**Base de données**

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

Prenons l'exemple d'un site web avec un système de news et de membres. On va utiliser une base de données MySQL pour stocker toutes les données du site : les news (avec la date de publication, le titre, le contenu, éventuellement l'auteur…) et les membres (leurs noms, leurs e-mails…).  
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 et interagir avec cette base**. Il faut pouvoir envoyer des messages à MySQL (messages que l'on appellera **"requêtes"**) afin de pouvoir ajouter des news, modifier des informations sur les membres, supprimer, et tout simplement afficher des éléments de la base.

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

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

**SGBD**

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

**Le paradigme client-serveur**

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

**SGBDR**

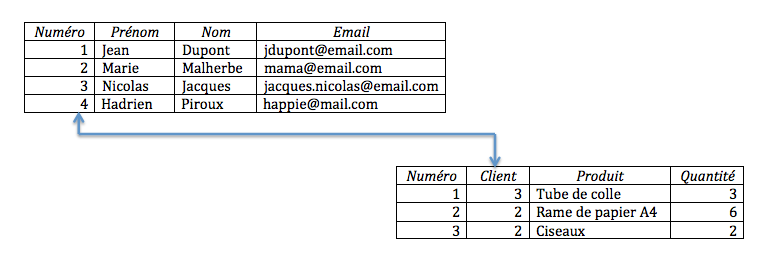
Le R de SGBDR signifie **"relationnel"**. Un SGBDR est un SGBD qui implémente la théorie relationnelle. MySQL implémente la théorie relationnelle ; c'est donc un SGBDR.

La théorie relationnelle dépasse le cadre de ce tutoriel, mais ne vous inquiétez pas, il n'est pas nécessaire de la maîtriser pour être capable d'utiliser convenablement un SGBDR. Il vous suffit de savoir que, dans un SGBDR, les données sont contenues dans ce que l'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 un en-tête avec quatre attributs : numéro, prénom, nom et e-mail.

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 (soit AA et BB deux relations) :

* la sélection (ou restriction) : obtenir les lignes de AA répondant à certains critères ;
* la projection : obtenir une partie des attributs des lignes de AA ;
* l'union - A∪BA∪B  : obtenir tout ce qui se trouve dans la relation AA ou dans la relation BB ;
* l'intersection - A∩BA∩B  : obtenir tout ce qui se trouve à la fois dans la relation AA et dans la relation BB ;
* la différence - A−BA−B  : obtenir ce qui se trouve dans la relation AA , mais pas dans la relation BB ;
* la jointure - A⋈BA⋈B  : obtenir l'ensemble des lignes provenant de la liaison de la relation AA et de la relation BB à l'aide d'une information commune.

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



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 c'est devenu standard en 1986 (pour la norme ANSI ; 1987 en ce qui concerne la norme ISO). Il est encore régulièrement amélioré.

### Présentation succincte de MySQL...



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

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

#### Un peu d'histoi

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

En 2008, MySQL AB est rachetée par la société Sun Microsystems, qui est elle-même rachetée par Oracle Corporation en 2010.  
On craint alors la fin de la gratuité de MySQL, étant donné qu'Oracle Corporation édite un des grands concurrents de MySQL : Oracle Database, qui est payant (et très cher). Oracle promit cependant de continuer à développer MySQL et de conserver la double licence GPL (libre) et commerciale jusqu'en 2015 au moins. À ce jour, les deux licences existent toujours.

#### 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'essaierai 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. Enfin, il est très permissif, et acceptera donc des requêtes qui généreraient une erreur sous d'autres SGBDR.

### ... et de ses concurrents

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

#### Oracle Database



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

#### PostgreSQL



Comme MySQL, PostgreSQL est un logiciel open source. Il est cependant moins utilisé, notamment par les débutants, car moins connu. La raison de cette méconnaissance réside sans doute en partie dans le fait que PostgreSQL a longtemps été disponible uniquement sous Unix. La première version Windows n'est apparue qu'à la sortie de la version 8.0 du logiciel, en 2005.  
PostgreSQL a longtemps été plus performant que MySQL, mais ces différences tendent à diminuer. MySQL semble être aujourd'hui équivalent à PostgreSQL en termes 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.

#### SQLite

Logo SQLite

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

### Organisation d'une base de données

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

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

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

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

**Utilisateur**

Il n'est pas très conseillé de travailler en tant que "root" dans MySQL, à moins d'en avoir spécifiquement besoin. En effet, "root" a tous les droits. Ce qui signifie que vous pouvez faire n'importe quelle bêtise dans n'importe quelle base de données pendant que j'ai le dos tourné. Pour éviter cela, nous allons créer un nouvel utilisateur qui aura des droits très restreints. Je l’appellerai "student", 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 autre chapitre. Nous verrons donc uniquement comment créer un utilisateur n'ayant accès qu'à une seule base de données. Tapez donc ces commandes dans mysql, en remplaçant *student* par le nom d'utilisateur que vous avez choisi, et *mot\_de\_passe* par le mot de passe que vous voulez lui attribuer :

CREATE USER 'student'@'localhost' IDENTIFIED BY 'mot\_de\_passe';

GRANT ALL PRIVILEGES ON elevage.\* TO 'student'@'localhost';

Je décortique donc rapidement :

* CREATE USER 'student'  : cette commande crée l'utilisateur *student.*
* @'localhost'  : définit à partir d'où l'utilisateur peut se connecter. Dans notre cas, 'localhost', donc il devra être connecté à partir de cet ordinateur.
* IDENTIFIED BY 'mot\_de\_passe'  : définit le mot de passe de l'utilisateur.
* GRANT ALL PRIVILEGES  : cette commande permet d'attribuer tous les droits (c'est-à-dire insertions de données, sélections, modifications, suppressions…).
* ON 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 'student'@'localhost'  : définit l'utilisateur (et son hôte) auquel on accorde ces droits.

## Distinguez les différents types de données

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

### Avertissement

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 donnée pourrait entraîner :

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

#### Nombres entiers

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

##### NUMERIC et DECIMAL

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

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

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

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

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

##### FLOAT, DOUBLE et REAL

Le mot-clé FLOAT  peut s'utiliser sans paramètres, 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 comparaisons 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). 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 pour le champ, celui-ci sera tronqué.

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

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

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

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

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

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

**SET et ENUM**

**ENUM**

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

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

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

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

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

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

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

Un ENUM  peut avoir au 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 que l'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 :

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

On pourra stocker dans cette colonne :

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

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

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

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

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

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

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

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

##### Avertissement

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

### 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 ! :D

##### DATE

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

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

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

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

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

##### DATETIME

Très proche de DATE, ce type permet de stocker une heure, en plus d'une date. 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 l'on utilise une chaîne de caractères, il faut séparer la date et l'heure par une espace. Quelques exemples corrects (H représente les heures, M les minutes et S les secondes) :

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

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

##### TIME

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

* 'HH:MM:SS'
* 'HHH:MM:SS'
* 'MM:SS'
* 'J HH:MM:SS'
* 'HHMMSS'
* HHMMSS  (nombre)

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

#### YEAR

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

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

#### TIMESTAMP

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

Le type TIMESTAMP  de MySQL est cependant un peu particulier. Prenons par exemple le 4 octobre 2011, à 21 h 5 min 51 s.  
Entre cette date et le 1er janvier 1970, 0 h 0 min 0 s, il s'est écoulé exactement 1 317 755 151 secondes. Le nombre 1 317 755 151 est donc, par définition, le timestamp de cette date du 4 octobre 2011, 21 h 5 min 51 s.  
Pourtant, pour stocker cette date dans un TIMESTAMP  SQL, ce n'est pas 1 317 755 151 que l'on utilisera, mais 20111004210551. C'est-à-dire l'équivalent, au format numérique, du DATETIME '2011-10-04 21:05:51'.  
Le TIMESTAMP  SQL n'a donc de timestamp que le nom. Il ne sert pas à stocker un nombre de secondes, mais bien une date sous format numérique AAAAMMJJHHMMSS (alors qu'un DATETIME  est stocké sous forme de chaîne de caractères).

Il n'est donc pas possible de stocker un "vrai" timestamp dans une colonne de type TIMESTAMP. C'est évidemment contre-intuitif, et source d'erreurs.  
Notez que, malgré cela, le TIMESTAMP  SQL a les mêmes limites qu'un vrai timestamp : il n'acceptera que des dates entre le 1er janvier 1970 à 0 h 0 min 0 s et le 19 janvier 2038 à 3 h 14 min 7 s.

#### La date par défaut

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

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

### 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 choisissez un autre nom de base, vous n'aurez aucun droit dessus.

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

CREATE DATABASE nom\_base;

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

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

CREATE DATABASE elevage CHARACTER *SET* 'utf8';

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

#### Suppression

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

DROP DATABASE elevage;

#### 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 d'abord la sélectionner. Une fois de plus, la commande est très simple :

USE elevage

#### Clé primaire

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

#### Auto-incrémentation

Il faut donc, pour chaque animal, décider d'une valeur pour id. Le plus simple, et le plus logique, est de donner le numéro 1 au premier individu enregistré, puis le numéro 2 au second, etc.  
Mais si vous ne vous souvenez pas quel numéro vous avez utilisé en dernier, pour insérer un nouvel animal il faudra récupérer cette information dans la base. Ensuite seulement, vous pourrez ajouter une ligne en lui donnant comme id le dernier id utilisé + 1.  
C'est bien sûr faisable, mais c'est fastidieux… Heureusement, il est possible de demander à MySQL de faire tout cela 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 y mettre lors de la prochaine insertion. MySQL va chercher cela tout seul, comme un grand, en prenant la dernière valeur insérée dans cette colonne et en l'incrémentant de 1.

### Les moteurs de tables

Les moteurs de tables sont une spécificité de MySQL. Ce sont des moteurs de stockage. Cela permet de gérer différemment les tables selon l'utilité que l'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](https://dev.mysql.com/doc/refman/5.5/en/storage-engines.html).  
Les deux moteurs les plus connus sont **MyISAM**et **InnoDB**.

##### MyISAM

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

##### InnoDB

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

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

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

ENGINE = moteur;

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

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

##### Création de la table

CREATE TABLE [IF NOT EXISTS] Nom\_table (

colonne1 description\_colonne1,

[colonne2 description\_colonne2,

colonne3 description\_colonne3,

...,]

[PRIMARY KEY (colonne\_clé\_primaire)]

)

[ENGINE=moteur];

Le IF NOT EXISTS  est facultatif (d'où l'utilisation de crochets [ ]), et a le même rôle que dans la commande CREATE DATABASE  : si une table de ce nom existe déjà dans la base de données, la requête renverra un warning plutôt qu'une erreur si IF NOT EXISTS  est spécifié.  
Ce n'est pas non plus une erreur de ne pas préciser la clé primaire directement à la création de la table. Il est tout à fait possible de l'ajouter par la suite. Nous verrons comment un peu plus tard.

##### Définition des colonnes

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

nom *VARCHAR*(30),

sexe *CHAR*(1)

#### 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 :

CREATE TABLE Animal (

id SMALLINT UNSIGNED NOT NULL AUTO\_INCREMENT,

espece *VARCHAR*(40) NOT NULL,

sexe *CHAR*(1),

date\_naissance DATETIME NOT NULL,

nom *VARCHAR*(30),

commentaires *TEXT*,

PRIMARY KEY (id)

)

ENGINE=INNODB;

## Modifiez une table

La création et la suppression de tables étant acquises, parlons maintenant des requêtes permettant de modifier une table.  
Plus précisément, ce chapitre portera sur la modification des colonnes d'une table (ajout d'une colonne, modification, suppression de colonnes).  
Il est possible de modifier d'autres éléments (des contraintes ou des index, par exemple), mais cela nécessite des notions que vous ne possédez pas encore, aussi n'en parlerai-je pas ici.

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

### Syntaxe de la requête

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

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  :

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 :

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

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

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 :

ALTER TABLE nom\_table

DROP [COLUMN] nom\_colonne;

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

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

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 :

ALTER TABLE nom\_table

CHANGE ancien\_nom nouveau\_nom description\_colonne;

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

ALTER TABLE Test\_tuto

CHANGE nom prenom *VARCHAR*(10) NOT NULL;

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

#### Changement du type de données

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

ALTER TABLE nom\_table

CHANGE ancien\_nom nouveau\_nom nouvelle\_description;

ALTER TABLE nom\_table

MODIFY nom\_colonne nouvelle\_description;

Des exemples pour illustrer :

ALTER TABLE Test\_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](https://dev.mysql.com/doc/refman/5.5/en/alter-table.html).

## Insérez des données

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

Nous verrons entre autres :

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

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

### Syntaxe de INSERT

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

#### Insertion sans préciser les colonnes

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

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

INSERT INTO Animal

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

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

INSERT INTO Animal

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

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

INSERT INTO Animal

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

#### Insertion en précisant les colonnes

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

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

Quelques exemples :

INSERT INTO Animal (espece, sexe, date\_naissance)

VALUES ('tortue', 'F', '2009-08-03 05:12:00');

INSERT INTO Animal (nom, commentaires, date\_naissance, espece)

VALUES ('Choupi', 'Né sans oreille gauche', '2010-10-03 16:44:00', 'chat');

INSERT INTO Animal (espece, date\_naissance, commentaires, nom, sexe)

VALUES ('tortue', '2009-06-13 08:17:00', 'Carapace bizarre', 'Bobosse', 'F');

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

#### Insertion multiple

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

INSERT INTO Animal (espece, sexe, date\_naissance, nom)

VALUES ('chien', 'F', '2008-12-06 05:18:00', 'Caroline'),

('chat', 'M', '2008-09-11 15:38:00', 'Bagherra'),

('tortue', NULL, '2010-08-23 05:18:00', NULL);

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

### Syntaxe alternative de MySQL

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

INSERT INTO Animal

*SET* nom='Bobo', espece='chien', sexe='M', date\_naissance='2010-07-21 15:41:00';

Cette syntaxe présente deux avantages.

* Le fait d'avoir l'un à côté de l'autre la colonne et la valeur qu'on lui attribue (nom = 'Bobo') rend la syntaxe plus lisible et plus facile à manipuler. En effet, ici, il n'y a que six colonnes, mais imaginez une table avec 20, voire 100 colonnes. Difficile d'être sûr que l'ordre dans lequel on a déclaré les colonnes est bien le même que l'ordre des valeurs qu'on leur donne…
* Elle est très semblable à la syntaxe de UPDATE, que nous verrons plus tard et qui permet de modifier des données existantes. C'est donc moins de choses à retenir (mais bon, une requête de plus ou de moins, ce n'est pas non plus énorme…).

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

En effet, cette syntaxe présente deux défauts majeurs.

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

### Utilisation de fichiers externes

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

Écrire toutes les commandes à la main dans la console 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, cela peut être assez long.

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

SOURCE monFichier.sql;

Ou

\. monFichier.sql;

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

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

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

SOURCE Users\taguan\dossierX\monFichier.sql;

Si vous utilisez Windows, utilisez "/" au lieu de "\"

SOURCE C:/"Document and Settings"/dossierX/monFichier.sql;

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

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

nom;prenom;date\_naissance

Charles;Myeur;1994-12-30

Bruno;Debor;1978-05-12

Mireille;Franelli;1990-08-23

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

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 du serveur, il est obligatoire, pour des raisons de sécurité, qu'il soit dans le répertoire de la base de données, c'est-à-dire dans le répertoire créé par MySQL à la création de la base de données, qui contient les fichiers dans lesquels sont stockées les données de la base. Pour ma part, j'utiliserai toujours LOCAL, afin de pouvoir mettre simplement mes fichiers dans mon dossier de travail.

Les clauses FIELDS  et LINES  permettent de définir le format de fichier utilisé. FIELDS  se rapporte aux colonnes, et LINES  aux lignes (si si ^^ ). Ces deux clauses sont facultatives. Les valeurs que j'ai mises ci-dessus sont les valeurs par défaut.

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

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

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

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

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

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

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

nom;prenom;date\_naissance

Charles;Myeur;1994-12-30

Bruno;Debor;1978-05-12

Mireille;Franelli;1990-08-23

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

LOAD DATA LOCAL INFILE 'personne.csv'

INTO TABLE Personne

FIELDS TERMINATED BY ';'

LINES TERMINATED BY '\n' -- ou '\r\n' selon l'ordinateur et le programme utilisés pour créer le fichier

IGNORE 1 LINES

(nom,prenom,date\_naissance);

### Remplissage de la base

Nous allons utiliser les deux techniques que je viens de vous montrer pour remplir un peu notre base. N'oubliez pas de modifier les commandes données pour ajouter le chemin vers vos fichiers.

#### Exécution de commandes SQL

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

INSERT INTO Animal (espece, sexe, date\_naissance, nom, commentaires) VALUES

('chien', 'F', '2008-02-20 15:45:00' , 'Canaille', NULL),

('chien', 'F','2009-05-26 08:54:00' , 'Cali', NULL),

('chien', 'F','2007-04-24 12:54:00' , 'Rouquine', NULL),

('chien', 'F','2009-05-26 08:56:00' , 'Fila', NULL),

('chien', 'F','2008-02-20 15:47:00' , 'Anya', NULL),

('chien', 'F','2009-05-26 08:50:00' ,'Louya' , NULL),

('chien', 'F', '2008-03-10 13:45:00','Welva' , NULL),

('chien', 'F','2007-04-24 12:59:00' ,'Zira' , NULL),

('chien', 'F', '2009-05-26 09:02:00','Java' , NULL),

('chien', 'M','2007-04-24 12:45:00' ,'Balou' , NULL),

('chien', 'M','2008-03-10 13:43:00' ,'Pataud' , NULL),

('chien', 'M','2007-04-24 12:42:00' , 'Bouli', NULL),

('chien', 'M', '2009-03-05 13:54:00','Zoulou' , NULL),

('chien', 'M','2007-04-12 05:23:00' ,'Cartouche' , NULL),

('chien', 'M', '2006-05-14 15:50:00', 'Zambo', NULL),

('chien', 'M','2006-05-14 15:48:00' ,'Samba' , NULL),

('chien', 'M', '2008-03-10 13:40:00','Moka' , NULL),

('chien', 'M', '2006-05-14 15:40:00','Pilou' , NULL),

('chat', 'M','2009-05-14 06:30:00' , 'Fiero', NULL),

('chat', 'M','2007-03-12 12:05:00' ,'Zonko', NULL),

('chat', 'M','2008-02-20 15:45:00' , 'Filou', NULL),

('chat', 'M','2007-03-12 12:07:00' , 'Farceur', NULL),

('chat', 'M','2006-05-19 16:17:00' ,'Caribou' , NULL),

('chat', 'M','2008-04-20 03:22:00' , 'Capou', NULL),

('chat', 'M','2006-05-19 16:56:00' , 'Raccou', 'Pas de queue depuis la naissance');

Vous n'avez alors qu'à taper :

SOURCE remplissageAnimal.sql;

#### LOAD DATA INFILE

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

"chat";"M";"2009-05-14 06:42:00";"Boucan";NULL

"chat";"F";"2006-05-19 16:06:00";"Callune";NULL

"chat";"F";"2009-05-14 06:45:00";"Boule";NULL

"chat";"F";"2008-04-20 03:26:00";"Zara";NULL

"chat";"F";"2007-03-12 12:00:00";"Milla";NULL

"chat";"F";"2006-05-19 15:59:00";"Feta";NULL

"chat";"F";"2008-04-20 03:20:00";"Bilba";"Sourde de l'oreille droite à 80%"

"chat";"F";"2007-03-12 11:54:00";"Cracotte";NULL

"chat";"F";"2006-05-19 16:16:00";"Cawette";NULL

"tortue";"F";"2007-04-01 18:17:00";"Nikki";NULL

"tortue";"F";"2009-03-24 08:23:00";"Tortilla";NULL

"tortue";"F";"2009-03-26 01:24:00";"Scroupy";NULL

"tortue";"F";"2006-03-15 14:56:00";"Lulla";NULL

"tortue";"F";"2008-03-15 12:02:00";"Dana";NULL

"tortue";"F";"2009-05-25 19:57:00";"Cheli";NULL

"tortue";"F";"2007-04-01 03:54:00";"Chicaca";NULL

"tortue";"F";"2006-03-15 14:26:00";"Redbul";"Insomniaque"

"tortue";"M";"2007-04-02 01:45:00";"Spoutnik";NULL

"tortue";"M";"2008-03-16 08:20:00";"Bubulle";NULL

"tortue";"M";"2008-03-15 18:45:00";"Relou";"Surpoids"

"tortue";"M";"2009-05-25 18:54:00";"Bulbizard";NULL

"perroquet";"M";"2007-03-04 19:36:00";"Safran";NULL

"perroquet";"M";"2008-02-20 02:50:00";"Gingko";NULL

"perroquet";"M";"2009-03-26 08:28:00";"Bavard";NULL

"perroquet";"F";"2009-03-26 07:55:00";"Parlotte";NULL

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

Exécutez ensuite la commande suivante :

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);

## Sélectionnez des données

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

Au menu :

* syntaxe de la requête SELECT  (que vous avez déjà croisée il y a quelque temps) ;
* sélection de données répondant à certaines conditions ;
* tri des données ;
* élimination des données en double ;
* récupération d'une partie des données seulement (uniquement les 10 premières lignes, par exemple).
* **Exemple :**
* SELECT 'Hello World !';
* SELECT 3+2;
* SELECT  permet également de sélectionner des données à partir d'une table. Pour cela, il faut ajouter une clause à la commande SELECT  : la clause FROMqui définit de quelle structure (dans notre cas, une table) viennent les données.
* SELECT colonne1, colonne2, ...
* FROM nom\_table;
* Par exemple, si l'on veut sélectionner l'espèce, le nom et le sexe des animaux présents dans la table Animal, on utilisera :
* 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 :
* SELECT \*
* FROM Animal;

### La clause WHERE

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

SELECT \*

FROM Animal

WHERE espece='chien';

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

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  :

SELECT \*

FROM Animal

WHERE espece='chat'

AND sexe='F';

-- OU

SELECT \*

FROM Animal

WHERE espece='chat'

&& sexe='F';

##### OR

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

SELECT \*

FROM Animal

WHERE espece='tortue'

OR espece='perroquet';

-- OU

SELECT \*

FROM Animal

WHERE espece='tortue'

|| espece='perroquet';

##### NOT

Sélection de tous les animaux femelles sauf les chiennes.

SELECT \*

FROM Animal

WHERE sexe='F'

AND NOT espece='chien';

-- OU

SELECT \*

FROM Animal

WHERE sexe='F'

AND ! (espece='chien');

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

##### XOR

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

SELECT \*

FROM Animal

WHERE sexe='M'

XOR espece='perroquet';

**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, il n'y a plus qu'à écrire la requête avec les bons opérateurs logiques !

SELECT \*

FROM Animal

WHERE date\_naissance > '2009-12-31'

OR

( espece='chat'

AND

( sexe='M'

OR

( sexe='F' 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 probablement 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 je ne vous ai pas donné d'explication sur cette mystérieuse précision "aussi valable pour NULL" :-° . Mais je vais me rattraper maintenant !  
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 :

SELECT \*

FROM Animal

WHERE nom = NULL; -- sélection des animaux sans nom

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 :

SELECT \*

FROM Animal

WHERE nom <=> NULL; -- sélection des animaux sans nom

-- OU

SELECT \*

FROM Animal

WHERE nom IS NULL;

SELECT \*

FROM Animal

WHERE commentaires IS NOT NULL; -- sélection des animaux pour lesquels un commentaire existe

### Tri des données

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 pourraient remplacer les anciennes dans l'ordre de MySQL. Or, bien souvent, vous voudrez trier à votre manière. Par date de naissance par exemple, ou bien par espèce, ou par sexe, etc.

