %Y après J.C.') AS jolie_date
FROM Animal
WHERE MONTH(date_naissance) = 5
AND DAY(date_naissance) <> 25
ORDER BY TIME(date_naissance) DESC;
```

Code : Console

```
+-----------------------------------------+
| jolie_date                              |
+-----------------------------------------+
| 19 mai, à 4h56PM, en l'an 2006 après J.C. |
| 19 mai, à 4h17PM, en l'an 2006 après J.C. |
| 19 mai, à 4h16PM, en l'an 2006 après J.C. |
| 19 mai, à 4h06PM, en l'an 2006 après J.C. |
| 19 mai, à 3h59PM, en l'an 2006 après J.C. |
| 14 mai, à 3h50PM, en l'an 2006 après J.C. |
| 14 mai, à 3h48PM, en l'an 2006 après J.C. |
| 14 mai, à 3h40PM, en l'an 2006 après J.C. |
| 26 mai, à 9h02AM, en l'an 2009 après J.C.  |
| 26 mai, à 8h56AM, en l'an 2009 après J.C.  |
| 26 mai, à 8h54AM, en l'an 2009 après J.C.  |
| 26 mai, à 8h50AM, en l'an 2009 après J.C.  |
| 14 mai, à 6h45AM, en l'an 2009 après J.C.  |
| 14 mai, à 6h42AM, en l'an 2009 après J.C.  |
```

```
| 14 mai, à 6h30AM, en l'an 2009 après J.C.  |
+--------------------------------------------+
```

Passons aux calculs

1. Moka était censé naître le 27 février 2008. Calculer le nombre de jours de retard de sa naissance.

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT DATEDIFF(date_naissance, '2008-02-27')  AS retard
FROM Animal
WHERE nom = 'Moka';
```

Code : Console

```
+--------+
| retard |
+--------+
|     12 |
+--------+
```

2. Afficher la date à laquelle chaque perroquet (espece_id = 4) fêtera son 25e anniversaire.

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT DATE(ADDDATE(date_naissance, INTERVAL 25 YEAR)) AS
Anniversaire
FROM Animal
WHERE espece_id = 4;
```

On ne demandait que la date (on fête rarement son anniversaire à l'heure pile de sa naissance), d'où l'utilisation de la fonction `DATE()`.

Code : Console

```
+--------------+
| Anniversaire |
+--------------+
| 2032-03-04   |
| 2033-02-20   |
| 2034-03-26   |
| 2034-03-26   |
+--------------+
```

3. Sélectionner les animaux nés dans un mois contenant exactement 29 jours.

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT  *
FROM Animal
WHERE DAY(LAST_DAY(date_naissance)) = 29;
```

Code : Console

```
+----+---------+---------------------+----------+-----------------------+--------
| id | sexe    | date_naissance      | nom      | commentaires          | espece_
+----+---------+---------------------+----------+-----------------------+--------
| 11 | femelle | 2008-02-20 15:45:00 | Canaille | NULL                  |
| 15 | femelle | 2008-02-20 15:47:00 | Anya     | NULL                  |
| 31 | male    | 2008-02-20 15:45:00 | Filou    | NULL                  |
| 58 | male    | 2008-02-20 02:50:00 | Gingko   | Coco veut un gâteau ! |
+----+---------+---------------------+----------+-----------------------+--------
```

4. Après douze semaines, un chaton est sevré (sauf exception bien sûr). Afficher la date à partir de laquelle les chats (espece_id = 2) de l'élevage peuvent être adoptés (qu'il s'agisse d'une date dans le passé ou dans le futur).

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT id, nom, DATE(DATE_ADD(date_naissance, INTERVAL 12 WEEK))
AS sevrage
FROM Animal
WHERE espece_id = 2;
```

Code : Console

```
+----+----------------+------------+
| id | nom            | sevrage    |
+----+----------------+------------+
|  2 | Roucky         | 2010-06-16 |
|  3 | Schtroumpfette | 2010-12-06 |
|  5 | Choupi         | 2010-12-26 |
|  8 | Bagherra       | 2008-12-04 |
| 29 | Fiero          | 2009-08-06 |
| 30 | Zonko          | 2007-06-04 |
| 31 | Filou          | 2008-05-14 |
| 33 | Caribou        | 2006-08-11 |
| 34 | Capou          | 2008-07-13 |
| 35 | Raccou         | 2006-08-11 |
| 36 | Boucan         | 2009-08-06 |
| 37 | Callune        | 2006-08-11 |
| 38 | Boule          | 2009-08-06 |
```