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

Par exemple, pour trier par date de naissance :

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 cela 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, etDESC  pour descendant. Par défaut, si vous ne précisez rien, c'est un tri ascendant qui est effectué : du plus petit nombre au plus grand, de la date la plus ancienne à la plus récente, et pour les chaînes de caractères et les textes, c'est l'ordre alphabétique normal qui est utilisé.  
Si, par contre, vous utilisez le mot DESC, l'ordre est inversé : plus grand nombre d'abord, date la plus récente d'abord, et ordre anti-alphabétique pour les caractères.

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

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  :

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. Non pas parce que MySQL fait des bêtises, mais tout simplement parce que certaines informations sont présentes plusieurs fois dans la table.

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

SELECT espece

FROM Animal;

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

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

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. Cela peut paraître logique, mais cela en perd plus d'un. Ne seront donc prises en compte que les colonnes que vous avez précisées dans votre SELECT. Uniquement espece donc, dans notre exemple.

### Restreindre les résultats

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

#### Syntaxe

LIMIT  s'utilise avec deux paramètres

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

LIMIT nombre\_de\_lignes [OFFSET decalage];

**Exemple :**

SELECT \*

FROM 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 petits id, puisque nous n'avons demandé aucun décalage (OFFSET 0).

**Exemple avec un seul paramètre :**

SELECT \*

FROM Animal

ORDER BY id

LIMIT 10;

Cette requête est donc équivalente à :

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.

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.

##### Rechercher '%' ou '\_'

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

**Exemple :**

SELECT \*

FROM Animal

WHERE commentaires LIKE '%\%%';

Résultat :

| **id** | **espece** | **sexe** | **date\_naissance** | **nom** | **commentaires** |
| --- | --- | --- | --- | --- | --- |
| 42 | chat | F | 2008-04-20 03:20:00 | Bilba | Sourde de l'oreille droite à 80% |

##### Exclure une chaîne de caractères

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

SELECT \*

FROM 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). Pour rappel, ceci est dû à l'interclassement. Nous avons gardé l'interclassement par défaut du jeu de caractères UTF-8, qui n'est pas sensible à la casse.  
Si vous désirez faire une recherche sensible à la casse, vous pouvez définir votre chaîne de recherche comme une chaîne de type binaire, et non plus comme une simple chaîne de caractères :

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 pure.

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

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 :

SELECT \*

FROM Animal

WHERE date\_naissance <= '2009-03-23'

AND date\_naissance >= '2008-01-05';

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

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 :

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 :( ? Eh bien, réjouissez-vous, car IN  est dans la place ! Cet opérateur vous permet de faire des recherches dans 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  :

SELECT \*

FROM Animal

WHERE nom IN ('Moka', 'Bilba', 'Tortilla', 'Balou', 'Dana', 'Redbul', 'Gingko');

### 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 :pirate:) et toutes les données peuvent disparaître. MySQL dispose donc d'un outil spécialement dédié à la sauvegarde des données sous forme de fichiers texte : mysqldump.

Cette fonction de sauvegarde s'utilise à partir de la console. Vous devez donc être déconnecté de MySQL pour la lancer. Si vous êtes connecté, tapez simplement  exit.

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

mysqldump -u user -p --opt nom\_de\_la\_base > sauvegarde.sql

Décortiquons cette commande.

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

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

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.

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 le dossier désiré. Ex. : C:\"Mes Documents"\mysql\sauvegardes\elevage\_sauvegarde.sql  au lieu de  elevage\_sauvegarde.sql

La base de données est donc sauvegardée. Notez que la commande pour créer la base elle-même n'est pas 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 exécuter la commande suivante (dans la console) :

mysql nom\_base < chemin\_fichier\_de\_sauvegarde.sql

Concrètement, dans notre cas :

mysql elevage < elevage\_sauvegarde.sql

Voici donc une troisième manière d'exécuter des commandes SQL en utilisant un fichier externe (en plus des commandes  SOURCE  et  LOAD DATA vues précédemment). Vous pouvez simplement, lorsque vous vous connectez à votre base de données via la ligne de commande, préciser un fichier contenant vos commandes SQL, en le précédant du caractère <. Elles seront alors exécutées une à une. Attention de ne pas oublier de sélectionner la base de données sur laquelle vous voulez travailler, soit directement à la connexion, soit en précisant le  USE nom\_de\_la\_base;  au début du fichier.

Ou directement à partir de MySQL :

USE nom\_base;

SOURCE fichier\_de\_sauvegarde.sql;

Donc :

USE elevage;

SOURCE elevage\_sauvegarde.sql;

Vous savez maintenant sauvegarder de manière simple vos bases de données. Notez que je ne vous ai donné ici qu'une manière d'utiliser mysqldump. En effet, cette commande possède de nombreuses options. Si cela vous intéresse, je vous renvoie à [**la documentation de MySQL**](https://dev.mysql.com/doc/refman/5.5/en/mysqldump.html) 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.

DELETE FROM nom\_table

WHERE critères;

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

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.

DELETE FROM Animal;

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

### Modification

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

UPDATE nom\_table

*SET* col1 = val1 [, col2 = val2, ...]

[WHERE ...];

Par exemple, 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 :

UPDATE Animal

*SET* sexe='F', 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 prudent !

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

UPDATE Animal

*SET* commentaires='modification de toutes les lignes';

## Index

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

**État actuel de la base de données**  
**Note :** les tables de test ne sont pas reprises.

*SET* NAMES utf8;

DROP TABLE IF EXISTS Animal;

CREATE TABLE Animal (

id smallint(6) UNSIGNED NOT NULL AUTO\_INCREMENT,

espece *varchar*(40) NOT NULL,

sexe *char*(1) DEFAULT NULL,

date\_naissance datetime NOT NULL,

nom *varchar*(30) DEFAULT NULL,

commentaires *text*,

PRIMARY KEY (id)

) ENGINE=InnoDB AUTO\_INCREMENT=61 DEFAULT CHARSET=utf8;

LOCK TABLES Animal WRITE;

INSERT INTO Animal VALUES

(1,'chien','M','2010-04-05 13:43:00','Rox','Mordille beaucoup'),(2,'chat',NULL,'2010-03-24 02:23:00','Roucky',NULL),(3,'chat','F','2010-09-13 15:02:00','Schtroumpfette',NULL),

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

(7,'chien','F','2008-12-06 05:18:00','Caroline',NULL),(8,'chat','M','2008-09-11 15:38:00','Bagherra',NULL),(9,'tortue',NULL,'2010-08-23 05:18:00',NULL,NULL),

(10,'chien','M','2010-07-21 15:41:00','Bobo',NULL),(11,'chien','F','2008-02-20 15:45:00','Canaille',NULL),(12,'chien','F','2009-05-26 08:54:00','Cali',NULL),

(13,'chien','F','2007-04-24 12:54:00','Rouquine',NULL),(14,'chien','F','2009-05-26 08:56:00','Fila',NULL),(15,'chien','F','2008-02-20 15:47:00','Anya',NULL),

(16,'chien','F','2009-05-26 08:50:00','Louya',NULL),(17,'chien','F','2008-03-10 13:45:00','Welva',NULL),(18,'chien','F','2007-04-24 12:59:00','Zira',NULL),

(19,'chien','F','2009-05-26 09:02:00','Java',NULL),(20,'chien','M','2007-04-24 12:45:00','Balou',NULL),(21,'chien','F','2008-03-10 13:43:00','Pataude',NULL),

(22,'chien','M','2007-04-24 12:42:00','Bouli',NULL),(24,'chien','M','2007-04-12 05:23:00','Cartouche',NULL),(25,'chien','M','2006-05-14 15:50:00','Zambo',NULL),

(26,'chien','M','2006-05-14 15:48:00','Samba',NULL),(27,'chien','M','2008-03-10 13:40:00','Moka',NULL),(28,'chien','M','2006-05-14 15:40:00','Pilou',NULL),

(29,'chat','M','2009-05-14 06:30:00','Fiero',NULL),(30,'chat','M','2007-03-12 12:05:00','Zonko',NULL),(31,'chat','M','2008-02-20 15:45:00','Filou',NULL),

(32,'chat','M','2007-03-12 12:07:00','Farceur',NULL),(33,'chat','M','2006-05-19 16:17:00','Caribou',NULL),(34,'chat','M','2008-04-20 03:22:00','Capou',NULL),

(35,'chat','M','2006-05-19 16:56:00','Raccou','Pas de queue depuis la naissance'),(36,'chat','M','2009-05-14 06:42:00','Boucan',NULL),(37,'chat','F','2006-05-19 16:06:00','Callune',NULL),

(38,'chat','F','2009-05-14 06:45:00','Boule',NULL),(39,'chat','F','2008-04-20 03:26:00','Zara',NULL),(40,'chat','F','2007-03-12 12:00:00','Milla',NULL),

(41,'chat','F','2006-05-19 15:59:00','Feta',NULL),(42,'chat','F','2008-04-20 03:20:00','Bilba','Sourde de l''oreille droite à 80%'),(43,'chat','F','2007-03-12 11:54:00','Cracotte',NULL),

(44,'chat','F','2006-05-19 16:16:00','Cawette',NULL),(45,'tortue','F','2007-04-01 18:17:00','Nikki',NULL),(46,'tortue','F','2009-03-24 08:23:00','Tortilla',NULL),

(47,'tortue','F','2009-03-26 01:24:00','Scroupy',NULL),(48,'tortue','F','2006-03-15 14:56:00','Lulla',NULL),(49,'tortue','F','2008-03-15 12:02:00','Dana',NULL),

(50,'tortue','F','2009-05-25 19:57:00','Cheli',NULL),(51,'tortue','F','2007-04-01 03:54:00','Chicaca',NULL),(52,'tortue','F','2006-03-15 14:26:00','Redbul','Insomniaque'),

(53,'tortue','M','2007-04-02 01:45:00','Spoutnik',NULL),(54,'tortue','M','2008-03-16 08:20:00','Bubulle',NULL),(55,'tortue','M','2008-03-15 18:45:00','Relou','Surpoids'),

(56,'tortue','M','2009-05-25 18:54:00','Bulbizard',NULL),(57,'perroquet','M','2007-03-04 19:36:00','Safran',NULL),(58,'perroquet','M','2008-02-20 02:50:00','Gingko',NULL),

(59,'perroquet','M','2009-03-26 08:28:00','Bavard',NULL),(60,'perroquet','F','2009-03-26 07:55:00','Parlotte',NULL);

UNLOCK TABLES;

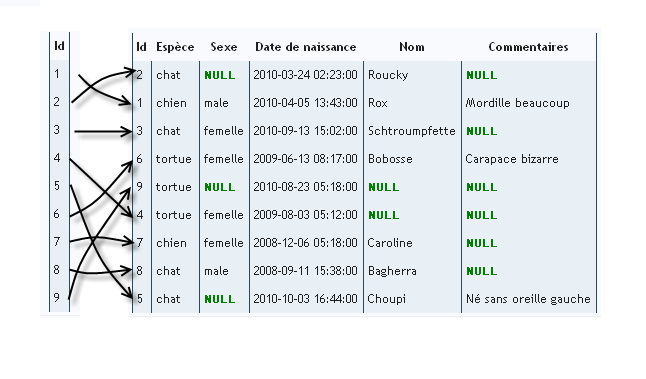
### Qu'est-ce qu'un index ?

Revoyons la définition d'un index.

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

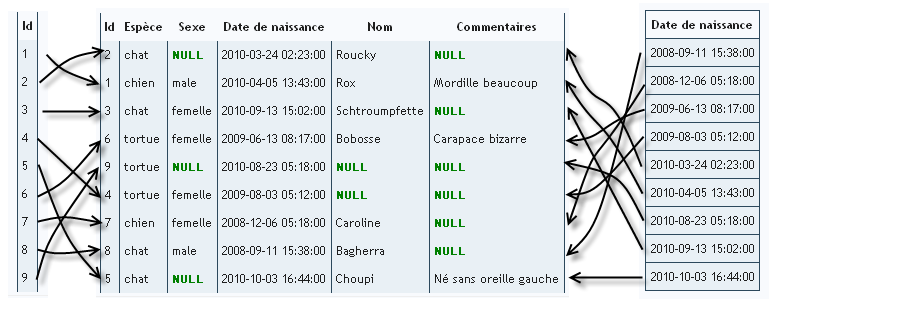
Lorsque vous créez un index sur une table, MySQL stocke cet index sous forme d'une structure particulière, contenant les valeurs des colonnes impliquées dans l'index. Cette structure stocke les valeurs **triées**et permet d'accéder à chacune de manière efficace et rapide. Voilà un petit schéma explicatif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

Pour permettre une compréhension plus facile, je représente ici l'index sous forme de table. En réalité, par défaut, MySQL stocke les index dans une structure de type "arbre" (l'index est alors de type BTREE). Le principe est cependant le même.



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 millions de lignes. Le gain de temps peut être assez considérable.  
Par ailleurs, avec les id triés par ordre croissant, pour rechercher un id particulier, MySQL n'est pas obligé de simplement parcourir les données ligne par ligne. Il peut utiliser des algorithmes de recherche puissants (comme la recherche dichotomique), toujours afin d’accélérer la recherche.

Mais pourquoi ne pas simplement trier la table complète sur la base de la colonne id ? Pourquoi créer et stocker une structure spécialement pour l'index ? Tout simplement parce qu'il peut y avoir plusieurs index sur une même table, et que l'ordre des lignes, pour chacun de ces index, n'est pas nécessairement le même. Par exemple, nous pouvons créer un second index pour notre table 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 qui utilisent des colonnes indexées comme critères de recherche. Par conséquent, si vous savez que, dans votre application, vous ferez énormément de recherches sur la colonne X, ajoutez donc un index sur cette colonne, vous ne vous en porterez que mieux.  
Les index permettent aussi d'assurer l'intégrité des données de la base. Pour cela, il existe plusieurs types d'index différents, et deux types de "clés". Lorsque je parle de garantir l'intégrité de vos données, cela signifie garantir la qualité de vos données, vous assurer que vos données ont du sens. Grâce aux clés et aux index, vous pouvez par exemple avoir la garantie que tous les clients auxquels vous faites référence dans la table Commande existent bien dans la table Client.

#### Désavantages

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

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

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

### 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 ! :p ), 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’insérerez deux fois la même valeur (ou combinaison de valeurs) dans la table.

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

Autre exemple, dans notre table Animal cette fois. Pour 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. Sans le savoir, ou sans savoir que c'était appelé comme cela, vous en avez déjà utilisé. En effet, lorsque vous empêchez une colonne d'accepter NULL, vous lui mettez une **contrainte**NOT NULL.  
De même, les valeurs par défaut que vous pouvez donner aux colonnes sont des contraintes. Vous contraignez la colonne à prendre une certaine valeur si aucune autre n'est précisée.

#### Index FULLTEXT

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

Une différence importante (très importante !!! :diable: ) 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 ! o_O ) ou KEY  et peuvent être créés de deux manières :

* soit directement lors de la création de la table ;
* soit en les ajoutant par la suite.

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

CREATE TABLE nom\_table (

colonne1 *INT* KEY, -- Crée un index simple sur colonne1

colonne2 *VARCHAR*(40) UNIQUE, -- Crée un index unique sur colonne2

);

Quelques petites remarques ici :

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

##### Liste d'index

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

CREATE TABLE nom\_table (

colonne1 description\_colonne1,

[colonne2 description\_colonne2,

colonne3 description\_colonne3,

...,]

[PRIMARY KEY (colonne\_clé\_primaire)],

[INDEX [nom\_index] (colonne1\_index [, colonne2\_index, ...])]

)

[ENGINE=moteur];

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

CREATE TABLE Animal (

id SMALLINT UNSIGNED NOT NULL AUTO\_INCREMENT,

espece *VARCHAR*(40) NOT NULL,

sexe *CHAR*(1),

date\_naissance DATETIME NOT NULL,

nom *VARCHAR*(30),

commentaires *TEXT*,

PRIMARY KEY (id),

INDEX ind\_date\_naissance (date\_naissance), -- index sur la date de naissance

INDEX ind\_nom (nom(10)) -- index sur le nom (le chiffre entre parenthèses étant le nombre de caractères pris en compte)

)

ENGINE=INNODB;

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

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

CREATE TABLE nom\_table (

colonne1 *INT* NOT NULL,

colonne2 *VARCHAR*(40),

colonne3 *TEXT*,

UNIQUE [INDEX] ind\_uni\_col2 (colonne2), -- Crée un index UNIQUE sur la colonne2, INDEX est facultatif

FULLTEXT [INDEX] ind\_full\_col3 (colonne3) -- Crée un index FULLTEXT sur la colonne3, INDEX est facultatif

)

ENGINE=MyISAM;

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

CREATE TABLE Animal (

id SMALLINT UNSIGNED NOT NULL AUTO\_INCREMENT,

espece *VARCHAR*(40) NOT NULL,

sexe *CHAR*(1),

date\_naissance DATETIME NOT NULL,

nom *VARCHAR*(30),

commentaires *TEXT*,

PRIMARY KEY (id),

INDEX ind\_date\_naissance (date\_naissance),

INDEX ind\_nom (nom(10)),

UNIQUE INDEX ind\_uni\_nom\_espece (nom, espece) -- Index sur le nom et l'espece

)

ENGINE=INNODB;

Avec cette syntaxe, KEY  ne peut être utilisé que pour les index simples et 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

ALTER TABLE nom\_table

ADD INDEX [nom\_index] (colonne\_index [, colonne2\_index ...]); --Ajout d'un index simple

ALTER TABLE nom\_table

ADD UNIQUE [nom\_index] (colonne\_index [, colonne2\_index ...]); --Ajout d'un index UNIQUE

ALTER TABLE nom\_table

ADD FULLTEXT [nom\_index] (colonne\_index [, colonne2\_index ...]); --Ajout d'un index FULLTEXT

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

Reprenons la table Test\_tuto, utilisée pour tester ALTER TABLE, et ajoutons-lui un index sur la colonne nom :

ALTER TABLE Test\_tuto

ADD INDEX ind\_nom (nom);

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

##### Ajout avec CREATE INDEX

La syntaxe de CREATE INDEX  est très simple :

CREATE INDEX nom\_index

ON nom\_table (colonne\_index [, colonne2\_index ...]); -- Crée un index simple

CREATE UNIQUE INDEX nom\_index

ON nom\_table (colonne\_index [, colonne2\_index ...]); -- Crée un index UNIQUE

CREATE FULLTEXT INDEX nom\_index

ON nom\_table (colonne\_index [, colonne2\_index ...]); -- Crée un index FULLTEXT

**Exemple :**

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

CREATE INDEX ind\_nom

ON Test\_tuto (nom);

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

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

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

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

CREATE TABLE nom\_table (

colonne1 *INT* NOT NULL,

colonne2 *VARCHAR*(40),

colonne3 *TEXT*,

CONSTRAINT [symbole\_contrainte] UNIQUE [INDEX] ind\_uni\_col2 (colonne2)

);

ALTER TABLE nom\_table

ADD CONSTRAINT [symbole\_contrainte] UNIQUE ind\_uni\_col2 (colonne2);

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

#### Suppression d'un index

Rien de bien compliqué :

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.

Les recherches via  FULLTEXT  fonctionnent plutôt pas mal, mais si vous avez énormément de données, si vous faites énormément de requêtes de recherche, ou des recherches très complexes, n'hésitez pas à vous tourner vers un outil spécialisé, un vrai moteur de recherche comme SolR, Sphinx ou ElasticSearch.

Quelques rappels d'abord :

* 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.

Pour illustrer 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).

CREATE TABLE Livre (

id *INT* UNSIGNED 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".  
Chacun de ces mots sera ensuite comparé avec les valeurs des colonnes sur lesquelles se fait la recherche. Si la colonne contient un des mots recherchés, on considère alors qu'elle correspond à la recherche.

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, about, after, once, under, the…) 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 extension 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, au fait 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  :

SELECT \* -- Vous mettez évidemment les colonnes que vous voulez.

FROM nom\_table

WHERE MATCH(colonne1[, colonne2, ...]) -- La (ou les) colonne(s) dans laquelle (ou lesquelles) on veut faire la recherche (index FULLTEXT correspondant nécessaire).

AGAINST ('chaîne recherchée'); -- La chaîne de caractères recherchée, entre guillemets bien sûr.

Si l'on veut préciser que l'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.

SELECT \*

FROM nom\_table

WHERE MATCH(colonne1[, colonne2, ...])

AGAINST ('chaîne recherchée' IN NATURAL LANGUAGE MODE);

**Premier exemple :** on recherche "Terry" dans la colonne auteur de la table Livre .

SELECT \*

FROM Livre

WHERE MATCH(auteur)

AGAINST ('Terry');

**Deuxième exemple :** on recherche d'abord "Petite", puis "Petit" dans la colonne *titre*.

SELECT \*

FROM Livre

WHERE MATCH(titre)

AGAINST ('Petite');

SELECT \*

FROM Livre

WHERE MATCH(titre)

AGAINST ('Petit');

a 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.

**Troisième exemple :** on recherche "Henri" dans la colonne *auteur*.

SELECT \*

FROM Livre

WHERE MATCH(auteur)

AGAINST ('Henri');

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

**Quatrième exemple :** on recherche "Jules", puis "Jules Verne" dans les colonnes *titre*et *auteur*.

SELECT \*

FROM Livre

WHERE MATCH(auteur, titre)

AGAINST ('Jules');

SELECT \*

FROM Livre

WHERE MATCH(titre, auteur)

AGAINST ('Jules Verne');

**Cinquième exemple :** affichage de la pertinence de la recherche.

SELECT \*, MATCH(titre, auteur) AGAINST ('Jules Verne Lune')

FROM Livre;

## Clés primaires et étrangères

Maintenant que les index n'ont plus de secrets 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 que l'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 sur la création des tables. Je vous avais alors donné la définition suivante :

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

Examinons plus en détail cette définition.

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

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

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

#### Choix de la clé primaire

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

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

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

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

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

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

##### PRIMARY KEY or not PRIMARY KEY

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, s'il vous plaît, pensez à mettre une clé primaire sur chacune de vos tables. Si l'utilité n'est pas complètement évidente pour vous pour le 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. :ange:

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

##### Lors de la création de la table

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

CREATE TABLE [IF NOT EXISTS] Nom\_table (

colonne1 description\_colonne1 PRIMARY KEY [,

colonne2 description\_colonne2,

colonne3 description\_colonne3,

...,]

)

[ENGINE=moteur];

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

CREATE TABLE Animal (

id SMALLINT AUTO\_INCREMENT PRIMARY KEY,

espece *VARCHAR*(40) NOT NULL,

sexe *CHAR*(1),

date\_naissance DATETIME NOT NULL,

nom *VARCHAR*(30),

commentaires *TEXT*

)

ENGINE=InnoDB;

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

CREATE TABLE [IF NOT EXISTS] Nom\_table (

colonne1 description\_colonne1 [,

colonne2 description\_colonne2,

colonne3 description\_colonne3,

...],

[CONSTRAINT [symbole\_contrainte]] PRIMARY KEY (colonne\_pk1 [, colonne\_pk2, ...]) -- comme pour les index UNIQUE, CONSTRAINT est facultatif

)

[ENGINE=moteur];

C'est ce que nous avions fait d'ailleurs pour la table Animal. Cette méthode permet bien sûr la création d'une clé composite (avec plusieurs colonnes).

**Exemple :** création d'Animal.

CREATE TABLE Animal (

id SMALLINT AUTO\_INCREMENT,

espece *VARCHAR*(40) NOT NULL,

sexe *CHAR*(1),

date\_naissance DATETIME NOT NULL,

nom *VARCHAR*(30),

commentaires *TEXT*,

PRIMARY KEY (id)

)

ENGINE=InnoDB;

##### Après création de la table

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

Si vous créez une clé primaire sur une table existante, assurez-vous que la ou les colonnes où vous souhaitez l'ajouter ne contiennent pas NULL.

ALTER TABLE nom\_table

ADD [CONSTRAINT [symbole\_contrainte]] PRIMARY KEY (colonne\_pk1 [, colonne\_pk2, ...]);

#### Suppression de la clé primaire

ALTER TABLE nom\_table

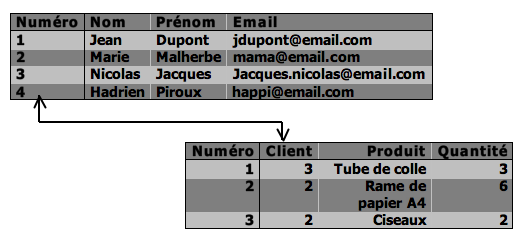
DROP PRIMARY KEY

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.



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 ? 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 sympathique 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 ?  
Ce lutin, ou plutôt cette lutine, existe ! Elle s'appelle "clé étrangère".

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

Avant d'entamer une danse de joie, parce que, quand même, le SQL, c'est génial, restez concentré cinq minutes, le temps de lire et de retenir quelques points **importants**.

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

#### Création

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

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

##### Lors de la création de la table

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

CREATE TABLE [IF NOT EXISTS] Nom\_table (

colonne1 description\_colonne1,

[colonne2 description\_colonne2,

colonne3 description\_colonne3,

...,]

[ [CONSTRAINT [symbole\_contrainte]] FOREIGN KEY (colonne(s)\_clé\_étrangère) REFERENCES table\_référence (colonne(s)\_référence)]

)

[ENGINE=moteur];

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

CREATE TABLE Commande (

numero *INT* UNSIGNED PRIMARY KEY AUTO\_INCREMENT,

client *INT* UNSIGNED NOT NULL,

produit *VARCHAR*(40),

quantite SMALLINT DEFAULT 1,

CONSTRAINT fk\_client\_numero -- On donne un nom à notre clé

FOREIGN KEY (client) -- Colonne sur laquelle on crée la clé

REFERENCES Client(numero) -- Colonne de référence

)

ENGINE=InnoDB; -- MyISAM interdit, je le rappelle encore une fois !

##### Après création de la table

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

ALTER TABLE Commande

ADD CONSTRAINT fk\_client\_numero FOREIGN KEY (client) REFERENCES Client(numero);

#### Suppression d'une clé étrangère

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

ALTER TABLE nom\_table

DROP FOREIGN KEY symbole\_contrainte

### 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 sur les jointures.

Jusqu'à maintenant, la seule information sur l'espèce des animaux de notre élevage était son nom courant. Nous voudrions maintenant stocker aussi 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'avez envisagé cette possibilité qu'une demi-seconde, car il est assez évident que c'est une très mauvaise solution. En effet, cela 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 cela aurait déjà pu poser problème. Imaginez que, pour un animal, vous fassiez une faute d'orthographe au nom de l'espèce, "chein" au 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 permet 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 commande que je vous propose d'exécuter pour créer la table Espece :

CREATE TABLE Espece (

id SMALLINT UNSIGNED AUTO\_INCREMENT,

nom\_courant *VARCHAR*(40) NOT NULL,

nom\_latin *VARCHAR*(40) NOT NULL UNIQUE,

description *TEXT*,

PRIMARY KEY(id)

)

ENGINE=InnoDB;

On met un index UNIQUE  sur la colonne nom\_latin pour être sûr que l'on ne rentrera pas deux fois la même espèce. Pourquoi sur le nom latin et pas sur le nom courant ? Tout simplement parce que le nom latin est beaucoup plus rigoureux et réglementé 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 :

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');

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 cela 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

Je vous conseille d'essayer d'écrire vous-même les requêtes avant de regarder comment je fais.

-- Ajout d'une colonne espece\_id

ALTER TABLE Animal ADD COLUMN espece\_id SMALLINT UNSIGNED; -- même type que la colonne id de Espece

-- Remplissage de espece\_id

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';

-- Suppression de la colonne espece

ALTER TABLE Animal DROP COLUMN espece;

-- Ajout de la clé étrangère

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 (qui n'existe donc pas dans la table Espece) :

INSERT INTO Animal (nom, espece\_id, date\_naissance)

VALUES ('Caouette', 5, '2009-02-15 12:45:00');

ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails (`elevage`.`animal`, CONSTRAINT `fk\_espece\_id` FOREIGN KEY (`espece\_id`) REFERENCES `espece` (`id`))

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 l'être ! Ajoutons également l'index UNIQUE  sur (nom, espece\_id) dont on a déjà discuté.

ALTER TABLE Animal MODIFY espece\_id SMALLINT UNSIGNED NOT NULL;

CREATE UNIQUE INDEX ind\_uni\_nom\_espece\_id ON Animal (nom, espece\_id);

Notez qu'il n'était pas possible de mettre la contrainte NOT NULL  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 également ajouter une colonne à la table Animal, qui contiendra l'id de la race de l'animal. Contrairement à la colonne espece\_id, celle-ci pourra être NULL. Il n'est pas impossible que nous accueillions des bâtards, ou que certaines espèces que nous élevons ne soient pas classées en plusieurs races différentes.  
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, qui contiendront respectivement l'id du père et l'id de la mère de l'animal.

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

-- --------------------------

-- CREATION DE LA TABLE Race

-- --------------------------

CREATE TABLE Race (

id SMALLINT UNSIGNED AUTO\_INCREMENT,

nom *VARCHAR*(40) NOT NULL,

espece\_id SMALLINT UNSIGNED NOT NULL, -- 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

)

ENGINE = InnoDB;

-- -----------------------

-- 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 UNSIGNED;

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 (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 UNSIGNED;

ALTER TABLE Animal

ADD CONSTRAINT fk\_mere\_id FOREIGN KEY (mere\_id) REFERENCES Animal(id);

ALTER TABLE Animal ADD COLUMN pere\_id SMALLINT UNSIGNED;

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 ce 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.

## Jointures

Vous vous attaquez maintenant au plus important chapitre de cette partie. Le principe des jointures est plutôt simple à comprendre (quoique l'on puisse faire des requêtes très compliquées avec les jointures), et 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 cela 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, si vous voulez afficher la description de l'espèce de Cartouche (votre petit préféré), vous avez besoin de deux requêtes

**Étape 1 :** on trouve l'*id* de l'espèce de Cartouche grâce à la table *Animal*.

SELECT espece\_id FROM Animal WHERE nom = 'Cartouche';

**Étape 2 :** on trouve la description de l'espèce grâce à son *id*.

SELECT description FROM Espece WHERE id = 1;

Ne serait-ce pas merveilleux de pouvoir faire tout cela (et plus encore) en une seule requête ?  
C'est là que les jointures entrent en jeu ; on va 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*.

SELECT Espece.description

FROM Espece

INNER JOIN Animal

ON Espece.id = Animal.espece\_id

WHERE Animal.nom = 'Cartouche';

### Jointure interne

L'air de rien, dans l'introduction, je vous ai déjà montré comment faire une jointure. La première émotion passée, vous devriez vous être dit : "Tiens, mais cela n'a pas l'air bien compliqué, en fait, les jointures."  
En effet, une fois que vous aurez compris comment réfléchir aux jointures, tout se fera tout seul. Personnellement, cela 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 une jointure **interne**.

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.

Si vous avez compris ça, vous avez tout compris !

#### Syntaxe

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

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 préciser 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 :

SELECT \*

FROM table1

INNER JOIN table2

ON table1.colonneA = table2.colonneJ

AND table1.colonneT = table2.colonneX

[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'avez vu dans notre requête, on utilise pour cela l'opérateur **.**(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 la table quand il s'agit de grosses requêtes avec plusieurs jointures. En revanche, pour les petites jointures courantes, il est vrai que c'est moins long à écrire si l'on ne précise pas la table.

**Exemple :** sélection du nom des animaux commençant par "Ch", ainsi que de l'id et la description de leur espèce.

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 'Ch%';

##### Utiliser les alias

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

**Exemple :** on renomme la table Espece "e", et la table Animal "a".

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 'Ch%';

Comme vous le voyez, le code est plus compact. Ici encore, 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 ainsi plus facile de s'y retrouver.

Une autre utilité des alias est de renommer les colonnes pour que le résultat soit plus clair. Observez le résultat de la requête précédente. Vous avez trois colonnes : id, description et nom. Le nom de la table dont provient la colonne n'est indiqué nulle part. A priori, vous savez ce que vous avez demandé. 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 l'animal, de son maître, du père, du fils ou du Saint-Esprit !  
Il est intéressant là aussi d'utiliser les alias.

**Exemple :** on donne des alias aux colonnes (id\_espece pour id de la table Espece, description\_espece pour Espece.description et nom\_bestiole pour Animal.nom).

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 'Ch%';

#### Pourquoi "interne" ?

INNER JOIN  permet donc de faire une jointure **interne** sur deux tables. Mais que signifie donc ce "interne" ?

C'est très simple ! Lorsque l'on fait une jointure interne, cela veut dire que l'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 :

SELECT \*

FROM A

INNER JOIN B

ON A.a = B.b

Ceci retournera **uniquement** les lignes pour lesquelles A.a et B.b correspondent.

**Exemple :** on veut connaître la race des chats.

SELECT Animal.nom AS nom\_animal, Race.nom AS race

FROM Animal

INNER JOIN Race

ON Animal.race\_id = Race.id

WHERE Animal.espece\_id = 2 -- ceci correspond aux chats

ORDER BY Race.nom, Animal.nom;

### 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.

Si l'on veut de nouveau connaître la race des chats, mais que cette fois-ci nous voulons également afficher les chats qui n'ont pas de race, on peut utiliser la jointure suivante (je ne prends que les chats dont le nom commence par "C" afin de réduire le nombre de lignes, mais vous pouvez choisir les conditions que vous voulez) :

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.espece\_id = 2

AND 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.espece\_id = 2

AND Animal.nom LIKE 'C%'

ORDER BY Race.nom, Animal.nom;

On ne connaît pas la race de Choupi, et pourtant il fait bien partie des lignes sélectionnées, alors qu'avec la jointure interne, il n'apparaissait pas. Simplement, les colonnes qui viennent de la table Race (la colonne Race.nom AS race  dans ce cas-ci) sont NULL  pour les lignes qui n'ont pas de correspondance (la ligne de Choupi ici).

#### 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.

**Exemple :** toujours avec les races de chats.

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;

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). Voici l'équivalent avec une jointure par la gauche de la requête que nous venons d'écrire :

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.

SELECT \*

FROM table1

[INNER | LEFT | RIGHT] JOIN table2 USING (colonneJ); -- colonneJ est présente dans les deux tables

-- équivalent 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 naturelles

Comme pour les jointures avec USING, il est possible d'utiliser les jointures naturelles dans le cas où les colonnes servant à la jointure ont **le même nom** dans les deux tables.  
Simplement, dans le cas d'une jointure naturelle, on ne donne pas la ou les colonnes sur lesquelles joindre les tables : c'est déterminé automatiquement. Donc si l'on a les trois tables suivantes :

* table1 : colonnes A, B, C
* table2 : colonnes B, E, F
* table3 : colonnes A, C, E

**Exemple 1** : jointure de table1 et table2 (une colonne ayant le même nom : B).

SELECT \*

FROM table1

NATURAL JOIN table2;

-- EST ÉQUIVALENT À

SELECT \*

FROM table1

INNER JOIN table2

ON table1.B = table2.B;

**Exemple 2 :** jointure de table1 et table3 (deux colonnes ayant le même nom : A et C).

SELECT \*

FROM table1

NATURAL JOIN table3;

-- EST ÉQUIVALENT À

SELECT \*

FROM table1

INNER JOIN table3

ON table1.A = table3.A AND table1.C = table3.C;

Pour utiliser ce type de jointure, il faut donc être certain que **toutes les colonnes nécessaires** à la jointure ont le même nom dans les deux tables, mais aussi que les colonnes ayant le même nom sont **uniquement celles qui servent à la jointure**.

Notez que vous pouvez également réaliser une jointure externe par la gauche avec les jointures naturelles à l'aide de NATURAL LEFT JOIN.

#### 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  (sans clause ON).

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’œil 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 de 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 œil à la solution. Le but ici est de vous faire réfléchir, et surtout de vous faire prendre conscience que l'on peut faire pas mal de choses en SQL, en combinant plusieurs techniques par exemple.

Pour jouer le jeu jusqu'au bout, je vous propose de 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; . Il est bien plus utile de faire une jointure avec la table Espece.

#### A/ Commençons par des choses faciles

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.

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%';

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 leur pelage.**

Dans la description des races, j'utilise parfois "pelage", parfois "poil", et parfois "robe".

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;

Il faut donc :

* soit que l'on ne connaisse pas la race (on n'a alors aucune information sur le pelage de la race, a fortiori) ;
* soit que l'on connaisse la race, mais que, dans la description de celle-ci, il n'y ait pas d'information sur le pelage.

**1/ On ne connaît pas la race**

Les animaux dont on ne connaît 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 que c'est fait, pour les sélectionner, il suffit de mettre la condition Animal.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'information 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 l'on met tout cela ensemble : il fallait faire une jointure externe des tables Animal et Race, et ensuite sélectionner les animaux qui répondaient à l'une ou l'autre des conditions (opérateur logique OR).

#### B/ Compliquons un peu les choses

Jointures sur deux tables, ou plus !

##### 1. La race ET l'espèce

**Vous devez obtenir la liste des chats et des perroquets amazones, avec leur sexe, leur espèce (nom latin) et leur race s'ils en ont une. Regroupez les chats ensemble, les perroquets ensemble et, au sein de l'espèce, regroupez les races.**

SELECT Animal.nom as nom\_animal, Animal.sexe, Espece.nom\_latin as espece, Race.nom as race

FROM Animal

INNER JOIN Espece

ON Animal.espece\_id = Espece.id

LEFT JOIN Race

ON Animal.race\_id = Race.id

WHERE Espece.nom\_courant IN ('Perroquet amazone', 'Chat')

ORDER BY Espece.nom\_latin, Race.nom;

Comme vous voyez, c'est non seulement très simple de faire des jointures sur plus d'une table, mais c'est également possible de mélanger jointures internes et externes. Si cela vous pose problème, essayez vraiment de vous imaginer les étapes.  
D'abord, on fait la jointure d'Animal et d'Espece. On se retrouve alors avec une grosse table qui possède toutes les colonnes d'Animal et toutes les colonnes d'Espece. Ensuite, 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 de la race. L'important ici était de trier d'abord sur une colonne d'Espece, ensuite sur une colonne de Race.

##### 2. Futures génitrices

**Vous devez obtenir la liste des chiennes dont on connaît la race, et qui sont en âge de procréer (c'est-à-dire nées avant juillet 2016). Affichez leurs nom, date de naissance et race.**

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 = 'F';

Cette fois, il fallait faire une jointure interne avec Race, puisque l'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 sœurs (Wohooooo)

**Vous devez obtenir la liste des chats dont on connaît les parents, ainsi que le nom de ces parents.**

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';

Si celle-là, vous l'avez trouvée 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 d'Animal dans FROM  des colonnes d'Animal dans JOIN  ? Vous savez maintenant qu'il suffit d'utiliser des alias.  
Il faut faire une jointure sur trois tables, puisque finalement, 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 au 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).**

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';

Après la requête précédente, celle-ci devrait vous sembler plutôt facile ! Notez qu'il y a plusieurs manières de faire, bien sûr. En voici une autre :

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 ! Évidemment, si vous avez utilisé 45 jointures et 74 conditions, alors ce n'est pas bon non plus. Du moment que vous n'avez joint que deux tables, cela devrait être bon.

##### 3. C'est un pure race ?

Courage, c'est la dernière (et la plus trash :ninja: )!

**Vous devez obtenir la liste des animaux dont on connaît 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).**

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;

Pfiou :euh: ! Le principe est exactement le même que pour avoir le nom des parents. Il suffit de 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.

## 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 le 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 seront traitées dans le prochain chapitre.

La plupart des requêtes de sélection que vous allez voir dans ce chapitre sont tout à fait réalisables 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 en général **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.

**Exemple :** on sélectionne toutes les femelles parmi les perroquets et les tortues .

SELECT Animal.id, Animal.sexe, Animal.date\_naissance, Animal.nom, Animal.espece\_id

FROM Animal

INNER JOIN Espece

ON Espece.id = Animal.espece\_id

WHERE sexe = 'F'

AND Espece.nom\_courant IN ('Tortue d''Hermann', 'Perroquet amazone');

Parmi ces femelles perroquets et tortues, on veut connaître la date de naissance de la plus âgée. On va donc faire une sélection dans la table des résultats de la requête.

SELECT MIN(date\_naissance)

FROM (

SELECT Animal.id, Animal.sexe, Animal.date\_naissance, Animal.nom, Animal.espece\_id

FROM Animal

INNER JOIN Espece

ON Espece.id = Animal.espece\_id

WHERE sexe = 'F'

AND Espece.nom\_courant IN ('Tortue d''Hermann', 'Perroquet amazone')

) AS tortues\_perroquets\_F;

##### 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, on l'a appelée tortues\_perroquets\_F.  
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 au fait qu'il ne s'agit pas de la table Animal, mais bien d'une table tirée d'Animal.  
Par conséquent, si vous voulez préciser le nom de la table dans le SELECT  principal, vous devez écrire SELECT MIN(tortues\_perroquets\_F.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 :

SELECT MIN(date\_naissance)

FROM (

SELECT Animal.id, Animal.nom

FROM Animal

INNER JOIN Espece

ON Espece.id = Animal.espece\_id

WHERE sexe = 'F'

AND Espece.nom\_courant IN ('Tortue d''Hermann', 'Perroquet amazone')

) AS tortues\_perroquets\_F;

En effet, tortues\_perroquets\_F 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 noms de colonnes ambigus. Une table, même intermédiaire, ne peut pas avoir deux colonnes ayant le même nom. Si deux colonnes ont le même nom, il est nécessaire de renommer explicitement au moins l'une des deux.

Donc, si l'on veut sélectionner la colonne Espece.id en plus dans la sous-requête, on peut procéder ainsi :

SELECT MIN(date\_naissance)

FROM (

SELECT Animal.id, Animal.sexe, Animal.date\_naissance, Animal.nom, Animal.espece\_id,

Espece.id AS espece\_espece\_id -- On renomme la colonne id de Espece, donc il n'y a plus de doublons.

FROM Animal -- Attention de ne pas la renommer espece\_id, puisqu'on sélectionne aussi la colonne espece\_id dans Animal !

INNER JOIN Espece

ON Espece.id = Animal.espece\_id

WHERE sexe = 'F'

AND Espece.nom\_courant IN ('Tortue d''Hermann', 'Perroquet amazone')

) AS tortues\_perroquets\_F;

### 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 ;
* plusieurs colonnes, mais une seule ligne ;
* 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.

##### Sous-requête renvoyant une valeur

Le cas le plus simple est évidemment d'utiliser une sous-requête qui renvoie une valeur.

SELECT id, sexe, nom, commentaires, espece\_id, race\_id

FROM Animal

WHERE race\_id =

(SELECT id FROM Race WHERE nom = 'Berger Allemand'); -- la sous-requête renvoie simplement 1

Remarquez que cette requête peut également s'écrire avec une jointure plutôt qu'une sous-requête :

SELECT Animal.id, sexe, Animal.nom, commentaires, Animal.espece\_id, race\_id

FROM Animal

INNER JOIN Race ON Race.id = Animal.race\_id

WHERE Race.nom = 'Berger Allemand';

Voici un exemple de requête avec sous-requête qu'il est impossible de faire avec une simple jointure :

SELECT id, nom, espece\_id

FROM Race

WHERE espece\_id = (

SELECT MIN(id) FROM Espece

-- Je rappelle que MIN() permet de récupérer la plus petite valeur de la colonne parmi les lignes sélectionnées

);

En ce qui concerne les autres opérateurs de comparaison, le principe est exactement le même :

SELECT id, nom, espece\_id

FROM Race

WHERE espece\_id < (

SELECT id

FROM Espece

WHERE nom\_courant = 'Tortue d''Hermann');

##### 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 :

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 !

SELECT id, sexe, nom, espece\_id, race\_id

FROM Animal

WHERE (id, race\_id) = (

SELECT id, espece\_id

FROM Race

WHERE id = 7);

Décomposons calmement. Voyons d'abord ce que la sous-requête donne comme résultat.

SELECT id, espece\_id

FROM Race

WHERE id = 7;

#### Conditions avec IN et NOT IN

##### IN

Vous connaissez déjà l'opérateur IN, qui compare une colonne avec une liste de valeurs.

**Exemple**

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 :

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 :

SELECT id -- On ne sélectionne bien qu'UNE SEULE COLONNE.

FROM Espece

WHERE nom\_courant IN ('Tortue d''Hermann', 'Perroquet amazone');

Ainsi, la requête principale sélectionnera les lignes qui ont un espece\_id parmi ceux renvoyés 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.

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, plus petit, 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 inférieur 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).

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, cela signifiera "Sélectionne les lignes de la table Animal, dont l'espece\_id est inférieur 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.

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 : 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 :

SELECT id

FROM Espece

WHERE nom\_courant = ANY ('Tortue d''Hermann', 'Perroquet amazone');

#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 :

SELECT colonne1

FROM tableA

WHERE colonne2 IN (

SELECT colonne3

FROM tableB

WHERE tableB.colonne4 = tableA.colonne5

);

Si l'on prend la sous-requête toute seule, on ne pourra pas l'exécuter :

SELECT colonne3

FROM tableB

WHERE tableB.colonne4 = tableA.colonne5

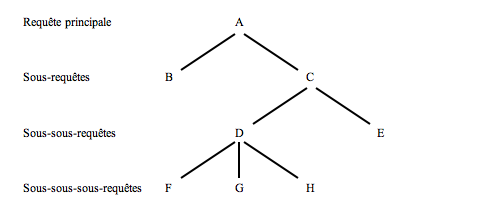
En effet, seule la tableB est sélectionnée dans la clause FROM, il n'y a pas de jointure avec la tableA, et pourtant, on utilise la tableA dans la condition.

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 :** si MySQL rencontre une table inconnue dans une sous-requête, il va aller chercher dans les **niveaux supérieurs uniquement** si cette table existe. Donc, imaginons que l'on a une requête avec 3 niveaux : la requête principale (niveau 1), une ou plusieurs sous-requêtes (niveau 2) et une ou plusieurs sous-sous-requêtes, c'est-à-dire une sous-requête dans une sous-requête (niveau 3).

* Une sous-requête (niveau 2) peut être corrélée à la requête principale (niveau 1).
* Une sous-sous-requête (niveau 3) peut être corrélée à la sous-requête **dont elle dépend** (niveau 2), ou à la requête principale (niveau 1).
* Mais une sous-requête (niveau 2) ne peut pas être corrélée à une autre sous-requête (niveau 2).

Si l'on prend le schéma suivant, on peut donc remonter l'arbre, mais jamais descendre d'un cran pour trouver les tables nécessaires.

Sous-requêtes corrélées

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.

Le temps de vous expliquer le fonctionnement de EXISTS  et NOT EXISTS, et nous verrons un exemple de sous-requête corrélée.

##### Conditions avec EXISTS et NOT EXISTS

Les conditions EXISTS  et NOT EXISTS  s'utilisent de la manière suivante :

SELECT \* FROM nom\_table

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.

**Exemple :** on sélectionne les races s'il existe un animal qui s'appelle Balou.

SELECT id, nom, espece\_id FROM Race

WHERE EXISTS (SELECT \* FROM Animal WHERE nom = 'Balou');

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.

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.

**Exemple :** je veux sélectionner toutes les races dont on ne possède aucun animal.

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.

En résultat, on a bien le Sphynx, puisque l'on n'en possède aucun.

### Modification

#### Utilisation des sous-requêtes

##### Pour la sélection

Imaginez que, pour une raison bizarre, vous vouliez que tous les perroquets aient en commentaire : "Coco veut un gâteau !"