```
| 39 | Zara           | 2008-07-13 |
| 40 | Milla          | 2007-06-04 |
| 41 | Feta           | 2006-08-11 |
| 42 | Bilba          | 2008-07-13 |
| 43 | Cracotte       | 2007-06-04 |
| 44 | Cawette        | 2006-08-11 |
| 62 | Yoda           | 2011-02-01 |
| 66 | Chaman         | 2012-01-18 |
+----+----------------+------------+
```

5. Rouquine, Zira, Bouli et Balou (id 13, 18, 20 et 22 respectivement) font partie de la même portée. Calculer combien de temps, en minutes, Balou est-il né avant Zira.

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT TIMESTAMPDIFF(MINUTE,
     (SELECT date_naissance
     FROM Animal
     WHERE nom = 'Balou'),
     (SELECT date_naissance
     FROM Animal
     WHERE nom = 'Zira'))
  AS nb_minutes;
```

Il fallait ici penser aux sous-requêtes, afin d'obtenir les dates de naissance de Balou et Zira pour les utliser dans la même fonction `TIMESTAMPDIFF()`.

Code : Console

```
+------------+
| nb_minutes |
+------------+
|         14 |
+------------+
```

Et pour finir, mélangeons le tout

Et quand on dit "tout", c'est tout ! Par conséquent, il est possible (et même fort probable) que vous ayez besoin de notions et fonctions vues dans les chapitres précédents (regroupements, sous-requêtes, etc.) pour résoudre ces exercices.

1. Rouquine, Zira, Bouli et Balou (id 13, 18, 20 et 22 respectivement) font partie de la même portée. Calculer combien de temps, en minutes, s'est écoulé entre le premier né et le dernier né de la portée.

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT TIMESTAMPDIFF(MINUTE,
      (
```

```
            SELECT MIN(date_naissance)
            FROM Animal
            WHERE id IN (13, 18, 20, 22)
           ),
           (
            SELECT MAX(date_naissance)
            FROM Animal
            WHERE id IN (13, 18, 20, 22)
           )
          ) AS nb_minutes;
```

Presque le même exercice qu'au-dessus, à ceci près qu'il fallait utiliser les fonctions d'agrégation.

Code : Console

```
+-------------+
| nb_minutes  |
+-------------+
|          17 |
+-------------+
```

2. Calculer combien d'animaux sont nés durant un mois pendant lequel les moules sont les plus consommables (c'est-à-dire les mois finissant en "bre").

Secret (cliquez pour afficher)

Code : SQL

```
SELECT COUNT(*)
FROM Animal
WHERE MONTHNAME(date_naissance) LIKE '%bre';
```

Il faut bien sûr avoir préalablement défini que le nom des mois et des jours doit être exprimé en français. Je rappelle la requête à utiliser : `SET lc_time_names = 'fr_FR';`

Code : Console

```
+-----------+
| COUNT(*)  |
+-----------+
|         7 |
+-----------+
```

3. Pour les chiens et les chats (espece_id = 1 et espece_id = 2 respectivement), afficher les différentes dates de naissances des portées d'au moins deux individus (format JJ/MM/AAA), ainsi que le nombre d'individus pour chacune de ces portées. Attention, il n'est pas impossible qu'une portée de chats soit née le même jour qu'une portée de chiens (il n'est pas non plus impossible que deux portées de chiens naissent le même jour, mais on considère que ce n'est pas le cas).

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT DATE_FORMAT(date_naissance, '%d/%m/%Y'), COUNT(*) as
nb_individus
FROM Animal
WHERE espece_id IN (1, 2)
GROUP BY DATE(date_naissance), espece_id
HAVING nb_individus > 1;
```

Il faut regrouper sur la date (et uniquement la date, pas l'heure) puis sur l'espèce pour éviter de mélanger chiens et chats. Une simple clause **HAVING** permet ensuite de sélectionner les portées de deux individus ou plus.