Si vous saviez que l'id de l'espèce est 4, ce serait facile :

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 !

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 ! À 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.

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. Pour cela, vous avez besoin de l'id de la race nebelung que vous venez d'ajouter.  
Vous vous en doutez, il est tout à fait possible de le faire grâce à une sous-requête :

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 que l'**on ne peut pas modifier un élément d'une table que l'on utilise dans une sous-requête**.

**Exemple :** vous trouvez que Callune ressemble quand même fichtrement à Cawette, et ses poils sont aussi longs. Vous vous dites donc que vous auriez dû également modifier la race de Callune. Vous essayez donc la requête suivante :

UPDATE Animal *SET* race\_id =

(SELECT race\_id FROM Animal WHERE nom = 'Cawette' AND espece\_id = 2)

WHERE nom = 'Callune';

Malheureusement :

ERROR 1093 (HY000): You can't specify target table 'Animal' for update in FROM clause

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 :

UPDATE Animal -- Classique !

INNER JOIN Espece -- Jointure.

ON Animal.espece\_id = Espece.id

-- Condition de la jointure.

*SET* Animal.commentaires = Espece.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 limité aux jointures internes ni à deux tables jointes.

### Suppression

Cette partie sera relativement courte, puisque l'utilisation des sous-requêtes et des jointures est assez ressemblante concernant la suppression et 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 :

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 !

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 pas supprimer des 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 :

DELETE Animal -- Je précise de quelles tables les données doivent être supprimées

FROM Animal -- Table principale

INNER JOIN Espece ON Animal.espece\_id = Espece.id

-- Jointure

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 d'Animal correspondantes.

## 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 comment cela fonctionne !

### Syntaxe

La syntaxe d'UNION  est simplissime : si vous avez deux requêtes SELECT  dont vous voulez additionner les résultats, il vous suffit d'ajouter UNION  entre ces deux requêtes.

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.

SELECT ...

UNION

SELECT ...

UNION

SELECT ...

....

UNION

SELECT ...

Par exemple, vous pouvez obtenir les chats et les tortues de la manière suivante :

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';

## 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 maintenant examiner ces options.

### Option sur suppression des clés étrangères

#### Petits rappels

##### La syntaxe

Voici comment on ajoute une clé étrangère à une table déjà existante :

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 un 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 précisément la raison d'être de notre clé étrangère.

Essayons :

DELETE FROM Race WHERE nom = 'Boxer';

ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint fails (`elevage`.`animal`, CONSTRAINT `fk\_race\_id` FOREIGN KEY (`race\_id`) REFERENCES `race` (`id`))

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 utilisé par certaines lignes de la table Animal).

Mais cela veut donc dire que, chaque fois que l'on veut supprimer des lignes de la table Race, il faut d'abord supprimer toutes les références à ces races. Dans notre base, cela 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 serait-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 :

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 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 l'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, cela 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 :

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 avez attribué cette race auparavant auront NULL  comme race\_id.

Vérifions en supprimant les boxers, depuis le temps que l'on essaye !

-- 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

ORDER BY race;

-- 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

ORDER BY race;

Les ex-boxers existent toujours dans la table Animal, mais ils n'appartiennent plus à aucune race.

##### CASCADE

Ce dernier comportement est le plus risqué (et le plus violent ! :pirate: ). En effet, cela supprime purement et simplement toutes les lignes qui référençaient la valeur supprimée !  
Donc, si l'on choisit ce comportement pour la clé étrangère sur la colonne espece\_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 sujet soient également supprimés. Plutôt pratique non ?

Je le 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'ancien 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.

**Exemple :** essayons de modifier l'id de la race "Singapura".

UPDATE Race *SET* id = 3 WHERE nom = 'Singapura';

ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint fails (`elevage`.`animal`, CONSTRAINT `fk\_race\_id` FOREIGN KEY (`race\_id`) REFERENCES `race` (`id`) ON DELETE SET NULL)

L'option permettant de définir le comportement en cas de modification est donc ON UPDATE {RESTRICT | NO ACTION | SET NULL | CASCADE}. Les quatre comportements 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 ! :soleil:

Modifions par exemple la clé étrangère sur Animal.race\_id, avant de modifier l'id de la race "Singapura" (jetez d'abord un œil aux données des tables Race et Animal, afin de voir les différences).

-- 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 ! :magicien:

En règle générale (dans 99,99 % des cas), c’est une très très mauvaise idée de changer la valeur d’un id (ou de votre clé primaire auto-incrémentée). En effet, vous risquez des problèmes avec l’auto-incrément, si vous donnez une valeur non 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 pourriez vous amuser à en changer la valeur autant de fois que vous le voudriez.

Modifions une dernière fois cette clé étrangère pour remettre l'option ON UPDATE  par défaut.

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;

### 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 laisser le comportement par défaut (RESTRICT) pour toutes nos clés étrangères. Les colonnes référencées sont chaque fois des colonnes auto-incrémentées, on ne devrait donc pas modifier leurs valeurs.

#### 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 parce que l'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 paraît 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 supprimer 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. Pas de risque de fausse manœuvre donc.  
Attention au fait 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.  
Pas de changement pour cette clé étrangère !

Il s'agit de mon point de vue personnel, bien sûr. Si vous pensez que c'est mieux de mettreON DELETE CASCADE, faites-le. On peut certainement trouver des arguments en faveur des deux possibilités.

##### Clés sur Animal.mere\_id et Animal.pere\_id

Ce n'est pas parce que l'on supprime un animal que tous ses enfants 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 (puisque l'on a laissé ON DELETE RESTRICT  pour la clé sur Animal.espece\_id), 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.

-- 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 Animal(id) ON DELETE *SET* NULL;

-- 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 Animal(id) ON DELETE *SET* NULL;

-- 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;

## 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 !

**État actuel de la base de données**  
**Note :** les tables de test ne sont pas reprises

*SET* NAMES utf8;

DROP TABLE IF EXISTS Animal;

DROP TABLE IF EXISTS Race;

DROP TABLE IF EXISTS Espece;

CREATE TABLE Espece (

id smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

nom\_courant *varchar*(40) NOT NULL,

nom\_latin *varchar*(40) NOT NULL,

description *text*,

PRIMARY KEY (id),

UNIQUE KEY nom\_latin (nom\_latin)

) ENGINE=InnoDB AUTO\_INCREMENT=7 DEFAULT CHARSET=latin1;

LOCK TABLES Espece WRITE;

INSERT INTO Espece VALUES (1,'Chien','Canis canis','Bestiole à quatre pattes qui aime les caresses et tire souvent la langue'),(2,'Chat','Felis silvestris','Bestiole à quatre pattes qui saute très haut et grimpe aux arbres'),(3,'Tortue d''Hermann','Testudo hermanni','Bestiole avec une carapace très dure'),

(4,'Perroquet amazone','Alipiopsitta xanthops','Joli oiseau parleur vert et jaune');

UNLOCK TABLES;

CREATE TABLE Race (

id smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

nom *varchar*(40) NOT NULL,

espece\_id smallint(6) unsigned NOT NULL,

description *text*,

PRIMARY KEY (id)

) ENGINE=InnoDB AUTO\_INCREMENT=9 DEFAULT CHARSET=latin1;

LOCK TABLES Race WRITE;

INSERT INTO Race VALUES (1,'Berger allemand',1,'Chien sportif et élégant au pelage dense, noir-marron-fauve, noir ou gris.'),(2,'Berger blanc suisse',1,'Petit chien au corps compact, avec des pattes courtes mais bien proportionnées et au pelage tricolore ou bicolore.'),

(3,'Singapura',2,'Chat de petite taille aux grands yeux en amandes.'),(4,'Bleu russe',2,'Chat aux yeux verts et à la robe épaisse et argentée.'),(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...');

UNLOCK TABLES;

CREATE TABLE Animal (

id smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

sexe *char*(1) DEFAULT NULL,

date\_naissance datetime NOT NULL,

nom *varchar*(30) DEFAULT NULL,

commentaires *text*,

espece\_id smallint(6) unsigned NOT NULL,

race\_id smallint(6) unsigned DEFAULT NULL,

mere\_id smallint(6) unsigned DEFAULT NULL,

pere\_id smallint(6) unsigned DEFAULT NULL,

PRIMARY KEY (id),

UNIQUE KEY ind\_uni\_nom\_espece\_id (nom,espece\_id)

) ENGINE=InnoDB AUTO\_INCREMENT=65 DEFAULT CHARSET=utf8;

LOCK TABLES Animal WRITE;

INSERT INTO Animal VALUES (1,'M','2010-04-05 13:43:00','Rox','Mordille beaucoup',1,1,18,22),

(2,NULL,'2010-03-24 02:23:00','Roucky',NULL,2,NULL,40,30),(3,'F','2010-09-13 15:02:00','Schtroumpfette',NULL,2,4,41,31),(4,'F','2009-08-03 05:12:00',NULL,'Bestiole avec une carapace très dure',3,NULL,NULL,NULL),

(5,NULL,'2010-10-03 16:44:00','Choupi','Né sans oreille gauche',2,NULL,NULL,NULL),(6,'F','2009-06-13 08:17:00','Bobosse','Carapace bizarre',3,NULL,NULL,NULL),(7,'F','2008-12-06 05:18:00','Caroline',NULL,1,2,NULL,NULL),

(8,'M','2008-09-11 15:38:00','Bagherra',NULL,2,5,NULL,NULL),(9,NULL,'2010-08-23 05:18:00',NULL,'Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(10,'M','2010-07-21 15:41:00','Bobo',NULL,1,NULL,7,21),

(11,'F','2008-02-20 15:45:00','Canaille',NULL,1,NULL,NULL,NULL),(12,'F','2009-05-26 08:54:00','Cali',NULL,1,2,NULL,NULL),(13,'F','2007-04-24 12:54:00','Rouquine',NULL,1,1,NULL,NULL),

(14,'F','2009-05-26 08:56:00','Fila',NULL,1,2,NULL,NULL),(15,'F','2008-02-20 15:47:00','Anya',NULL,1,NULL,NULL,NULL),(16,'F','2009-05-26 08:50:00','Louya',NULL,1,NULL,NULL,NULL),

(17,'F','2008-03-10 13:45:00','Welva',NULL,1,NULL,NULL,NULL),(18,'F','2007-04-24 12:59:00','Zira',NULL,1,1,NULL,NULL),(19,'F','2009-05-26 09:02:00','Java',NULL,1,2,NULL,NULL),

(20,'M','2007-04-24 12:45:00','Balou',NULL,1,1,NULL,NULL),(21,'F','2008-03-10 13:43:00','Pataude',NULL,1,NULL,NULL,NULL),(22,'M','2007-04-24 12:42:00','Bouli',NULL,1,1,NULL,NULL),

(24,'M','2007-04-12 05:23:00','Cartouche',NULL,1,NULL,NULL,NULL),(25,'M','2006-05-14 15:50:00','Zambo',NULL,1,1,NULL,NULL),(26,'M','2006-05-14 15:48:00','Samba',NULL,1,1,NULL,NULL),

(27,'M','2008-03-10 13:40:00','Moka',NULL,1,NULL,NULL,NULL),(28,'M','2006-05-14 15:40:00','Pilou',NULL,1,1,NULL,NULL),(29,'M','2009-05-14 06:30:00','Fiero',NULL,2,3,NULL,NULL),

(30,'M','2007-03-12 12:05:00','Zonko',NULL,2,5,NULL,NULL),(31,'M','2008-02-20 15:45:00','Filou',NULL,2,4,NULL,NULL),(32,'M','2009-07-26 11:52:00','Spoutnik',NULL,3,NULL,52,NULL),

(33,'M','2006-05-19 16:17:00','Caribou',NULL,2,4,NULL,NULL),(34,'M','2008-04-20 03:22:00','Capou',NULL,2,5,NULL,NULL),(35,'M','2006-05-19 16:56:00','Raccou','Pas de queue depuis la naissance',2,4,NULL,NULL),

(36,'M','2009-05-14 06:42:00','Boucan',NULL,2,3,NULL,NULL),(37,'F','2006-05-19 16:06:00','Callune',NULL,2,8,NULL,NULL),(38,'F','2009-05-14 06:45:00','Boule',NULL,2,3,NULL,NULL),

(39,'F','2008-04-20 03:26:00','Zara',NULL,2,5,NULL,NULL),(40,'F','2007-03-12 12:00:00','Milla',NULL,2,5,NULL,NULL),(41,'F','2006-05-19 15:59:00','Feta',NULL,2,4,NULL,NULL),

(42,'F','2008-04-20 03:20:00','Bilba','Sourde de l''oreille droite à 80%',2,5,NULL,NULL),(43,'F','2007-03-12 11:54:00','Cracotte',NULL,2,5,NULL,NULL),(44,'F','2006-05-19 16:16:00','Cawette',NULL,2,8,NULL,NULL),

(45,'F','2007-04-01 18:17:00','Nikki','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(46,'F','2009-03-24 08:23:00','Tortilla','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(47,'F','2009-03-26 01:24:00','Scroupy','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),

(48,'F','2006-03-15 14:56:00','Lulla','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(49,'F','2008-03-15 12:02:00','Dana','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(50,'F','2009-05-25 19:57:00','Cheli','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),

(51,'F','2007-04-01 03:54:00','Chicaca','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(52,'F','2006-03-15 14:26:00','Redbul','Insomniaque',3,NULL,NULL,NULL),(54,'M','2008-03-16 08:20:00','Bubulle','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),

(55,'M','2008-03-15 18:45:00','Relou','Surpoids',3,NULL,NULL,NULL),(56,'M','2009-05-25 18:54:00','Bulbizard','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(57,'M','2007-03-04 19:36:00','Safran','Coco veut un gâteau !',4,NULL,NULL,NULL),

(58,'M','2008-02-20 02:50:00','Gingko','Coco veut un gâteau !',4,NULL,NULL,NULL),(59,'M','2009-03-26 08:28:00','Bavard','Coco veut un gâteau !',4,NULL,NULL,NULL),(60,'F','2009-03-26 07:55:00','Parlotte','Coco veut un gâteau !',4,NULL,NULL,NULL),

(61,'M','2010-11-09 00:00:00','Yoda',NULL,2,5,NULL,NULL);

UNLOCK TABLES;

ALTER TABLE Race ADD CONSTRAINT fk\_race\_espece\_id FOREIGN KEY (espece\_id) REFERENCES Espece (id) ON DELETE CASCADE;

ALTER TABLE Animal ADD CONSTRAINT fk\_pere\_id FOREIGN KEY (pere\_id) REFERENCES Animal (id) ON DELETE *SET* NULL;

ALTER TABLE Animal ADD CONSTRAINT fk\_espece\_id FOREIGN KEY (espece\_id) REFERENCES Espece (id);

ALTER TABLE Animal ADD CONSTRAINT fk\_mere\_id FOREIGN KEY (mere\_id) REFERENCES Animal (id) ON DELETE *SET* NULL;

ALTER TABLE Animal ADD CONSTRAINT fk\_race\_id FOREIGN KEY (race\_id) REFERENCES Race (id) ON DELETE *SET* NULL;

##### Modification de notre base de 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 différents id. Or, il n'est pas très intéressant (et cela peut même être dangereux) de manipuler les id.  
Par conséquent, nous allons ajouter une colonne prix à nos tables Espece et Race, qui contiendra donc le prix à payer pour adopter un animal de telle espèce ou telle race. Cette colonne sera du type DECIMAL, avec deux chiffres après la virgule. Elle ne pourra pas contenir de nombres négatifs, et sera donc également UNSIGNED.

Voici donc les commandes à exécuter :

ALTER TABLE Race

ADD COLUMN prix *DECIMAL*(7,2) UNSIGNED;

ALTER TABLE Espece

ADD COLUMN prix *DECIMAL*(7,2) UNSIGNED;

-- Remplissage des colonnes "prix"

UPDATE Espece *SET* prix = 200 WHERE id = 1;

UPDATE Espece *SET* prix = 150 WHERE id = 2;

UPDATE Espece *SET* prix = 140 WHERE id = 3;

UPDATE Espece *SET* prix = 700 WHERE id = 4;

UPDATE Espece *SET* prix = 10 WHERE id = 5;

UPDATE Espece *SET* prix = 75 WHERE id = 6;

UPDATE Race *SET* prix = 450 WHERE id = 1;

UPDATE Race *SET* prix = 900 WHERE id = 2;

UPDATE Race *SET* prix = 950 WHERE id = 3;

UPDATE Race *SET* prix = 800 WHERE id = 4;

UPDATE Race *SET* prix = 700 WHERE id = 5;

UPDATE Race *SET* prix = 1200 WHERE id = 7;

UPDATE Race *SET* prix = 950 WHERE id = 8;

UPDATE Race *SET* prix = 600 WHERE id = 9;

Le prix de la race sera prioritaire sur le prix de l'espèce. Donc pour un berger allemand, on ira chercher le prix dans la table Race, tandis que pour un bâtard ou une tortue d'Hermann, on prendra le prix de la table Espece.

##### Opérations sur données sélectionnées

**Exemple 1 :** imaginons que nous voulons connaître le prix à payer pour acheter 3 individus de la même espèce (sans considérer la race). Facile, il suffit de multiplier le prix de chaque espèce par 3.

SELECT nom\_courant, prix\*3 AS prix\_trio

FROM Espece;

**Exemple 2 :** toutes les opérations courantes peuvent bien entendu être utilisées.

SELECT nom\_courant, prix,

prix+100 AS addition, prix/2 AS division,

prix-50.5 AS soustraction, prix%3 AS modulo

FROM Espece;

##### 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.

**Exemple :**la commande suivante va augmenter le prix de toutes les races de 35.

UPDATE Race

*SET* prix = prix + 35;

C'est quand même plus élégant que de devoir faire une requête SELECT  pour savoir le prix actuel des races, calculer ces prix +35, puis faire un UPDATE  pour chaque race avec le résultat.

Bien entendu, on peut faire ce genre de manipulation également dans une requête de typeINSERT INTO ... SELECT, par exemple. En fait, on peut faire ce genre de manipulation partout.

### Définition d'une fonction

Nous avons déjà utilisé quelques fonctions dans ce cours. Par exemple, dans le chapitre sur les sous-requêtes, je vous proposais la requête suivante :

SELECT MIN(date\_naissance) -- On utilise ici une fonction !

FROM (

SELECT Animal.id, Animal.sexe, Animal.date\_naissance, 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')

) 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 cela !

##### 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 ces instructions** (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 paramètres, 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 que l'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

-- Fonction sans paramètre

SELECT PI(); -- renvoie le nombre Pi, avec 5 décimales

-- Fonction avec un paramètre

SELECT MIN(prix) AS minimum -- il est bien sûr possible d'utiliser les alias !

FROM Espece;

-- Fonction avec plusieurs paramètres

SELECT REPEAT('fort ! Trop ', 4); -- répète une chaîne (ici : 'fort ! Trop ', répété 4 fois)

-- Même chose qu'au-dessus, mais avec les paramètres dans le mauvais ordre

SELECT REPEAT(4, 'fort ! Trop '); -- la chaîne de caractères 'fort ! Trop ' va être convertie en entier par MySQL, ce qui donne 0. "4" va donc être répété zéro fois...

#### 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 seule). Un petit exemple rendra cette explication lumineuse.

##### Fonction scalaire

La fonction ROUND(X)  arrondit X à l'entier le plus proche. Il s'agit d'une fonction scalaire.

SELECT nom, prix, ROUND(prix)

FROM Race;

##### Fonction d'agrégation

La fonction MIN(X), par contre, est une fonction d'agrégation.

SELECT MIN(prix)

FROM Race;

### 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 des exemples, mais nous ne verrons pas les petits cas particuliers, les exceptions, ni les éventuels comportements étranges et imprévisibles. Pour cela, la documentation officielle est, et restera, votre meilleure compagne. 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 répétera jamais assez, n'hésitez pas à faire un tour sur la documentation 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.

SELECT VERSION();

**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 cela 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 droits, tandis que CURRENT\_USER()  vous donnera l'utilisateur anonyme.

Dans notre cas, l’utilisateur "sdz" (ou n'importe quel user que vous avez créé) ayant des droits, les deux fonctions renverront exactement la même chose.

SELECT CURRENT\_USER(), USER();

#### Informations sur la dernière requête

##### Dernier ID généré 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 le dernier id créé 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 recevrez).

INSERT INTO Race (nom, espece\_id, description, prix)

VALUES ('Rottweiller', 1, 'Chien d''apparence solide, bien musclé, à la robe noire avec des taches feu bien délimitées.', 600.00);

INSERT INTO Animal (sexe, date\_naissance, nom, espece\_id, race\_id)

VALUES ('M', '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 rapportées.

SELECT id, nom, espece\_id, prix

FROM Race;

SELECT FOUND\_ROWS();

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 rapportées en l'absence de LIMIT. L'option à ajouter dans le SELECT  est SQL\_CALC\_FOUND\_ROWS, et elle se place juste après le mot-clé SELECT.

SELECT id, nom, espece\_id, prix -- Sans option

FROM Race

LIMIT 3;

SELECT FOUND\_ROWS() AS sans\_option;

SELECT SQL\_CALC\_FOUND\_ROWS id, nom, espece\_id, prix -- Avec option

FROM Race

LIMIT 3;

SELECT FOUND\_ROWS() AS avec\_option;

#### 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 :** conversions automatiques.

SELECT \*

FROM Espece

WHERE id = '3';

INSERT INTO Espece (nom\_latin, nom\_courant, description, prix)

VALUES ('Rattus norvegicus', 'Rat brun', 'Petite bestiole avec de longues moustaches et une longue queue sans poils', '10.00');

La colonne id est de type INT. Pourtant, la comparaison avec une chaîne de caractères renvoie bien un résultat. De même, la colonne prix est de type DECIMAL, 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-entenduINT), SIGNED  (sous-entendu INT), DECIMAL.

**Exemple :** conversion d'une chaîne de caractères en date.

SELECT CAST('870303' AS *DATE*);

## Fonctions scalaires

Comme prévu, ce chapitre sera consacré aux **fonctions scalaires** permettant la 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.

À 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 cœur 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 dans la documentation.

### Manipulation de nombres

Voici donc quelques fonctions scalaires qui vont vous permettre de manipuler les nombres : faire des calculs, des arrondis, prendre un nombre au hasard, etc.

Toutes les fonctions de cette partie retournent NULL  en cas d'erreur !

#### Arrondis

Arrondir un nombre, c'est trouver une valeur proche de ce nombre avec une précision donnée et selon certains critères.  
La précision est en général représentée par le nombre de décimales désirées. Par exemple, pour un prix, on travaille rarement avec plus de deux décimales. Pour un âge, on préférera généralement un nombre entier (c'est-à-dire aucune décimale).  
Quant aux critères, il s'agit de décider si l'on veut arrondir au plus proche (ex. : 4,3 arrondi à l'entier le plus proche vaut 4), arrondir au supérieur (ex. : 4,3 arrondi à l'entier supérieur vaut 5) ou arrondir à l'inférieur (ex. : 4,3 arrondi à l'entier inférieur vaut 4).

Voici quatre fonctions permettant d'arrondir les nombres selon ces 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.

SELECT CEIL(3.2), CEIL(3.7);

##### FLOOR()

FLOOR(n)  arrondit au nombre entier inférieur.

SELECT FLOOR(3.2), FLOOR(3.7);

##### 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.

SELECT ROUND(3.22, 1), ROUND(3.55, 1), ROUND(3.77, 1);

SELECT ROUND(3.2), ROUND(3.5), ROUND(3.7);

##### TRUNCATE()

TRUNCATE(n, d)  arrondit en enlevant purement et simplement les décimales en trop (donc arrondi à l'inférieur pour les nombres positifs, au supérieur pour les nombres négatifs).

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);

#### Exposants et racines

##### Exposants

POWER(n, e)  (ou POW(n, e)) retourne le résultat de n exposant e ( nene ).

Pour rappel, n exposant e ( nene ) veut dire que l'on multiplie n par lui-même, e fois. Donc par exemple, 23=2×2×2=823=2×2×2=8

SELECT POW(2, 5), POWER(5, 2);

##### Racines

Prendre la racine nième d'un nombre x ( x−−√nxn ), c'est trouver le ou les nombres y qui répondent à la condition suivante : yn=xyn=x . Donc la racine cinquième de 32 ( 32−−√5325 ) vaut 2, puisque 25=2×2×2×2×2=3225=2×2×2×2×2=32

SQRT(n)  donne la racine carrée positive de n ( n−−√=n−−√2n=n2 ).

SELECT SQRT(4);

#### 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.

##### ABS()

ABS(n)  retourne la valeur absolue de n, donc sa valeur sans le signe.

##### MOD()

La fonction MOD(n, div)  retourne le modulo, donc le reste de la division entière de n par div (comme l'opérateur %  ou MOD).

### Exemples d'application et exercices

Ce chapitre a été 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 prix de l'espèce, pour chaque espèce

Par exemple, afficher "Un chat coûte 100 euros.", ou une autre phrase de ce genre, et ce pour les cinq espèces enregistrées.

SELECT CONCAT('Un(e) ', nom\_courant, ' coûte ', prix, ' euros.') AS Solution

FROM Espece;

-- OU

SELECT CONCAT\_WS(' ','Un(e)', nom\_courant, 'coûte', prix, 'euros.') AS Solution

FROM Espece;

##### 2. Afficher les chats dont la deuxième lettre du nom est un "a"

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 perroquets Kikoolol

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’occurrence, 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

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().  
Pour savoir si un nombre est pair, il faut utiliser les modulos : lorsqu'un nombre est pair, le reste d'une division entière de ce nombre par 2 est 0, donc ce nombre **modulo** 2 vaut 0.

## Fonctions d'agrégation

Les fonctions d'agrégation, ou de groupement, sont 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 plupart des fonctions d'agrégation vont vous permettre de faire des statistiques sur vos données.

#### Nombre de lignes

La fonction COUNT()  permet de savoir combien de lignes sont sélectionnées par la requête.

-- 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';

##### 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 l'on utilise COUNT(colonne), seules les lignes dont la valeur de colonne n'est pas NULL  seront prises en compte.

**Exemple :** comptons les lignes de la table Animal, avec COUNT(\*)  et COUNT(race\_id).

SELECT COUNT(race\_id), COUNT(\*)

FROM Animal;

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.

**Exemple :** comptons le nombre de races distinctes définies dans la table Animal.

SELECT COUNT(DISTINCT race\_id)

FROM Animal;

#### 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.

SELECT MIN(prix), MAX(prix)

FROM Race;

Notez que MIN()  et MAX()  ne s'utilisent pas uniquement sur des données numériques. Si vous lui passez des chaînes de caractères, MIN()  récupérera la première chaîne dans l'ordre alphabétique, MAX()  la dernière. Avec des dates, MIN()  renverra la plus ancienne et MAX()  la plus récente.

**Exemple :**

SELECT MIN(nom), MAX(nom), MIN(date\_naissance), MAX(date\_naissance)

FROM Animal;

**Somme et moyenne**

**Somme**

La fonction SUM(x)  renvoie la somme de *x*.

SELECT SUM(prix)

FROM Espece;

**Moyenne**

La fonction AVG(x)  (du mot anglais *average*) renvoie la valeur moyenne de *x*.

SELECT AVG(prix)

FROM Espece;

### 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.  
Ceci est faisable avec GROUP\_CONCAT(nom\_colonne). Cette fonction concatène les valeurs de nom\_colonne pour chaque groupement réalisé.

**Exemple :** on récupère la somme des prix de chaque espèce, et on affiche les espèces concernées par la même occasion.

SELECT SUM(prix), GROUP\_CONCAT(nom\_courant)

FROM Espece;

#### Syntaxe

Voici la syntaxe de cette fonction :

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

-- --------------------------------------

-- CONCATENATION DE PLUSIEURS COLONNES --

-- --------------------------------------

SELECT SUM(Race.prix), GROUP\_CONCAT(Race.nom, Espece.nom\_courant)

FROM Race

INNER JOIN Espece ON Espece.id = Race.espece\_id;

-- ---------------------------------------------------

-- CONCATENATION DE PLUSIEURS COLONNES EN PLUS JOLI --

-- ---------------------------------------------------

SELECT SUM(Race.prix), GROUP\_CONCAT(Race.nom, ' (', Espece.nom\_courant, ')')

FROM Race

INNER JOIN Espece ON Espece.id = Race.espece\_id;

-- ---------------------------

-- ELIMINATION DES DOUBLONS --

-- ---------------------------

SELECT SUM(Espece.prix), 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(Race.prix), GROUP\_CONCAT(Race.nom, ' (', Espece.nom\_courant, ')' ORDER BY Race.nom DESC)

FROM Race

INNER JOIN Espece ON Espece.id = Race.espece\_id;

-- ----------------------------

-- CHANGEMENT DE SEPARATEUR --

-- ----------------------------

SELECT SUM(Race.prix), GROUP\_CONCAT(Race.nom, ' (', Espece.nom\_courant, ')' SEPARATOR ' - ')

FROM Race

INNER JOIN Espece ON Espece.id = Race.espece\_id;

## Regroupement

Vous savez donc que les fonctions d'agrégation groupent plusieurs lignes ensemble. Jusqu'à maintenant, toutes les lignes étaient 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 espèce, 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 la base de critères.

### Regroupement sur un critère

Pour regrouper les lignes selon un critère, il faut utiliser GROUP BY, qui se place après l'éventuelle clause WHERE  (sinon directement après FROM), suivi du nom de la colonne à utiliser comme critère de regroupement.

SELECT ...

FROM nom\_table

[WHERE condition]

GROUP BY nom\_colonne;

**Exemple 1 :** comptons les lignes dans la table Animal, en regroupant sur le critère de l'espèce (donc avec la colonne espece\_id).

SELECT COUNT(\*) AS nb\_animaux

FROM Animal

GROUP BY espece\_id;

**Exemple 2 :** même chose, mais on ne prend que les mâles cette fois-ci.

SELECT COUNT(\*) AS nb\_males

FROM Animal

WHERE sexe = 'M'

GROUP BY espece\_id;

#### olonnes sélectionnées

##### La règle SQL

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 colonnes **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 :

SELECT nom\_courant, COUNT(\*) AS nb\_animaux, date\_naissance

FROM Animal

INNER JOIN Espece ON Animal.espece\_id = Espece.id

GROUP BY nom\_courant;

##### Le cas MySQL

On ne le répétera 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 la 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.

SELECT nom\_courant, COUNT(\*) AS nb\_animaux, date\_naissance

FROM Animal

INNER JOIN Espece ON Animal.espece\_id = Espece.id

GROUP BY espece\_id;

**Tri des données**

Selon le même principe, il n'est possible de faire un tri des données qu'à partir d'une colonne qui fait partie des critères de regroupement, ou à partir d'une fonction d'agrégation. Cela 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 écrire ceci :

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;

### Regroupement sur plusieurs critères

J'ai mentionné le fait qu'il était possible de grouper sur plusieurs colonnes, mais jusqu'à maintenant, cela n'a servi qu'à pouvoir afficher correctement les colonnes voulues, sans que cela n'influe sur les groupes. On n'avait donc qu'un seul critère, représenté par plusieurs colonnes. Voyons maintenant un exemple avec deux critères différents (qui 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.

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;

**Exemple 1 :** 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;

**Exemple 2 :** 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;

### 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

SELECT nom\_courant, COUNT(\*) as nb\_animaux

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

GROUP BY nom\_courant WITH ROLLUP;

##### Exemple avec deux critères de regroupement

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;

C'est en utilisant WITH ROLLUP  que l'on se rend compte que l'ordre des critères est vraiment important. En effet, voyons ce qui se passe si l'on échange les deux critères *nom\_courant* et *sexe*.

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;

**NULL, c'est pas joli**

Il est possible d'éviter d'avoir ces NULL dans les lignes des super-groupes. Pour cela, on peut utiliser la fonction COALESCE().  
Cette fonction prend autant de paramètres que l'on veut, et renvoie le premier paramètre non NULL.

**Exemples :**

SELECT COALESCE(1, NULL, 3, 4); -- 1

SELECT COALESCE(NULL, 2); -- 2

SELECT COALESCE(NULL, NULL, 3); -- 3

Voici comment l'utiliser dans le cas des super-agrégats.

SELECT COALESCE(nom\_courant, 'Total'), COUNT(\*) as nb\_animaux

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

GROUP BY nom\_courant WITH ROLLUP;

Tant qu'il s'agit de simples groupes, *nom\_courant* contient bien le nom de l'espèce. COALESCE()  renvoie donc celui-ci. Par contre, quand il s'agit du super-groupe, la colonne *nom\_courant* du résultat contient NULL, et donc COALESCE()  va renvoyer "Total".

Si vous utilisez COALESCE()  dans ce genre de situation, il est impératif que vos critères de regroupement ne contiennent pas NULL  (ou que vous éliminiez ces lignes-là). Sinon, vous aurez "Total" à des lignes qui ne sont pas des super-groupes.

**Exemple :** groupons sur le sexe, sans éliminer les lignes pour lesquelles le sexe n'est pas défini.

SELECT COALESCE(sexe, 'Total'), COUNT(\*) as nb\_animaux

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

GROUP BY sexe WITH ROLLUP;

### Conditions sur les fonctions d'agrégation

Il n'est pas possible d'utiliser la clause WHERE  pour faire des conditions sur une fonction d'agrégation. Donc, si l'on veut afficher les espèces dont on possède plus de 15 individus, la requête suivante ne fonctionnera pas.

SELECT nom\_courant, COUNT(\*)

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

WHERE COUNT(\*) > 15

GROUP BY nom\_courant;

ERROR 1111 (HY000): Invalid use of group function

Il faut utiliser une clause spéciale : HAVING. Cette clause se place juste après le GROUP BY.

SELECT nom\_courant, COUNT(\*)

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

GROUP BY nom\_courant

HAVING COUNT(\*) > 15;

Il est également possible d'utiliser un alias dans une condition HAVING  :

SELECT nom\_courant, COUNT(\*) as nombre

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

GROUP BY nom\_courant

HAVING nombre > 15;

**Optimisation**

Les conditions données dans la clause HAVING  ne doivent pas nécessairement comporter une fonction d'agrégation. Les deux requêtes suivantes donneront par exemple des résultats équivalents :

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';

-- Deux conditions dans HAVING

SELECT nom\_courant, COUNT(\*) as nombre

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

WHERE SUBSTRING(nom\_courant, 1, 1) = 'C'

-- Une condition dans WHERE

GROUP BY nom\_courant

HAVING nombre > 6;

-- Et une dans HAVING

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, à l'inverse des conditions WHERE.

## Exercices sur les agrégats

Jusqu'à maintenant, tout a été très théorique. Or, la meilleure 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-mêmes avant de regarder la solution, ne restez cependant pas bloqué trop longtemps sur un exercice, et prenez toujours le temps de bien comprendre la solution.

### Du simple…

#### 1. Combien de races avons-nous dans la table Race ?

SELECT COUNT(\*)

FROM Race;

Simple échauffement.

#### 2. De combien de chiens connaissons-nous le père ?

SELECT COUNT(pere\_id)

FROM Animal

INNER JOIN Espece 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 où pere\_id est non NULL. Si vous avez fait directement WHERE espece\_id = 1  au lieu d'utiliser une jointure pour sélectionner les chiens, ce n'est pas bien grave.

#### 3. Quelle est la date de naissance de notre plus jeune femelle ?

SELECT MAX(date\_naissance)

FROM Animal

WHERE sexe = 'F';

#### 4. En moyenne, quel est le prix d'un chien ou d'un chat de race, par espèce, et en général ?

SELECT nom\_courant AS Espece, AVG(Race.prix) AS prix\_moyen

FROM Race

INNER JOIN Espece ON Race.espece\_id = Espece.id

WHERE Espece.nom\_courant IN ('Chat', 'Chien')

GROUP BY Espece.nom\_courant WITH ROLLUP;

Ne pas oublier WITH ROLLUP  pour avoir le résultat général.

#### 5. Combien avons-nous de perroquets mâles et femelles, et quels sont leurs noms (en une seule requête, bien sûr) ?

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 souvenir 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, puisque l'on avait restreint à une seule espèce avec la clause WHERE. Cependant, cela n'influe pas sur le résultat, mais sur la rapidité de la requête.

### …Vers le complexe

#### 1. Quelles sont les races dont nous ne possédons aucun individu ?

SELECT Race.nom, COUNT(Animal.race\_id) AS nombre

FROM Race

LEFT JOIN Animal ON Animal.race\_id = Race.id

GROUP BY Race.nom

HAVING nombre = 0;

Ici, il ne fallait pas oublier de faire une jointure externe (LEFT  ou RIGHT, selon votre requête), ainsi que de mettre la colonne Animal.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 ?

SELECT Espece.nom\_latin, COUNT(espece\_id) AS nombre

FROM Espece

LEFT JOIN Animal ON Animal.espece\_id = Espece.id

WHERE sexe = 'M' OR Animal.id IS NULL

GROUP BY Espece.nom\_latin

HAVING nombre < 5;

À nouveau, une jointure externe et espece\_id en argument de COUNT(), mais il y avait ici une petite subtilité en plus.  
Puisque l'on demandait des informations sur les mâles uniquement, il fallait une condition WHERE sexe = 'M'. 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 mettre cette condition sur toute autre colonne d'Animal ayant la contrainte NOT NULL, et qui ne sera NULL  que lors d'une jointure externe, en cas de non-correspondance avec l'autre table.  
Il n'y a alors plus qu'à ajouter la clause HAVING  pour sélectionner les espèces ayant moins de cinq mâles.

#### 3. Combien de mâles et de femelles de chaque race avons-nous, avec un compte total intermédiaire pour les races (mâles et femelles confondues) et pour les espèces ? Afficher le nom de la race et le nom courant de l'espèce.

SELECT Animal.sexe, Race.nom, Espece.nom\_courant, COUNT(\*) AS nombre

FROM Animal

INNER JOIN Espece ON Animal.espece\_id = Espece.id

INNER JOIN Race ON Animal.race\_id = Race.id

WHERE Animal.sexe IS NOT NULL

GROUP BY Espece.nom\_courant, Race.nom, sexe WITH ROLLUP;

Deux jointures sont nécessaires pour pouvoir afficher les noms des races et des espèces. 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. Quel serait le coût, par espèce et au total, de l'adoption de Parlotte, Spoutnik, Caribou, Cartouche, Cali, Canaille, Yoda, Zambo et Lulla ?

Petit indice, pour avoir le prix d'un animal selon que sa race soit définie ou non, vous pouvez utiliser une fonction que nous venons de voir au chapitre précédent.

SELECT Espece.nom\_courant, SUM(COALESCE(Race.prix, Espece.prix)) AS somme

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

LEFT JOIN Race ON Race.id = Animal.race\_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, puisque l'on veut le prix total par groupe. Sans oublier le WITH ROLLUP  pour avoir également le prix total tous groupes confondus.  
Quant au prix de chaque animal, c'est typiquement une situation où l'on peut utiliser COALESCE()  !

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 fonctionnent, couplées à GROUP BY  ou non.

## Obtenez la date/l'heure actuelle

Avant de plonger la 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.

**État actuel de la base de données**  
**Note :** les tables de test ne sont pas reprises.

*SET* NAMES utf8;

DROP TABLE IF EXISTS Animal;

DROP TABLE IF EXISTS Race;

DROP TABLE IF EXISTS Espece;

CREATE TABLE Espece (

id smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

nom\_courant *varchar*(40) NOT NULL,

nom\_latin *varchar*(40) NOT NULL,

description *text*,

prix *decimal*(7,2) unsigned DEFAULT NULL,

PRIMARY KEY (id),

UNIQUE KEY nom\_latin (nom\_latin)

) ENGINE=InnoDB AUTO\_INCREMENT=6 DEFAULT CHARSET=latin1;

LOCK TABLES Espece WRITE;

INSERT INTO Espece VALUES (1,'Chien','Canis canis','Bestiole à quatre pattes qui aime les caresses et tire souvent la langue',200.00),(2,'Chat','Felis silvestris','Bestiole à quatre pattes qui saute très haut et grimpe aux arbres',150.00),(3,'Tortue d''Hermann','Testudo hermanni','Bestiole avec une carapace très dure',140.00),

(4,'Perroquet amazone','Alipiopsitta xanthops','Joli oiseau parleur vert et jaune',700.00),(5,'Rat brun','Rattus norvegicus','Petite bestiole avec de longues moustaches et une longue queue sans poils',10.00);

UNLOCK TABLES;

CREATE TABLE Race (

id smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

nom *varchar*(40) NOT NULL,

espece\_id smallint(6) unsigned NOT NULL,

description *text*,

prix *decimal*(7,2) unsigned DEFAULT NULL,

PRIMARY KEY (id)

) ENGINE=InnoDB AUTO\_INCREMENT=10 DEFAULT CHARSET=latin1;

LOCK TABLES Race WRITE;

INSERT INTO Race VALUES (1,'Berger allemand',1,'Chien sportif et élégant au pelage dense, noir-marron-fauve, noir ou gris.',485.00),(2,'Berger blanc suisse',1,'Petit chien au corps compact, avec des pattes courtes mais bien proportionnées et au pelage tricolore ou bicolore.',935.00),(3,'Singapura',2,'Chat de petite taille aux grands yeux en amandes.',985.00),

(4,'Bleu russe',2,'Chat aux yeux verts et à la robe épaisse et argentée.',835.00),(5,'Maine coon',2,'Chat de grande taille, à poils mi-longs.',735.00),(7,'Sphynx',2,'Chat sans poils.',1235.00),

(8,'Nebelung',2,'Chat bleu russe, mais avec des poils longs...',985.00),(9,'Rottweiller',1,'Chien d''apparence solide, bien musclé, à la robe noire avec des taches feu bien délimitées.',600.00);

UNLOCK TABLES;

CREATE TABLE Animal (

id smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

sexe *char*(1) DEFAULT NULL,

date\_naissance datetime NOT NULL,

nom *varchar*(30) DEFAULT NULL,

commentaires *text*,

espece\_id smallint(6) unsigned NOT NULL,

race\_id smallint(6) unsigned DEFAULT NULL,

mere\_id smallint(6) unsigned DEFAULT NULL,

pere\_id smallint(6) unsigned DEFAULT NULL,

PRIMARY KEY (id),

UNIQUE KEY ind\_uni\_nom\_espece\_id (nom,espece\_id)

) ENGINE=InnoDB AUTO\_INCREMENT=63 DEFAULT CHARSET=utf8;

LOCK TABLES Animal WRITE;

INSERT INTO Animal VALUES (1,'M','2010-04-05 13:43:00','Rox','Mordille beaucoup',1,1,18,22),(2,NULL,'2010-03-24 02:23:00','Roucky',NULL,2,NULL,40,30),(3,'F','2010-09-13 15:02:00','Schtroumpfette',NULL,2,4,41,31),

(4,'F','2009-08-03 05:12:00',NULL,'Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(5,NULL,'2010-10-03 16:44:00','Choupi','Né sans oreille gauche',2,NULL,NULL,NULL),(6,'F','2009-06-13 08:17:00','Bobosse','Carapace bizarre',3,NULL,NULL,NULL),

(7,'F','2008-12-06 05:18:00','Caroline',NULL,1,2,NULL,NULL),(8,'M','2008-09-11 15:38:00','Bagherra',NULL,2,5,NULL,NULL),(9,NULL,'2010-08-23 05:18:00',NULL,'Bestiole avec une carapace très dure',3,NULL,NULL,NULL),

(10,'M','2010-07-21 15:41:00','Bobo',NULL,1,NULL,7,21),(11,'F','2008-02-20 15:45:00','Canaille',NULL,1,NULL,NULL,NULL),(12,'F','2009-05-26 08:54:00','Cali',NULL,1,2,NULL,NULL),

(13,'F','2007-04-24 12:54:00','Rouquine',NULL,1,1,NULL,NULL),(14,'F','2009-05-26 08:56:00','Fila',NULL,1,2,NULL,NULL),(15,'F','2008-02-20 15:47:00','Anya',NULL,1,NULL,NULL,NULL),

(16,'F','2009-05-26 08:50:00','Louya',NULL,1,NULL,NULL,NULL),(17,'F','2008-03-10 13:45:00','Welva',NULL,1,NULL,NULL,NULL),(18,'F','2007-04-24 12:59:00','Zira',NULL,1,1,NULL,NULL),

(19,'F','2009-05-26 09:02:00','Java',NULL,1,2,NULL,NULL),(20,'M','2007-04-24 12:45:00','Balou',NULL,1,1,NULL,NULL),(21,'F','2008-03-10 13:43:00','Pataude',NULL,1,NULL,NULL,NULL),

(22,'M','2007-04-24 12:42:00','Bouli',NULL,1,1,NULL,NULL),(24,'M','2007-04-12 05:23:00','Cartouche',NULL,1,NULL,NULL,NULL),(25,'M','2006-05-14 15:50:00','Zambo',NULL,1,1,NULL,NULL),

(26,'M','2006-05-14 15:48:00','Samba',NULL,1,1,NULL,NULL),(27,'M','2008-03-10 13:40:00','Moka',NULL,1,NULL,NULL,NULL),(28,'M','2006-05-14 15:40:00','Pilou',NULL,1,1,NULL,NULL),

(29,'M','2009-05-14 06:30:00','Fiero',NULL,2,3,NULL,NULL),(30,'M','2007-03-12 12:05:00','Zonko',NULL,2,5,NULL,NULL),(31,'M','2008-02-20 15:45:00','Filou',NULL,2,4,NULL,NULL),

(32,'M','2009-07-26 11:52:00','Spoutnik',NULL,3,NULL,52,NULL),(33,'M','2006-05-19 16:17:00','Caribou',NULL,2,4,NULL,NULL),(34,'M','2008-04-20 03:22:00','Capou',NULL,2,5,NULL,NULL),

(35,'M','2006-05-19 16:56:00','Raccou','Pas de queue depuis la naissance',2,4,NULL,NULL),(36,'M','2009-05-14 06:42:00','Boucan',NULL,2,3,NULL,NULL),(37,'F','2006-05-19 16:06:00','Callune',NULL,2,8,NULL,NULL),

(38,'F','2009-05-14 06:45:00','Boule',NULL,2,3,NULL,NULL),(39,'F','2008-04-20 03:26:00','Zara',NULL,2,5,NULL,NULL),(40,'F','2007-03-12 12:00:00','Milla',NULL,2,5,NULL,NULL),

(41,'F','2006-05-19 15:59:00','Feta',NULL,2,4,NULL,NULL),(42,'F','2008-04-20 03:20:00','Bilba','Sourde de l''oreille droite à 80%',2,5,NULL,NULL),(43,'F','2007-03-12 11:54:00','Cracotte',NULL,2,5,NULL,NULL),

(44,'F','2006-05-19 16:16:00','Cawette',NULL,2,8,NULL,NULL),(45,'F','2007-04-01 18:17:00','Nikki','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(46,'F','2009-03-24 08:23:00','Tortilla','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),

(47,'F','2009-03-26 01:24:00','Scroupy','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(48,'F','2006-03-15 14:56:00','Lulla','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(49,'F','2008-03-15 12:02:00','Dana','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),

(50,'F','2009-05-25 19:57:00','Cheli','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(51,'F','2007-04-01 03:54:00','Chicaca','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(52,'F','2006-03-15 14:26:00','Redbul','Insomniaque',3,NULL,NULL,NULL),

(54,'M','2008-03-16 08:20:00','Bubulle','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(55,'M','2008-03-15 18:45:00','Relou','Surpoids',3,NULL,NULL,NULL),(56,'M','2009-05-25 18:54:00','Bulbizard','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),

(57,'M','2007-03-04 19:36:00','Safran','Coco veut un gâteau !',4,NULL,NULL,NULL),(58,'M','2008-02-20 02:50:00','Gingko','Coco veut un gâteau !',4,NULL,NULL,NULL),(59,'M','2009-03-26 08:28:00','Bavard','Coco veut un gâteau !',4,NULL,NULL,NULL),

(60,'F','2009-03-26 07:55:00','Parlotte','Coco veut un gâteau !',4,NULL,NULL,NULL),(61,'M','2010-11-09 00:00:00','Yoda',NULL,2,5,NULL,NULL),(62,'M','2010-11-05 00:00:00','Pipo',NULL,1,9,NULL,NULL);

UNLOCK TABLES;

ALTER TABLE Race ADD CONSTRAINT fk\_race\_espece\_id FOREIGN KEY (espece\_id) REFERENCES Espece (id) ON DELETE CASCADE;

ALTER TABLE Animal ADD CONSTRAINT fk\_race\_id FOREIGN KEY (race\_id) REFERENCES Race (id) ON DELETE *SET* NULL;

ALTER TABLE Animal ADD CONSTRAINT fk\_espece\_id FOREIGN KEY (espece\_id) REFERENCES Espece (id);

ALTER TABLE Animal ADD CONSTRAINT fk\_mere\_id FOREIGN KEY (mere\_id) REFERENCES Animal (id) ON DELETE *SET* NULL;

ALTER TABLE Animal ADD CONSTRAINT fk\_pere\_id FOREIGN KEY (pere\_id) REFERENCES Animal (id) ON DELETE *SET* NULL;

### 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](http://www.siteduzero.com/tutoriel-3-464508-les-types-de-donnees.html#ss_part_3) 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 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 24 heures, 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'une 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 comporte 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, 0 h 0 min 0 s (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 secondes, 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 1er janvier 1970 0 h 0 min 0s au 19 janvier 2038 3 h 14 min 7 s.