Code : Console

```
+----------------------------------------+--------------+
| DATE_FORMAT(date_naissance, '%d/%m/%Y') | nb_individus |
+----------------------------------------+--------------+
| 14/05/2006                             |            3 |
| 19/05/2006                             |            5 |
| 12/03/2007                             |            3 |
| 24/04/2007                             |            4 |
| 20/02/2008                             |            2 |
| 10/03/2008                             |            3 |
| 20/04/2008                             |            3 |
| 14/05/2009                             |            3 |
| 26/05/2009                             |            4 |
+----------------------------------------+--------------+
```

4. Calculer combien de chiens (espece_id = 1) sont nés en moyenne chaque année entre 2006 et 2010 (sachant qu'on a eu au moins une naissance chaque année).

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT AVG(nb)
FROM (
 SELECT COUNT(*) AS nb
 FROM Animal
 WHERE espece_id = 1
 AND YEAR(date_naissance) >= 2006
 AND YEAR(date_naissance) <= 2010
 GROUP BY YEAR(date_naissance)
) AS tableIntermedaire;
```

Ici, il fallait penser à faire une sous-requête dans la clause **FROM**. Si vous n'avez pas trouvé, rappelez-vous qu'il faut penser par étapes. Vous voulez la moyenne du nombre de chiens nés chaque année, commencez par obtenir le nombre de chiens nés chaque année, puis seulement demandez-vous comment faire la moyenne.

Code : Console

```
+---------+
| AVG(nb) |
+---------+
|  4.2000 |
+---------+
```

5. Afficher la date au format ISO du 5e anniversaire des animaux dont on connait soit le père, soit la mère.

Secret (cliquez pour afficher)

Code : SQL

```sql
SELECT DATE_FORMAT(DATE_ADD(date_naissance, INTERVAL 5 YEAR),
GET_FORMAT(DATE, 'ISO')) AS dateIso
FROM Animal
WHERE pere_id IS NOT NULL
OR mere_id IS NOT NULL;
```

Pour cette dernière question, il fallait juste imbriquer plusieurs fonctions différentes. A nouveau, si vous n'avez pas réussi, c'est sans doute parce que vous ne décomposez pas le problème correctement.

Code : Console

```
+------------+
| dateIso    |
+------------+
| 2015-04-05 |
| 2015-03-24 |
| 2015-09-13 |
| 2015-07-21 |
| 2014-07-26 |
+------------+
```

Voilà qui clôt cette quatrième partie. L'utilisation des dates MySQL devrait maintenant vous sembler relativement simple.

À partir de maintenant, si j'en vois encore un stocker ses dates sous forme de chaîne de caractères, ou sous forme de `INT` pour stocker un timestamp UNIX, je le mords !
Avec cette partie sur les dates s'achèvent les parties "basiques" de ce tutoriel. Vous devriez maintenant avoir les connaissances suffisantes pour gérer la base de données d'un petit site web ou d'une application toute simple, sans toutefois exploiter vraiment les possibilités de MySQL. Les parties suivantes aborderont des notions un peu plus avancées.
Pour aller plus loin ou avoir des informations complémentaires, voici quelques liens qui pourraient vous être utiles.

- Documentation officielle de MySQL : à mettre dans vos favoris !
- Tutoriel W3School : c'est par là que j'ai moi-même commencé. Attention, site en anglais !
- Quelques tutoriels sur developpez.com : je n'ai pas tout lu, loin de là, mais vous devriez pouvoir y trouver pas mal d'informations.
- [SdZ] Choisir les bons types de champs SQL : un bon tutoriel sur le choix des types de colonnes.

- [SdZ] Jeux de caractères et interclassements : tutoriel très complet sur les jeux de caractères et les interclassements.
- Exercices SQL de Jérôme Darmont (Université de Lyon) : pas mal d'exercices, en commençant par le basique.

Ce tutoriel est en cours de rédaction. Les notions suivantes devraient être abordées aux cours des prochaines chapitres (liste non exhaustive) :

- Les structures conditionnelles
- Les vues
- Les triggers
- La gestion des utilisateurs et des droits

Tout commentaire, bon ou mauvais, est bienvenu !