#### 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 au format DATE) :

* CURDATE();
* CURRENT\_DATE(), qui peut également s'utiliser sans parenthèses, comme plusieurs autres fonctions temporelles.

SELECT CURDATE(), CURRENT\_DATE(), CURRENT\_DATE;

### 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.

SELECT CURTIME(), CURRENT\_TIME(), CURRENT\_TIME;

| **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.

SELECT NOW(), SYSDATE();

##### 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()

SELECT LOCALTIME, CURRENT\_TIMESTAMP(), LOCALTIMESTAMP;

**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 trois 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.

-- 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;

**Timestamp Unix**

Il existe encore une fonction qui peut donner des informations sur la date et l'heure actuelles sous forme de timestamp Unix, 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 !

SELECT UNIX\_TIMESTAMP();

## Formatez 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 cela, il se prive des nombreux avantages des formats temporels SQL (en particulier toutes les fonctions que nous voyons 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…).

SELECT nom, date\_naissance,

*DATE*(date\_naissance) AS uniquementDate

FROM Animal

WHERE espece\_id = 4;

##### 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 :**

SELECT nom, *DATE*(date\_naissance) AS date\_naiss,

DAY(date\_naissance) AS jour,

DAYOFMONTH(date\_naissance) AS jour,

DAYOFWEEK(date\_naissance) AS jour\_sem,

WEEKDAY(date\_naissance) AS jour\_sem2,

DAYNAME(date\_naissance) AS nom\_jour,

DAYOFYEAR(date\_naissance) AS jour\_annee

FROM Animal

WHERE espece\_id = 4;

Tout cela 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 :

*SET* lc\_time\_names = 'fr\_FR';

Et voilà le travail :

SELECT nom, date\_naissance,

DAYNAME(date\_naissance) AS jour\_semaine

FROM Animal

WHERE espece\_id = 4;

##### La semaine

À partir d'une date, il est possible de calculer la semaine à laquelle elle correspond. S'agit-il de la première semaine de l'année ? De la quinzième ? Ceci 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 7 x 52 = 364, donc en un an, il y a 52 semaines et 1 ou 2 jours 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.

SELECT nom, date\_naissance, WEEK(date\_naissance) AS semaine, WEEKOFYEAR(date\_naissance) AS semaine2, YEARWEEK(date\_naissance) AS semaine\_annee

FROM Animal

WHERE espece\_id = 4;

WEEK()  et YEARWEEK()  peuvent également accepter un deuxième argument, qui sert à spécifier si la semaine doit commencer le lundi ou le dimanche, et ce que l'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.

SELECT nom, date\_naissance, MONTH(date\_naissance) AS numero\_mois, MONTHNAME(date\_naissance) AS nom\_mois

FROM Animal

WHERE espece\_id = 4;

**L'année**

Enfin, la fonction YEAR(date)  extrait l'année.

SELECT nom, date\_naissance, YEAR(date\_naissance)

FROM Animal

WHERE espece\_id = 4;

#### Informations sur l'heure

En ce qui concerne l'heure, voici quatre fonctions intéressantes (et faciles à retenir) qui s'appliquent à une donnée de type DATETIME  ou TIME  :

* TIME(datetime)  : qui extrait l'heure complète (le TIME).
* HOUR(heure)  : qui extrait l'heure.
* MINUTE(heure)  : qui extrait les minutes.
* SECOND(heure)  : qui extrait les secondes.

SELECT nom, date\_naissance,

TIME(date\_naissance) AS time\_complet,

HOUR(date\_naissance) AS heure,

MINUTE(date\_naissance) AS minutes,

SECOND(date\_naissance) AS secondes

FROM Animal

WHERE espece\_id = 4;

### 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".

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;

**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 suivant.

| **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 :

SELECT nom, date\_naissance, DATE\_FORMAT(date\_naissance, 'le %W %e %M %Y') AS jolie\_date

FROM Animal

WHERE espece\_id = 4;

##### Autres exemples

Attention à bien échapper les guillemets éventuels dans la chaîne de caractères du format.

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;

**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 ne sert qu'à formater les heures (et ne doit 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 NULLou 0  est renvoyé.  
Si un mauvais format de TIME  ou DATETIME  est donné à TIME\_FORMAT()  (par exemple, si on lui donne une DATE), MySQL va tenter d'interpréter la donnée et renverra bien un résultat, mais celui-ci n'aura peut-être pas beaucoup de sens (pour vous du moins).

**Exemples :**

-- Sur une DATETIME

SELECT TIME\_FORMAT(NOW(), '%r') AS sur\_datetime,

TIME\_FORMAT(CURTIME(), '%r') AS sur\_time,

TIME\_FORMAT(NOW(), '%M %r') AS mauvais\_specificateur,

TIME\_FORMAT(CURDATE(), '%r') AS sur\_date;

#### Formats standard

Il existe un certain nombre de formats de date et d'heure standard 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.

Il existe cinq formats standard :

* 'EUR'
* 'USA'
* 'ISO'
* 'JIS'
* 'INTERNAL'

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 13.44.33 |
| GET\_FORMAT(DATETIME,'JIS') | '%Y-%m-%d %H:%i:%s' | 1988-10-30 13:44:33 |
| GET\_FORMAT(DATETIME,'ISO') | '%Y-%m-%d %H:%i:%s' | 1988-10-30 13:44:33 |
| GET\_FORMAT(DATETIME,'EUR') | '%Y-%m-%d %H.%i.%s' | 1988-10-30 13.44.33 |
| GET\_FORMAT(DATETIME,'INTERNAL') | '%Y%m%d%H%i%s' | 19881030134433 |
| GET\_FORMAT(TIME,'USA') | '%h:%i:%s %p' | 1:44:33 PM |
| GET\_FORMAT(TIME,'JIS') | '%H:%i:%s' | 13:44:33 |
| GET\_FORMAT(TIME,'ISO') | '%H:%i:%s' | 13:44:33 |
| GET\_FORMAT(TIME,'EUR') | '%H.%i.%S' | 13.44.33 |
| GET\_FORMAT(TIME,'INTERNAL') | '%H%i%s' | 134433 |

**Exemples :**

SELECT DATE\_FORMAT(NOW(), GET\_FORMAT(*DATE*, 'EUR')) AS date\_eur,

DATE\_FORMAT(NOW(), GET\_FORMAT(TIME, 'JIS')) AS heure\_jis,

DATE\_FORMAT(NOW(), GET\_FORMAT(DATETIME, 'USA')) AS date\_heure\_usa;

### 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 :**

SELECT STR\_TO\_DATE('03/04/2011 à 09h17', '%d/%m/%Y à %Hh%i') AS StrDate,

STR\_TO\_DATE('15blabla', '%Hblabla') StrTime;

Il est bien sûr possible d'utiliser GET\_FORMAT()  aussi avec STR\_TO\_DATE().

SELECT STR\_TO\_DATE('11.21.2011', GET\_FORMAT(*DATE*, 'USA')) AS date\_usa,

STR\_TO\_DATE('12.34.45', GET\_FORMAT(TIME, 'EUR')) AS heure\_eur,

STR\_TO\_DATE('20111027133056', GET\_FORMAT(*TIMESTAMP*, 'INTERNAL')) AS date\_heure\_int;

## 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 ; soit utiliser les fonctions SQL prévues à cet effet. 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'autres 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.

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;

**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.

-- 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;

**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 complétera cette date avec l'heure par défaut '00:00:00'.

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;

## 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 trouvé une requête faisant ce qui est demandé, mais différente de la réponse indiquée.

### Commençons par le format

##### 1. Sélectionner tous les animaux nés en juin.

SELECT id, date\_naissance, nom

FROM Animal

WHERE MONTH(date\_naissance) = 6;

##### 2. Sélectionner tous les animaux nés dans les huit premières semaines d'une année.

SELECT id, date\_naissance, nom

FROM Animal

WHERE WEEKOFYEAR(date\_naissance) < 9;

##### 3. Afficher le jour (en chiffres) et le mois de naissance (en toutes lettres) des tortues et des chats nés avant 2007 (en deux colonnes).

SELECT DAY(date\_naissance), MONTHNAME(date\_naissance)

FROM Animal

INNER JOIN Espece ON Animal.espece\_id = Espece.id

WHERE Espece.nom\_courant IN ('Chat', 'Tortue d''Hermann')

AND YEAR(date\_naissance) < 2007;

##### 4. Même chose qu'à la question précédente, mais en une seule colonne.

SELECT DATE\_FORMAT(date\_naissance, '%e %M')

FROM Animal

INNER JOIN Espece ON Animal.espece\_id = Espece.id

WHERE Espece.nom\_courant IN ('Chat', 'Tortue d''Hermann')

AND YEAR(date\_naissance) < 2007;

##### 5. Sélectionner tous les animaux nés en avril, mais pas un 24 avril, triés par heure de naissance décroissante (heure dans le sens commun du terme, donc heures, minutes, secondes) et afficher leur date de naissance au même format que dans l'exemple

**Format :** 8 janvier, à 6h30PM, en l'an 2010 après J.-C.

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) = 4

AND DAY(date\_naissance) <> 24

ORDER BY TIME(date\_naissance) DESC;

### Passons aux calculs

##### 1. Moka était censé naître le 27 février 2008. Calculer le nombre de jours de retard de sa naissance.

SELECT DATEDIFF(date\_naissance, '2008-02-27') AS retard

FROM Animal

WHERE nom = 'Moka';

##### 2. Afficher la date à laquelle chaque perroquet (espece\_id = 4) fêtera son 25e anniversaire.

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().

##### 3. Sélectionner les animaux nés dans un mois contenant exactement 29 jours.

SELECT id, date\_naissance, nom

FROM Animal

WHERE DAY(LAST\_DAY(date\_naissance)) = 29;

##### 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).

SELECT id, nom, *DATE*(DATE\_ADD(date\_naissance, INTERVAL 12 WEEK)) AS sevrage

FROM Animal

WHERE espece\_id = 2;

##### 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 né avant Zira.

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 utiliser dans la même fonction TIMESTAMPDIFF().

### 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.

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.

##### 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").

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';.

##### 3. Pour les chiens et les chats (espece\_id = 1 et espece\_id = 2 respectivement), afficher les différentes dates de naissance des portées d'au moins deux individus (format JJ/MM/AAAA), 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).

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.

##### 4. Calculer combien de chiens (espece\_id = 1) sont nés en moyenne chaque année entre 2006 et 2010 (sachant que l'on a eu au moins une naissance chaque année).

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 étape. 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.

##### 5. Afficher la date au format ISO du 5e anniversaire des animaux dont on connaît soit le père, soit la mère.

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. À nouveau, si vous n'avez pas réussi, c'est sans doute parce que vous ne décomposez pas le problème correctement.

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'achève ce cours sur les bases de MySQL. 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 toutes les possibilités de MySQL.

## Transactions

Pour commencer cette partie, nous allons voir ce que sont les transactions, à quoi elles servent exactement, et comment les utiliser avec MySQL.

Les transactions sont une fonctionnalité absolument indispensable, permettant de sécuriser une application utilisant une base de données. Sans transactions, certaines opérations risqueraient d'être à moitié réalisées, et la moindre erreur, la moindre interruption pourraient avoir des conséquences énormes. En effet, les transactions permettent de regrouper des requêtes dans des blocs, et de faire en sorte que tout le bloc soit exécuté en une seule fois, cela afin de préserver l'intégrité des données de la base.

Les transactions ont été implémentées assez tard dans MySQL, et qui plus est, elles ne sont pas utilisables pour tous les types de tables. C'est d'ailleurs un des principaux arguments des détracteurs de MySQL.

**État actuel de la base de données**  
**Note :** les tables de test ne sont pas reprises.

*SET* NAMES utf8;

DROP TABLE IF EXISTS Animal;

DROP TABLE IF EXISTS Race;

DROP TABLE IF EXISTS Espece;

CREATE TABLE Espece (

id smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

nom\_courant *varchar*(40) NOT NULL,

nom\_latin *varchar*(40) NOT NULL,

description *text*,

prix *decimal*(7,2) unsigned DEFAULT NULL,

PRIMARY KEY (id),

UNIQUE KEY nom\_latin (nom\_latin)

) ENGINE=InnoDB AUTO\_INCREMENT=6 DEFAULT CHARSET=latin1;

LOCK TABLES Espece WRITE;

INSERT INTO Espece VALUES (1,'Chien','Canis canis','Bestiole à quatre pattes qui aime les caresses et tire souvent la langue',200.00),(2,'Chat','Felis silvestris','Bestiole à quatre pattes qui saute très haut et grimpe aux arbres',150.00),(3,'Tortue d''Hermann','Testudo hermanni','Bestiole avec une carapace très dure',140.00),

(4,'Perroquet amazone','Alipiopsitta xanthops','Joli oiseau parleur vert et jaune',700.00),(5,'Rat brun','Rattus norvegicus','Petite bestiole avec de longues moustaches et une longue queue sans poils',10.00);

UNLOCK TABLES;

CREATE TABLE Race (

id smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

nom *varchar*(40) NOT NULL,

espece\_id smallint(6) unsigned NOT NULL,

description *text*,

prix *decimal*(7,2) unsigned DEFAULT NULL,

PRIMARY KEY (id)

) ENGINE=InnoDB AUTO\_INCREMENT=10 DEFAULT CHARSET=latin1;

LOCK TABLES Race WRITE;

INSERT INTO Race VALUES (1,'Berger allemand',1,'Chien sportif et élégant au pelage dense, noir-marron-fauve, noir ou gris.',485.00),(2,'Berger blanc suisse',1,'Petit chien au corps compact, avec des pattes courtes mais bien proportionnées et au pelage tricolore ou bicolore.',935.00),(3,'Singapura',2,'Chat de petite taille aux grands yeux en amandes.',985.00),

(4,'Bleu russe',2,'Chat aux yeux verts et à la robe épaisse et argentée.',835.00),(5,'Maine coon',2,'Chat de grande taille, à poils mi-longs.',735.00),(7,'Sphynx',2,'Chat sans poils.',1235.00),

(8,'Nebelung',2,'Chat bleu russe, mais avec des poils longs...',985.00),(9,'Rottweiller',1,'Chien d''apparence solide, bien musclé, à la robe noire avec des taches feu bien délimitées.',600.00);

UNLOCK TABLES;

CREATE TABLE Animal (

id smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

sexe *char*(1) DEFAULT NULL,

date\_naissance datetime NOT NULL,

nom *varchar*(30) DEFAULT NULL,

commentaires *text*,

espece\_id smallint(6) unsigned NOT NULL,

race\_id smallint(6) unsigned DEFAULT NULL,

mere\_id smallint(6) unsigned DEFAULT NULL,

pere\_id smallint(6) unsigned DEFAULT NULL,

PRIMARY KEY (id),

UNIQUE KEY ind\_uni\_nom\_espece\_id (nom,espece\_id)

) ENGINE=InnoDB AUTO\_INCREMENT=63 DEFAULT CHARSET=utf8;

LOCK TABLES Animal WRITE;

INSERT INTO Animal VALUES (1,'M','2010-04-05 13:43:00','Rox','Mordille beaucoup',1,1,18,22),(2,NULL,'2010-03-24 02:23:00','Roucky',NULL,2,NULL,40,30),(3,'F','2010-09-13 15:02:00','Schtroumpfette',NULL,2,4,41,31),

(4,'F','2009-08-03 05:12:00',NULL,'Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(5,NULL,'2010-10-03 16:44:00','Choupi','Né sans oreille gauche',2,NULL,NULL,NULL),(6,'F','2009-06-13 08:17:00','Bobosse','Carapace bizarre',3,NULL,NULL,NULL),

(7,'F','2008-12-06 05:18:00','Caroline',NULL,1,2,NULL,NULL),(8,'M','2008-09-11 15:38:00','Bagherra',NULL,2,5,NULL,NULL),(9,NULL,'2010-08-23 05:18:00',NULL,'Bestiole avec une carapace très dure',3,NULL,NULL,NULL),

(10,'M','2010-07-21 15:41:00','Bobo',NULL,1,NULL,7,21),(11,'F','2008-02-20 15:45:00','Canaille',NULL,1,NULL,NULL,NULL),(12,'F','2009-05-26 08:54:00','Cali',NULL,1,2,NULL,NULL),

(13,'F','2007-04-24 12:54:00','Rouquine',NULL,1,1,NULL,NULL),(14,'F','2009-05-26 08:56:00','Fila',NULL,1,2,NULL,NULL),(15,'F','2008-02-20 15:47:00','Anya',NULL,1,NULL,NULL,NULL),

(16,'F','2009-05-26 08:50:00','Louya',NULL,1,NULL,NULL,NULL),(17,'F','2008-03-10 13:45:00','Welva',NULL,1,NULL,NULL,NULL),(18,'F','2007-04-24 12:59:00','Zira',NULL,1,1,NULL,NULL),

(19,'F','2009-05-26 09:02:00','Java',NULL,1,2,NULL,NULL),(20,'M','2007-04-24 12:45:00','Balou',NULL,1,1,NULL,NULL),(21,'F','2008-03-10 13:43:00','Pataude',NULL,1,NULL,NULL,NULL),

(22,'M','2007-04-24 12:42:00','Bouli',NULL,1,1,NULL,NULL),(24,'M','2007-04-12 05:23:00','Cartouche',NULL,1,NULL,NULL,NULL),(25,'M','2006-05-14 15:50:00','Zambo',NULL,1,1,NULL,NULL),

(26,'M','2006-05-14 15:48:00','Samba',NULL,1,1,NULL,NULL),(27,'M','2008-03-10 13:40:00','Moka',NULL,1,NULL,NULL,NULL),(28,'M','2006-05-14 15:40:00','Pilou',NULL,1,1,NULL,NULL),

(29,'M','2009-05-14 06:30:00','Fiero',NULL,2,3,NULL,NULL),(30,'M','2007-03-12 12:05:00','Zonko',NULL,2,5,NULL,NULL),(31,'M','2008-02-20 15:45:00','Filou',NULL,2,4,NULL,NULL),

(32,'M','2009-07-26 11:52:00','Spoutnik',NULL,3,NULL,52,NULL),(33,'M','2006-05-19 16:17:00','Caribou',NULL,2,4,NULL,NULL),(34,'M','2008-04-20 03:22:00','Capou',NULL,2,5,NULL,NULL),

(35,'M','2006-05-19 16:56:00','Raccou','Pas de queue depuis la naissance',2,4,NULL,NULL),(36,'M','2009-05-14 06:42:00','Boucan',NULL,2,3,NULL,NULL),(37,'F','2006-05-19 16:06:00','Callune',NULL,2,8,NULL,NULL),

(38,'F','2009-05-14 06:45:00','Boule',NULL,2,3,NULL,NULL),(39,'F','2008-04-20 03:26:00','Zara',NULL,2,5,NULL,NULL),(40,'F','2007-03-12 12:00:00','Milla',NULL,2,5,NULL,NULL),

(41,'F','2006-05-19 15:59:00','Feta',NULL,2,4,NULL,NULL),(42,'F','2008-04-20 03:20:00','Bilba','Sourde de l''oreille droite à 80%',2,5,NULL,NULL),(43,'F','2007-03-12 11:54:00','Cracotte',NULL,2,5,NULL,NULL),

(44,'F','2006-05-19 16:16:00','Cawette',NULL,2,8,NULL,NULL),(45,'F','2007-04-01 18:17:00','Nikki','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(46,'F','2009-03-24 08:23:00','Tortilla','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),

(47,'F','2009-03-26 01:24:00','Scroupy','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(48,'F','2006-03-15 14:56:00','Lulla','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(49,'F','2008-03-15 12:02:00','Dana','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),

(50,'F','2009-05-25 19:57:00','Cheli','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(51,'F','2007-04-01 03:54:00','Chicaca','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(52,'F','2006-03-15 14:26:00','Redbul','Insomniaque',3,NULL,NULL,NULL),

(54,'M','2008-03-16 08:20:00','Bubulle','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),(55,'M','2008-03-15 18:45:00','Relou','Surpoids',3,NULL,NULL,NULL),(56,'M','2009-05-25 18:54:00','Bulbizard','Bestiole avec une carapace très dure',3,NULL,NULL,NULL),

(57,'M','2007-03-04 19:36:00','Safran','Coco veut un gâteau !',4,NULL,NULL,NULL),(58,'M','2008-02-20 02:50:00','Gingko','Coco veut un gâteau !',4,NULL,NULL,NULL),(59,'M','2009-03-26 08:28:00','Bavard','Coco veut un gâteau !',4,NULL,NULL,NULL),

(60,'F','2009-03-26 07:55:00','Parlotte','Coco veut un gâteau !',4,NULL,NULL,NULL),(61,'M','2010-11-09 00:00:00','Yoda',NULL,2,5,NULL,NULL),(62,'M','2010-11-05 00:00:00','Pipo',NULL,1,9,NULL,NULL);

UNLOCK TABLES;

ALTER TABLE Race ADD CONSTRAINT fk\_race\_espece\_id FOREIGN KEY (espece\_id) REFERENCES Espece (id) ON DELETE CASCADE;

ALTER TABLE Animal ADD CONSTRAINT fk\_race\_id FOREIGN KEY (race\_id) REFERENCES Race (id) ON DELETE *SET* NULL;

ALTER TABLE Animal ADD CONSTRAINT fk\_espece\_id FOREIGN KEY (espece\_id) REFERENCES Espece (id);

ALTER TABLE Animal ADD CONSTRAINT fk\_mere\_id FOREIGN KEY (mere\_id) REFERENCES Animal (id) ON DELETE *SET* NULL;

ALTER TABLE Animal ADD CONSTRAINT fk\_pere\_id FOREIGN KEY (pere\_id) REFERENCES Animal (id) ON DELETE *SET* NULL;

### Principe

Une transaction, c'est un **ensemble de requêtes** qui sont **exécutées en un seul bloc**. Ainsi, si une des requêtes du bloc échoue, on peut décider d'annuler tout le bloc de requêtes (ou de quand même valider les requêtes qui ont réussi).

À quoi cela sert-il ?

Imaginez que M. Durant fait un virement de 300 euros à M. Dupont via sa banque en ligne. Il remplit toutes les petites cases du virement, puis valide. L'application de la banque commence à traiter le virement quand soudain, une violente panne de courant provoque l'arrêt des serveurs de la banque.

Deux jours plus tard, M. Durant reçoit un coup de fil de M. Dupont, très énervé, qui lui demande pourquoi le paiement convenu n'a toujours pas été fait. Intrigué, M. Durant va vérifier son compte, et constate qu'il a bien été débité de 300 euros.

Mais que s'est-il donc passé ?

Normalement, le traitement d'un virement est plutôt simple, deux étapes suffisent :

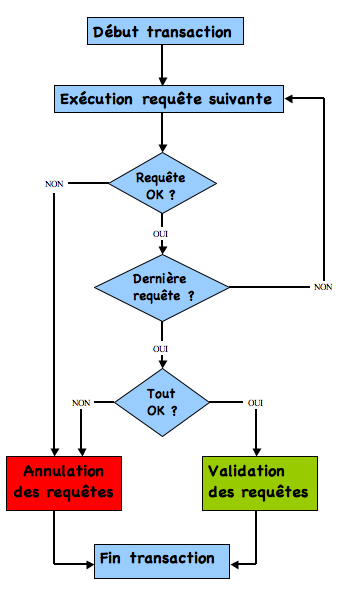
* étape 1 : on retire le montant du virement du compte du donneur d'ordre ;
* étape 2 : on ajoute le montant du virement au compte du bénéficiaire.

Seulement voilà, pas de chance pour M. Durant, la panne de courant qui a éteint les serveurs est survenue pile entre l'étape 1 et l'étape 2. Du coup, son compte a été débité, mais le compte de M. Dupont n'a jamais été crédité.

La banque de M. Durant n'utilisait pas les transactions. Si cela avait été le cas, la seconde requête du traitement n'ayant jamais été exécutée, la première requête n'aurait jamais été validée.

##### Comment se déroule une transaction ?

Voici un schéma qui devrait vous éclairer sur le principe des transactions.



* On démarre une transaction.
* On exécute les requêtes désirées une à une.
* Si une des requêtes échoue, on annule toutes les requêtes, et on termine la transaction.
* Par contre, si à la fin des requêtes, tout s'est bien passé, on valide tous les changements, et on termine la transaction.
* Si le traitement est interrompu (entre deux requêtes par exemple), les changements ne sont jamais validés, et donc les données de la base restent les mêmes qu'avant la transaction.

#### Support des transactions

Il n'est pas possible d'utiliser les transactions sur n'importe quelle table. Pour les supporter, une table doit être **transactionnelle**, ce qui, avec MySQL, est **défini par le moteur de stockage** utilisé pour la table.  
Rappelez-vous, nous avons vu, dans le [chapitre sur la création des tables](http://www.siteduzero.com/tutoriel-3-464519-creation-de-tables.html#ss_part_3), qu'il existait différents moteurs de stockage possibles avec MySQL, dont les plus connus sont **MyISAM**et **InnoDB**.  
MyISAM ne supportant pas les contraintes de clés étrangères, nos tables ont été créées avec le moteur InnoDB, ce qui tombe plutôt bien pour la suite de ce chapitre. En effet :

* les tables MyISAM sont non transactionnelles, donc ne supportent pas les transactions ;
* les tables InnoDB sont transactionnelles, donc supportent les transactions.

### Syntaxe et utilisation

##### Vocabulaire

Lorsque l'on valide les requêtes d'une transaction, on dit aussi que l'on **commite** les changements. À l'inverse, l'annulation des requêtes s'appelle un **rollback**.

##### Comportement par défaut

Vous l'aurez compris, par défaut, MySQL ne travaille pas avec les transactions. Chaque requête effectuée est directement **commitée**(validée). On ne peut pas revenir en arrière. On peut donc considérer que chaque requête constitue une transaction qui est automatiquement commitée. Par défaut, MySQL est donc en mode **"autocommit"**.

Pour quitter ce mode, il suffit de lancer la requête suivante :

*SET* autocommit=0;

Une fois que vous n'êtes plus en autocommit, chaque modification de donnée devra être commitée pour prendre effet. Tant que vos modifications ne sont pas validées, vous pouvez à tout moment les annuler (faire un rollback).

#### Valider/annuler les changements

Les commandes pour commiter et faire un rollback sont relativement faciles à retenir :

COMMIT; -- pour valider les requêtes

ROLLBACK; -- pour annuler les requêtes

##### Exemples de transactions en mode non-autocommit

Si ce n'est pas déjà fait, changez le mode par défaut de MySQL grâce à la commande que nous venons de voir.

**Première expérience :** annulation des requêtes.

Exécutez ces quelques requêtes :

INSERT INTO Animal (nom, espece\_id, date\_naissance, sexe)

VALUES ('Baba', 5, '2012-02-13 15:45:00', 'F');

INSERT INTO Animal (nom, espece\_id, date\_naissance, sexe)

VALUES ('Bibo', 5, '2012-02-13 15:48:00', 'M');

INSERT INTO Animal (nom, espece\_id, date\_naissance, sexe)

VALUES ('Buba', 5, '2012-02-13 18:32:00', 'F');

-- Insertion de 3 rats bruns

UPDATE Espece

*SET* prix = 20

WHERE id = 5;

-- Les rats bruns coûtent maintenant 20 euros au lieu de 10

Faites maintenant un SELECT  sur les tables Espece et Animal.

SELECT \*

FROM Animal

WHERE espece\_id = 5;

SELECT \*

FROM Espece

WHERE id = 5;

Les changements faits sont bien visibles. Les rats bruns valent maintenant 20 euros, et nos trois nouvelles bestioles ont bien été insérées. Cependant, un simple rollback va annuler ces changements.

ROLLBACK;

Nos rats coûtent à nouveau 10 euros, et Baba, Bibo et Buba ont disparu.

**Deuxième expérience :** interruption de la transaction.

Exécutez à nouveau les trois requêtes INSERT  et la requête UPDATE.  
Ensuite, quittez votre client MySQL (fermez simplement la fenêtre, ou tapez quit ou exit).

Reconnectez-vous et vérifiez vos données : les rats valent 10 euros, et Baba, Bibo et Buba n'existent pas. Les changements n'ont pas été commités, c'est comme s'il ne s'était rien passé !

Le mode autocommit est de nouveau activé ! Le fait de faire SET autocommit = 0;  n'est valable que pour la session courante. Or, en ouvrant une nouvelle connexion, vous avez créé une nouvelle session. Désactivez donc à nouveau ce mode.

**Troisième expérience :** validation et annulation.

Exécutez la séquence de requêtes suivante :

INSERT INTO Animal (nom, espece\_id, date\_naissance, sexe)

VALUES ('Baba', 5, '2012-02-13 15:45:00', 'F');

INSERT INTO Animal (nom, espece\_id, date\_naissance, sexe)

VALUES ('Bibo', 5, '2012-02-13 15:48:00', 'M');

INSERT INTO Animal (nom, espece\_id, date\_naissance, sexe)

VALUES ('Buba', 5, '2012-02-13 18:32:00', 'F');

-- Insertion de 3 rats bruns

COMMIT;

UPDATE Espece

*SET* prix = 20

WHERE id = 5;

-- Les rats valent 20 euros

ROLLBACK;

Si vous n'avez pas oublié de réactiver le mode non-autocommit, vous avez maintenant trois nouveaux rats bruns (les requêtes d'insertion ayant été validées), et ils ne valent toujours que 10 euros chacun (la modification de l'espèce ayant été annulée).

**Quatrième expérience :** visibilité des changements non commités.

Exécutez la requête suivante :

UPDATE Animal

*SET* commentaires = 'Queue coupée'

WHERE nom = 'Bibo' AND espece\_id = 5;

Ensuite, tout en laissant ce client MySQL ouvert, ouvrez-en un deuxième. Connectez-vous comme d'habitude à la base de données elevage. Vous avez maintenant deux sessions ouvertes, connectées à votre base de données.  
Sélectionnez les rats bruns.

SELECT id, sexe, nom, commentaires, espece\_id, race\_id

FROM Animal

WHERE espece\_id = 5;

Les commentaires de Bibo sont toujours vides. Les changements non commités ne sont donc pas visibles à l'extérieur de la transaction qui les a faits. En particulier, une autre session n'a pas accès à ces changements.

Annulez la modification de Bibo dans la première session avec un ROLLBACK. Vous pouvez fermer la seconde session.

**Démarrer explicitement une transaction**

En désactivant le mode autocommit, en réalité, on démarre une transaction. Et chaque fois que l'on fait un rollback ou un commit (ce qui met fin à la transaction), une nouvelle transaction est créée automatiquement, et ce tant que la session est ouverte.

Il est également possible de démarrer explicitement une transaction, auquel cas on peut laisser le mode autocommit activé, et décider au cas par cas des requêtes qui doivent être faites dans une transaction.

Repassons donc en mode autocommit :

*SET* autocommit=1;

Pour démarrer une transaction, il suffit de lancer la commande suivante :

START TRANSACTION;

Avec MySQL, il est également possible de démarrer une transaction avec BEGIN  ou BEGIN WORK. Cependant, il est conseillé d'utiliser plutôt START TRANSACTION, car il s'agit de la commande SQL standard.

Une fois la transaction ouverte, les requêtes devront être validées pour prendre effet. Attention au fait qu'un COMMIT  ou un ROLLBACK  met fin automatiquement à la transaction, donc les commandes suivantes seront à nouveau commitées automatiquement si une nouvelle transaction n'est pas ouverte.

**Exemples de transactions en mode autocommit**

-- Insertion d'un nouveau rat brun, plus vieux

INSERT INTO Animal (nom, espece\_id, date\_naissance, sexe)

VALUES ('Momy', 5, '2008-02-01 02:25:00', 'F');

-- Ouverture d'une transaction

START TRANSACTION;

-- La nouvelle rate est la mère de Buba et Baba

UPDATE Animal

*SET* mere\_id = LAST\_INSERT\_ID()

WHERE espece\_id = 5

AND nom IN ('Baba', 'Buba');

-- On annule les requêtes de la transaction, ce qui termine celle-ci

ROLLBACK;

-- La nouvelle rate est la mère de Bibo

UPDATE Animal

*SET* mere\_id = LAST\_INSERT\_ID()

WHERE espece\_id = 5

AND nom = 'Bibo';

-- Nouvelle transaction

START TRANSACTION;

-- Suppression de Buba

DELETE FROM Animal

WHERE espece\_id = 5

AND nom = 'Buba';

-- On valide les requêtes de la transaction, ce qui termine celle-ci

COMMIT;

Si vous avez bien suivi, vous devriez savoir les changements qui ont été faits.

* On a inséré Momy (insertion hors transaction).
* Momy n'est pas la mère de Baba (modification dans une transaction dont les requêtes ont été annulées).
* Momy est la mère de Bibo (modification hors transaction).
* Buba a été supprimée (suppression dans une transaction dont les requêtes ont été commitées).

SELECT id, nom, espece\_id, mere\_id

FROM Animal

WHERE espece\_id = 5;

#### Jalon de transaction

Lorsque l'on travaille dans une transaction et que l'on constate que certaines requêtes posent problème, on n'a pas toujours envie de faire un rollback depuis le début de la transaction, annulant toutes les requêtes alors qu'une partie aurait pu être validée.  
**Il n'est pas possible de démarrer une transaction à l'intérieur d'une transaction**. Par contre, on peut poser des **jalons de transaction**. Il s'agit de **points de repère** qui permettent d'annuler toutes les requêtes exécutées depuis ce jalon, et non toutes les requêtes de la transaction.

##### Syntaxe

Trois nouvelles commandes suffisent pour pouvoir utiliser pleinement les jalons :

SAVEPOINT nom\_jalon;

-- Crée un jalon avec comme nom "nom\_jalon"

ROLLBACK [WORK] TO [SAVEPOINT] nom\_jalon;

-- Annule les requêtes exécutées depuis le jalon "nom\_jalon", WORK et SAVEPOINT ne sont pas obligatoires

RELEASE SAVEPOINT nom\_jalon;

-- Retire le jalon "nom\_jalon" (sans annuler, ni valider les requêtes faites depuis)

**Exemple :** exécutez les requêtes suivantes.

START TRANSACTION;

INSERT INTO Animal (nom, espece\_id, date\_naissance, sexe)

VALUES ('Popi', 5, '2007-03-11 12:45:00', 'M');

SAVEPOINT jalon1;

INSERT INTO Animal (nom, espece\_id, date\_naissance, sexe)

VALUES ('Momo', 5, '2007-03-12 05:23:00', 'M');

ROLLBACK TO SAVEPOINT jalon1;

INSERT INTO Animal (nom, espece\_id, date\_naissance, sexe)

VALUES ('Mimi', 5, '2007-03-12 22:03:00', 'F');

COMMIT;

On n'utilise qu'une seule transaction, on valide à la fin, et pourtant, la seconde insertion n'a pas été faite finalement, puisqu'elle a été annulée grâce au jalon. Seuls Popi et Mimi existent.

SELECT id, sexe, date\_naissance, nom, espece\_id, mere\_id, pere\_id

FROM Animal

WHERE espece\_id = 5;

### Validation implicite et commandes non annulables

Vous savez déjà que, pour terminer une transaction, il faut utiliser les commandes COMMIT  ou ROLLBACK, selon que l'on veut valider les requêtes ou les annuler.

Ça, c'est la manière classique et recommandée. Mais il faut savoir qu'un certain nombre d'autres commandes auront aussi pour effet de clôturer une transaction. Et pas seulement de la clôturer, mais également de **valider toutes les requêtes** qui ont été faites dans cette transaction, exactement comme si vous utilisiez COMMIT.

Par ailleurs, ces commandes ne peuvent pas être annulées par un ROLLBACK.

##### Commandes DDL

Toutes les commandes qui créent, modifient, suppriment des objets dans la base de données valident implicitement les transactions.

Ces commandes forment ce que l'on appelle les requêtes DDL, pour Data Definition Langage.

Cela comprend donc :

* la création et suppression de bases de données : CREATE DATABASE, DROP DATABASE  ;
* la création, modification, suppression de tables : CREATE TABLE, ALTER TABLE, RENAME TABLE, DROP TABLE  ;
* la création, modification, suppression d'index : CREATE INDEX, DROP INDEX  ;
* la création d'objets comme les procédures stockées, les vues, etc., dont nous parlerons plus tard.

De manière générale, tout ce qui influe sur la **structure de la base de données**, et non sur les données elles-mêmes.

##### Utilisateurs

La création, la modification et la suppression d'utilisateurs (voir partie 7) provoquent aussi une validation implicite.

##### Transactions et verrous

Je vous ai signalé qu'il n'était pas possible d'imbriquer des transactions, donc d'avoir une transaction à l'intérieur d'une transaction. En fait, la commande START TRANSACTION  provoque également une validation implicite si elle est exécutée à l'intérieur d'une transaction.  
Le fait d'activer le mode autocommit (s'il n'était pas déjà activé) a le même effet.

La création et suppression de **verrous de table** clôturent aussi une transaction en la validant implicitement (voir chapitre suivant).

##### Chargement de données

Enfin, le chargement de données avec LOAD DATA  provoque également une validation implicite.

### ACID

Derrière ce titre mystérieux se cache un concept très important !

Quels sont les **critères**qu'un système utilisant les transactions doit respecter pour être **fiable**?

Il a été défini que ces critères sont au nombre de quatre : **A**tomicité, **C**ohérence, **I**solation et **D**urabilité. Soit, si l'on prend la première lettre de chaque critère : ACID.  
Voyons donc en détail ces quatre critères.

#### A pour Atomicité

Atome signifie étymologiquement **"qui ne peut être divisé"**.  
Une transaction doit être atomique, c'est-à-dire qu'elle doit former une **entité complète et indivisible**. Chaque élément de la transaction, chaque requête effectuée, ne peut exister que dans la transaction.

Si l'on reprend l'exemple du virement bancaire, en utilisant les transactions, les deux étapes (débit du compte donneur d'ordre, crédit du compte bénéficiaire) ne peuvent exister indépendamment l'une de l'autre. Si l'une est exécutée, l'autre doit l'être également. Il s'agit d'un tout.

Peut-on dire que nos transactions sont atomiques ?

Oui. Si une transaction en cours est interrompue, aucune des requêtes exécutées ne sera validée. De même, en cas d'erreur, il suffit de faire un ROLLBACK  pour annuler toute la transaction. Et si tout se passe bien, un COMMIT  validera l'intégralité de la transaction en une fois.

#### C pour cohérence

Les données doivent rester **cohérentes dans tous les cas :** que la transaction se termine sans encombre, qu'une erreur survienne, ou que la transaction soit interrompue. Un virement dont seule l'étape de débit du donneur d'ordre est exécutée produit des données incohérentes (la disparition de 300 euros jamais arrivés chez le bénéficiaire). Avec une transaction, cette incohérence n'apparaît jamais. Tant que la totalité des étapes n'a pas été réalisée avec succès, les données restent dans leur état initial.

Nos transactions permettent-elles d'assurer la cohérence des données ?

Oui, les changements de données ne sont validés qu'une fois que toutes les étapes ont été exécutées. De l'extérieur de la transaction, le moment entre les deux étapes d'un virement n'est jamais visible.

#### I pour Isolation

Chaque transaction doit être isolée, donc **ne pas interagir avec une autre transaction**.

Nos transactions sont-elles isolées ?

##### Test

Dans votre client MySQL, exécutez les requêtes suivantes (ne commitez pas) pour modifier le pere\_id du rat Bibo :

Copiez-collez tout le bloc dans votre client MySQL.

START TRANSACTION; -- On ouvre une transaction

UPDATE Animal -- On modifie Bibo

*SET* pere\_id = 73

WHERE espece\_id = 5 AND nom = 'Bibo';

SELECT id, nom, commentaires, pere\_id, mere\_id

FROM Animal

WHERE espece\_id = 5;

À nouveau, ouvrez une deuxième session, tout en laissant la première ouverte (démarrez un deuxième client SQL et connectez-vous à votre base de données).  
Exécutez les requêtes suivantes, pour modifier les commentaires de Bibo.

À nouveau, prenez bien tout le bloc d'un coup, vous suivrez plus facilement les explications qui suivent.

START TRANSACTION; -- On ouvre une transaction

SELECT id, nom, commentaires, pere\_id, mere\_id

FROM Animal

WHERE espece\_id = 5;

UPDATE Animal -- On modifie la perruche Bibo

*SET* commentaires = 'Agressif'

WHERE espece\_id = 5 AND nom = 'Bibo';

SELECT id, nom, commentaires, pere\_id, mere\_id

FROM Animal

WHERE espece\_id = 5;

Le résultat n'est pas du tout le même dans les deux sessions. En effet, dans la première, on a la confirmation que la requête UPDATE  a été effectuée :

Query OK, 1 row affected (0.00 sec)

Rows matched: 1  Changed: 1  Warnings: 0

Et le SELECT  renvoie bien les données modifiées (pere\_id n'est plus NULL  pour Bibo) :

Par contre, dans la deuxième session, on a d'abord fait un SELECT, et Bibo n'a toujours pas de père (puisque cela n'a pas été commité dans la première session). Donc on s'attendrait à ce que la requête UPDATE  laisse pere\_id à NULL  et modifie commentaires.

Seulement voilà, la requête UPDATE  ne fait rien ! La session semble bloquée : pas de message de confirmation après la requête UPDATE, et le second SELECT  n'a pas été effectué.

mysql>

mysql> UPDATE Animal       -- On modifie Bibo

    -> SET commentaires = 'Agressif'

    -> WHERE espece\_id = 5 AND nom = 'Bibo';

\_

Commitez maintenant les changements dans la première session (celle qui n'est pas bloquée). Retournez voir dans la seconde session ; elle s'est débloquée et indique maintenant un message de confirmation aussi :

Query OK, 1 row affected (5.17 sec)

Rows matched: 1  Changed: 1  Warnings: 0

Qui plus est, le SELECT  a été exécuté (vous devrez peut-être appuyer sur Entrée pour que ce soit envoyé au serveur) et les modifications ayant été faites par la session 1 ont été prises en compte : *commentaires*vaut 'Agressif'  et *pere\_id*vaut 73  !

Il est possible que votre seconde session indique ceci : ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction. Cela signifie que la session est restée bloquée trop longtemps et que par conséquent la transaction a été automatiquement fermée (avec un rollback des requêtes effectuées). Dans ce cas, recommencez l'opération.

Il n'y a plus qu'à commiter les changements faits par la deuxième session, et c'est terminé ! Si vous ne commitez pas, commentaires restera NULL. Par contre, pere\_id vaudra toujours 73, puisque ce changement-là a été commité par la première session.

##### Conclusion

La deuxième session n'a pas interagi avec les changements faits par la première session, chaque transaction est bien **isolée**.

Et la première session qui bloque la seconde, ce n'est pas une interaction, ça ?

Pas dans le cadre des critères ACID. Oui, la première session provoque un retard dans l'exécution des requêtes de la deuxième session, mais les critères de fiabilité que nous examinons ici concernent **les données impactées par les transactions**, et non le déroulement de celles-ci (qui importe peu, finalement).  
Ce blocage a pour effet d'empêcher la deuxième session d'écraser un changement fait par la première. Donc, ce blocage a bien pour effet l'isolation des transactions.

##### Verrous

Le blocage de la deuxième session vient de ce que la première session, en faisant sa requête UPDATE, a **automatiquement posé un verrou** sur la ligne contenant Bobi le rat, empêchant toute modification tant que la transaction était en cours. Les verrous faisant l'objet du prochain chapitre, je n'en dis pas plus pour l'instant.

##### Utilité

Je vous l'accorde, vous n'allez pas vous amuser tous les jours à ouvrir deux sessions MySQL. Par contre, pour une application pouvant être utilisée par plusieurs personnes en même temps (qui toutes travaillent sur la même base de données), il est impératif que ce critère soit respecté.  
Prenons l'exemple simple d'un jeu par navigateur : de nombreux joueurs peuvent être connectés en même temps, et effectuer des opérations différentes. Si les transactions ne sont pas isolées, une partie des actions des joueurs risquerait de se voir annuler. On isole donc les transactions grâce aux verrous (qui sont ici automatiquement posés, mais ce n'est pas toujours le cas).

#### D pour Durabilité

Une fois la transaction terminée, les données résultant de cette transaction doivent être **stockées de manière durable** et pouvoir être récupérées, en cas de crash du serveur, par exemple.

Nos transactions modifient-elles les données de manière durable ?

Oui, une fois les changements commités, ils sont stockés définitivement (jusqu'à modification par une nouvelle transaction).

## Verrous

Complément indispensable des transactions, les verrous permettent de **sécuriser les requêtes** en bloquant ponctuellement et partiellement l'accès aux données.

Il s'agit d'un gros chapitre, avec beaucoup d'informations. Il y a par conséquent un maximum d'exemples pour vous aider à comprendre le comportement des verrous selon les situations.

Au sommaire de ce chapitre :

* Qu'est-ce qu'un verrou ?
* Quel est le comportement par défaut de MySQL par rapport aux verrous ?
* Quand et comment poser un verrou de table ?
* Quand et comment poser un verrou de ligne ?
* Comment modifier le comportement par défaut de MySQL ?

### Principe

Lorsqu'une session MySQL pose un verrou sur un élément de la base de données, cela veut dire qu'elle **restreint, voire interdit, l'accès à cet élément aux autres sessions MySQL** qui voudraient y accéder.

#### Verrous de table et verrous de ligne

Il est possible de poser un **verrou sur une table entière**, ou **seulement sur une ou plusieurs lignes d'une table**. Étant donné qu'un verrou empêche l'accès d'autres sessions, il est en général plus intéressant de poser un verrou sur la plus petite partie de la base possible.

Par exemple, si l'on travaille avec les chiens de la table Animal.

* On peut poser un verrou sur toute la table Animal. Dans ce cas, les autres sessions n'auront pas accès à cette table, tant que le verrou sera posé. Qu'elles veuillent en utiliser les chiens, les chats, ou autre, tout leur sera refusé.
* On peut aussi poser un verrou uniquement sur les lignes de la table qui contiennent des chiens. De cette manière, les autres sessions pourront accéder aux chats, aux perroquets, etc. Elles pourront toujours travailler, tant qu'elles n'utilisent pas les chiens.

Cette notion d'accès simultané aux données par plusieurs sessions différentes s'appelle la **concurrence**. Plus la concurrence est possible, donc plus le nombre de sessions pouvant accéder aux données simultanément est grand, mieux c'est.  
En effet, prenons l'exemple d'un site web. En général, on préfère permettre à plusieurs utilisateurs de surfer en même temps, sans devoir attendre entre chaque action de pouvoir accéder aux informations chacun à son tour. Or, chaque utilisateur crée une session chaque fois qu'il se connecte à la base de données (pour lire les informations ou les modifier).  
Préférez donc (autant que possible) les verrous de ligne aux verrous de table !

#### Avertissements

Les informations données dans ce chapitre **concernent exclusivement MySQL**, et en particulier les tables utilisant **les moteurs MyISAM ou InnoDB** (selon le type de verrou utilisé). En effet, les verrous sont implémentés différemment selon les SGBD, et même selon le moteur de table en ce qui concerne MySQL. Si le principe général reste toujours le même, certains comportements et certaines options peuvent différer d'une implémentation à l'autre. N'hésitez pas à vous renseigner plus avant.

Par ailleurs, je vous présente ici les principes généraux et les principales options, mais il faut savoir qu'il y a énormément à dire sur les verrous, et que j'ai donc dû faire un sérieux tri des informations avant de rédiger ce chapitre. À nouveau, en cas de doute, ou si vous avez besoin d'informations précises, je vous conseille vraiment de consulter la documentation officielle (si possible en anglais, car elle est infiniment plus complète qu'en français).

Enfin, dernier avertissement : de nombreux changements dans l'implémentation des verrous sont advenus lors du développement des dernières versions de MySQL. Aussi, la différence entre les verrous dans la version 5.0 et la version 5.5 est assez importante. Tout ce que je présente dans ce chapitre est valable à partir de la version 5.5. Vérifiez bien votre version, et si vous consultez la documentation officielle, prenez bien celle qui concerne votre propre version.

#### Modification de notre base de données

Nous allons ajouter deux tables à notre base de données, afin d'illustrer au mieux l'intérêt et l'utilisation des verrous : une table Client, qui contiendra les coordonnées des clients de notre élevage, et une table Adoption, qui contiendra les renseignements concernant les adoptions faites par nos clients.  
Dorénavant, certains animaux présents dans notre table Animal ne seront plus disponibles, car ils auront été adoptés. Nous les garderons cependant dans notre base de données. Avant toute adoption, il nous faudra donc vérifier la disponibilité de l'animal.

Voici les requêtes à effectuer pour faire ces changements.

-- Table Client

CREATE TABLE Client (

id SMALLINT UNSIGNED AUTO\_INCREMENT NOT NULL,

nom *VARCHAR*(100) NOT NULL,

prenom *VARCHAR*(60) NOT NULL,

adresse *VARCHAR*(200),

code\_postal *VARCHAR*(6),

ville *VARCHAR*(60),

pays *VARCHAR*(60),

email VARBINARY(100),

PRIMARY KEY (id),

UNIQUE INDEX ind\_uni\_email (email)

) ENGINE = InnoDB;

-- Table Adoption

CREATE TABLE Adoption (

client\_id SMALLINT UNSIGNED NOT NULL,

animal\_id SMALLINT UNSIGNED NOT NULL,

date\_reservation *DATE* NOT NULL,

date\_adoption *DATE*,

prix *DECIMAL*(7,2) UNSIGNED NOT NULL,

paye TINYINT(1) NOT NULL DEFAULT 0,

PRIMARY KEY (client\_id, animal\_id),

CONSTRAINT fk\_client\_id FOREIGN KEY (client\_id) REFERENCES Client(id),

CONSTRAINT fk\_adoption\_animal\_id FOREIGN KEY (animal\_id) REFERENCES Animal(id),

UNIQUE INDEX ind\_uni\_animal\_id (animal\_id)

) ENGINE = InnoDB;

-- Insertion de quelques clients

INSERT INTO Client (prenom, nom, adresse, code\_postal, ville, pays, email) VALUES ('Jean', 'Dupont', 'Rue du Centre, 5', '45810', 'Houtsiplou', 'France', 'jean.dupont@email.com');

INSERT INTO Client (prenom, nom, adresse, code\_postal, ville, pays, email) VALUES ('Marie', 'Boudur', 'Place de la Gare, 2', '35840', 'Troudumonde', 'France', 'marie.boudur@email.com');

INSERT INTO Client (prenom, nom, adresse, code\_postal, ville, pays, email) VALUES ('Fleur', 'Trachon', 'Rue haute, 54b', '3250', 'Belville', 'Belgique', 'fleurtrachon@email.com');

INSERT INTO Client (prenom, nom, adresse, code\_postal, ville, pays, email) VALUES ('Julien', 'Van Piperseel', NULL, NULL, NULL, NULL, 'jeanvp@email.com');

INSERT INTO Client (prenom, nom, adresse, code\_postal, ville, pays, email) VALUES ('Johan', 'Nouvel', NULL, NULL, NULL, NULL, 'johanetpirlouit@email.com');

INSERT INTO Client (prenom, nom, adresse, code\_postal, ville, pays, email) VALUES ('Frank', 'Germain', NULL, NULL, NULL, NULL, 'francoisgermain@email.com');

INSERT INTO Client (prenom, nom, adresse, code\_postal, ville, pays, email) VALUES ('Maximilien', 'Antoine', 'Rue Moineau, 123', '4580', 'Trocoul', 'Belgique', 'max.antoine@email.com');

INSERT INTO Client (prenom, nom, adresse, code\_postal, ville, pays, email) VALUES ('Hector', 'Di Paolo', NULL, NULL, NULL, NULL, 'hectordipao@email.com');

INSERT INTO Client (prenom, nom, adresse, code\_postal, ville, pays, email) VALUES ('Anaelle', 'Corduro', NULL, NULL, NULL, NULL, 'ana.corduro@email.com');

INSERT INTO Client (prenom, nom, adresse, code\_postal, ville, pays, email) VALUES ('Eline', 'Faluche', 'Avenue circulaire, 7', '45870', 'Garduche', 'France', 'elinefaluche@email.com');

INSERT INTO Client (prenom, nom, adresse, code\_postal, ville, pays, email) VALUES ('Carine', 'Penni', 'Boulevard Haussman, 85', '1514', 'Plasse', 'Suisse', 'cpenni@email.com');

INSERT INTO Client (prenom, nom, adresse, code\_postal, ville, pays, email) VALUES ('Virginie', 'Broussaille', 'Rue du Fleuve, 18', '45810', 'Houtsiplou', 'France', 'vibrousaille@email.com');

INSERT INTO Client (prenom, nom, adresse, code\_postal, ville, pays, email) VALUES ('Hannah', 'Durant', 'Rue des Pendus, 66', '1514', 'Plasse', 'Suisse', 'hhdurant@email.com');

INSERT INTO Client (prenom, nom, adresse, code\_postal, ville, pays, email) VALUES ('Elodie', 'Delfour', 'Rue de Flore, 1', '3250', 'Belville', 'Belgique', 'e.delfour@email.com');

INSERT INTO Client (prenom, nom, adresse, code\_postal, ville, pays, email) VALUES ('Joel', 'Kestau', NULL, NULL, NULL, NULL, 'joel.kestau@email.com');

-- Insertion de quelques adoptions

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (1, 39, '2008-08-17', '2008-08-17', 735.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (1, 40, '2008-08-17', '2008-08-17', 735.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (2, 18, '2008-06-04', '2008-06-04', 485.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (3, 27, '2009-11-17', '2009-11-17', 200.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (4, 26, '2007-02-21', '2007-02-21', 485.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (4, 41, '2007-02-21', '2007-02-21', 835.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (5, 21, '2009-03-08', '2009-03-08', 200.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (6, 16, '2010-01-27', '2010-01-27', 200.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (7, 5, '2011-04-05', '2011-04-05', 150.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (8, 42, '2008-08-16', '2008-08-16', 735.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (9, 55, '2011-02-13', '2011-02-13', 140.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (9, 54, '2011-02-13', '2011-02-13', 140.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (10, 49, '2010-08-17', '2010-08-17', 140.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (11, 62, '2011-03-01', '2011-03-01', 630.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (12, 69, '2007-09-20', '2007-09-20', 10.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (13, 57, '2012-01-10', '2012-01-10', 700.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (14, 58, '2012-02-25', '2012-02-25', 700.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (15, 30, '2008-08-17', '2008-08-17', 735.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (11, 32, '2008-08-17', '2010-03-09', 140.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (9, 33, '2007-02-11', '2007-02-11', 835.00, 1);

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, date\_adoption, prix, paye) VALUES (2, 3, '2011-03-12', '2011-03-12', 835.00, 1);

La table Adoption ne contient pas de colonne id auto-incrémentée. Par contre, on a bien défini une clé primaire, mais une clé primaire **composite**(sur plusieurs colonnes). En effet, une adoption est définie par un client adoptant un animal. Il n'est pas nécessaire d'ajouter une colonne supplémentaire pour définir individuellement chaque ligne ; le couple (client\_id, animal\_id) fait très bien l'affaire (il est composé de deux SMALLINT, donc les recherches sur cette clé seront rapides).  
Notez que nous définissons également un index UNIQUE  sur la colonne animal\_id. Par conséquent, on aurait même pu définir directement animal\_id comme étant la clé primaire. Je trouvais cependant plus logique d'inclure le client dans la définition d'une adoption. C'est un choix plutôt arbitraire qui a surtout comme avantage de vous montrer un exemple de clé composite.

### Syntaxe et utilisation : verrous de table

Les verrous de table sont **les seuls supportés par MyISAM**. Ils sont d'ailleurs principalement utilisés pour pallier en partie l'absence de transactions dans MyISAM.  
Les tables InnoDB peuvent également utiliser ce type de verrou.

Pour verrouiller une table, il faut utiliser la commande LOCK TABLES  :

LOCK TABLES nom\_table [AS alias\_table] [READ | WRITE] [, ...];

* En utilisant READ, un verrou de lecture sera posé ; c'est-à-dire que les autres sessions pourront toujours lire les données des tables verrouillées, mais ne pourront plus les modifier.
* En utilisant WRITE, un verrou d'écriture sera posé. Les autres sessions ne pourront plus ni lire ni modifier les données des tables verrouillées.

Pour déverrouiller les tables, on utilise UNLOCK TABLES. Cela déverrouille toutes les tables verrouillées. Il n'est pas possible de préciser les tables à déverrouiller. Tous les verrous de table d'une session sont relâchés en même temps.

##### Session ayant obtenu le verrou

Lorsqu'une session acquiert un ou plusieurs verrous de table, cela a plusieurs conséquences pour cette session :

* elle ne peut plus accéder **qu'aux tables sur lesquelles elle a posé un verrou ;**
* elle ne peut accéder à ces tables qu'**en utilisant les noms qu'elle a donnés** lors du verrouillage (soit le nom de la table, soit le/les alias donné(s)) ;
* s'il s'agit d'un **verrou de lecture** (READ), elle peut **uniquement lire les données**, pas les modifier.

**Exemples :** on pose deux verrous, l'unREAD, l'autreWRITE, l'un en donnant un alias au nom de la table, l'autre sans.

LOCK TABLES Espece READ,

-- On pose un verrou de lecture sur Espece

Adoption AS adopt WRITE;

-- et un verrou d'écriture sur Adoption avec l'alias adopt

Voyons maintenant le résultat de ces différentes requêtes.

**1.** Sélection dans Espece, sans alias.

SELECT id, nom\_courant FROM Espece;

**2.** Sélection dans *Espece*, avec alias.

SELECT id, nom\_courant

FROM Espece AS table\_espece;

ERROR 1100 (HY000): Table 'table\_espece' was not locked with LOCK TABLES

Par contre, si l'on essaye d'utiliser un alias, cela ne fonctionne pas. Le verrou est posé sur *Espece*, pas sur *Espece*AS*table\_espece*.

**3.** Modification dans *Espece*, sans alias.

UPDATE Espece

*SET* description = 'Petit piaf bruyant'

WHERE id = 4;

ERROR 1099 (HY000): Table 'Espece' was locked with a READ lock and can't be updated

Avec ou sans alias, impossible de modifier la table *Espece*, puisque le verrou que l'on possède dessus est un verrou de lecture.

**4.** Sélection dans *Adoption*, sans alias.

SELECT client\_id, animal\_id

FROM Adoption;

ERROR 1100 (HY000): Table 'Adoption' was not locked with LOCK TABLES

Cette fois, c'est le contraire ; sans alias, cela ne passe pas.

**5.** Sélection dans *Adoption*, avec alias.

SELECT client\_id, animal\_id

FROM Adoption AS adopt

WHERE client\_id = 4;

**6.** Modification dans *Adoption*, sans alias.

UPDATE Adoption

*SET* paye = 0

WHERE client\_id = 10 AND animal\_id = 49;

ERROR 1100 (HY000): Table 'Adoption' was not locked with LOCK TABLES

Idem pour la modification, l'alias est indispensable.

**7.** Modification dans *Adoption*, avec alias.

UPDATE Adoption AS adopt

*SET* paye = 0

WHERE client\_id = 10 AND animal\_id = 49;

Query OK, 1 row affected (0.03 sec)

Il faut donc penser à acquérir tous les verrous nécessaires aux requêtes à exécuter. De plus, il faut les obtenir **en une seule requête**LOCK TABLES. En effet, LOCK TABLES  commence par enlever tous les verrous de table de la session avant d'en acquérir de nouveaux.  
Il est bien entendu possible de poser plusieurs verrous sur la même table en une seule requête afin de verrouiller son nom ainsi qu'un ou plusieurs alias.

**Exemples :** on pose un verrou de lecture sur *Adoption*, puis avec une seconde requête, on pose deux verrous de lecture sur la table *Espece*, l'un avec alias, l'autre sans.

UNLOCK TABLES; -- On relâche d'abord les deux verrous précédents

LOCK TABLES Adoption READ;

LOCK TABLES Espece READ, Espece AS table\_espece READ;

Une fois ces deux requêtes effectuées, nous aurons donc bien deux verrous de lecture sur la table *Espece* : un avec son nom, l'autre avec un alias. Par contre, le verrou sur *Adoption* n'existera plus puisqu'il aura été relâché par l'exécution de la seconde requête LOCK TABLES.

**1.** Sélection dans *Espece*, sans alias.

SELECT id, nom\_courant FROM Espece;

**2.** Sélection dans *Espece*, avec alias.

SELECT id, nom\_courant FROM Espece AS table\_espece;

**3.** Sélection dans *Espece*, avec mauvais alias.

SELECT id, nom\_courant FROM Espece AS table\_esp;

ERROR 1100 (HY000): Table 'table\_esp' was not locked with LOCK TABLES

Bien entendu, cela ne fonctionne que pour l'alias que l'on a donné lors du verrouillage.

**4.** Sélection dans *Adoption*, sans alias.

SELECT \* FROM Adoption;

ERROR 1100 (HY000): Table 'Adoption' was not locked with LOCK TABLES

Le verrou sur *Adoption* a été relâché lorsque l'on a posé les verrous sur *Espece*. On ne peut donc pas lire les données d'*Adoption* (avec ou sans alias).

**Conséquences pour les autres sessions**

Si une session a obtenu un**verrou de lecture** sur une table, les autres sessions :

* peuvent lire les données de la table ;
* peuvent également acquérir un verrou de lecture sur cette table ;
* ne peuvent pas modifier les données ni acquérir un verrou d'écriture sur cette table.

Si par contre une session a obtenu un **verrou d'écriture**, les autres sessions ne peuvent absolument pas accéder à cette table tant que ce verrou existe.

**Exemples :** ouvrez un deuxième client MySQL et connectez-vous à votre base de données afin d'avoir deux sessions ouvertes.

**1.** Sélection sur des tables verrouillées à partir d'une autre session.

Session 1 :

LOCK TABLES Client READ,

-- Verrou de lecture sur Client

Adoption WRITE;

-- Verrou d'écriture sur Adoption

Session 2 :

SELECT id, nom, prenom, ville, email

FROM Client

WHERE ville = 'Houtsiplou';

La sélection sur Client se fait sans problème.

Session 2 :

SELECT \*

FROM Adoption

WHERE client\_id = 4;

Par contre, la sélection sur Adoption ne passe pas. La session se bloque, jusqu'à ce que la session 1 déverrouille les tables avec UNLOCK TABLES.

**2.** Modification sur des tables verrouillées à partir d'une autre session.

Reverrouillez les tables avec la session 1 :

LOCK TABLES Client READ,

-- Verrou de lecture sur Client

Adoption WRITE;

-- Verrou d'écriture sur Adoption

Session 2 :

UPDATE Client

*SET* pays = 'Suisse'

WHERE id = 5;

La modification sur Client, contrairement à la sélection, est bloquée jusqu'au déverrouillage. Déverrouillez, puis verrouillez à nouveau avec la session 1.

Session 2 :

UPDATE Adoption

*SET* paye = 1

WHERE client\_id = 3;

Bien entendu, la modification sur la table Adoption attend également que les verrous soient relâchés par la session 1.

En ce qui concerne la pose de verrous de table par les autres sessions, faites vos propres tests. Simplement, si une session peut lire les données d'une table, elle peut également poser un verrou de lecture ; si une session peut modifier les données d'une table, elle peut également poser un verrou d'écriture.

##### Interaction avec les transactions

Si l'on utilise des tables MyISAM, il n'y a évidemment aucune précaution particulière à prendre par rapport aux transactions lorsque l'on utilise des verrous de table (les tables MyISAM étant non transactionnelles). Par contre, si l'on utilise des tables InnoDB, il convient d'être prudent. En effet :

* START TRANSACTION  ôte les verrous de table ;
* les commandes LOCK TABLES  et UNLOCK TABLES  provoquent une validation implicite si elles sont exécutées à l'intérieur d'une transaction.

Pour utiliser à la fois les transactions et les verrous de table, il faut renoncer à démarrer explicitement les transactions, et donc utiliser le mode non-autocommit. Lorsque l'on est dans ce mode, il est facile de contourner la validation implicite provoquée par LOCK TABLES  et UNLOCK TABLES  : il suffit d'appeler LOCK TABLES  avant toute modification de données, et de commiter/annuler les modifications avant d'exécuter UNLOCK TABLES.

**Exemple :**

*SET* autocommit = 0;

LOCK TABLES Adoption WRITE;

-- La validation implicite ne commite rien puisque aucun changement n'a été fait

UPDATE Adoption *SET* date\_adoption = NOW() WHERE client\_id = 9 AND animal\_id = 54;

SELECT client\_id, animal\_id, date\_adoption FROM Adoption WHERE client\_id = 9;

ROLLBACK;

UNLOCK TABLES;

-- On a annulé les changements juste avant donc la validation implicite n'a aucune conséquence

SELECT client\_id, animal\_id, date\_adoption FROM Adoption WHERE client\_id = 9;

*SET* autocommit = 1;

### Syntaxe et utilisation : verrous de ligne

Ces verrous ne peuvent pas être posés sur une table utilisant le moteur MyISAM ! Tout ce qui est dit ici concerne les tables **InnoDB**uniquement.

Comme les verrous de table, les verrous de ligne peuvent être de deux types :

* **Les verrous partagés :** permettent aux autres sessions de lire les données, mais pas de les modifier (équivalents aux verrous de table de lecture) ;
* **Les verrous exclusifs :** ne permettent ni la lecture ni la modification des données (équivalents aux verrous d'écriture).

#### Requêtes de modification, insertion et suppression

* Les requêtes de **modification et suppression** des données **posent automatiquement un verrou exclusif sur les lignes concernées**, à savoir les lignes sélectionnées par la clause WHERE, ou toutes les lignes s'il n'y a pas de clause WHERE  (ou sur les colonnes utilisées s'il n'y a pas d'index,  comme nous verrons plus loin).
* Les requêtes d'insertion quant à elles posent un **verrou exclusif sur la ligne insérée**.

#### Requêtes de sélection

Les requêtes de sélection, par défaut, ne posent pas de verrous. Il faut donc en poser explicitement au besoin.

##### Verrou partagé

Pour poser un verrou partagé, on utilise LOCK IN SHARE MODE  à la fin de la requête SELECT.

SELECT \* FROM Animal WHERE espece\_id = 5 LOCK IN SHARE MODE;

Cette requête pose donc un verrou partagé sur les lignes de la table Animal pour lesquelles espece\_id vaut 5.

Ce verrou signifie en fait, pour les autres sessions : "Je suis en train de lire ces données. Vous pouvez venir les lire aussi, mais pas les modifier tant que je n'ai pas terminé."

##### Verrou exclusif

Pour poser un verrou exclusif, on utilise FOR UPDATE  à la fin de la requête SELECT.

SELECT \* FROM Animal WHERE espece\_id = 5 FOR UPDATE;

Ce verrou signifie aux autres sessions : "Je suis en train de lire ces données dans le but probable de faire une modification. Ne les lisez pas avant que j'aie fini (et bien sûr, ne les modifiez pas)."

#### Transactions et fin d'un verrou de ligne

Les verrous de ligne ne sont donc pas posés par des commandes spécifiques, mais par des requêtes de sélection, d'insertion ou de modification.  
Ces verrous existent donc uniquement tant que la requête qui les a posés interagit avec les données.

Par conséquent, ce type de verrou s'utilise en conjonction avec les transactions. En effet, hors transaction, dès qu'une requête est lancée, elle est effectuée et les éventuelles modifications des données sont immédiatement validées.  
Par contre, dans le cas d'une requête faite dans une transaction, les changements ne sont pas validés tant que la transaction n'a pas été commitée. Donc, à partir du moment où une requête a été exécutée dans une transaction, et jusqu'à la fin de la transaction (COMMIT  ou ROLLBACK), la requête a **potentiellement** un effet sur les données.  
C'est à ce moment-là (quand une requête a été exécutée, mais pas validée ou annulée) qu'il est intéressant de verrouiller les données qui vont **potentiellement être modifiées** (ou supprimées) par la transaction.

Un verrou de ligne est donc lié à la transaction dans laquelle il est posé. Dès que l'on fait un COMMIT  ou un ROLLBACK  de la transaction, le verrou est levé.

#### Exemples

##### Verrou posé par une requête de modification

Session 1 :

START TRANSACTION;

UPDATE Client *SET* pays = 'Suisse'

WHERE id = 8;

-- un verrou exclusif sera posé sur la ligne avec id = 8

Session 2 :

START TRANSACTION;

SELECT \* FROM Client

WHERE id = 8; -- pas de verrou

SELECT \* FROM Client

WHERE id = 8

LOCK IN SHARE MODE; -- on essaye de poser un verrou partagé

La première session a donc posé un verrou exclusif automatiquement en faisant un UPDATE.

La seconde session fait d'abord une simple sélection, sans poser de verrou. Pas de problème, la requête passe.

Ah ? La requête passe ? Et c'est normal ? Et le verrou exclusif alors ?

Oui, c'est normal et c'est important de comprendre pourquoi.  
En fait, lorsqu'une session **démarre une transaction**, **elle prend en quelque sorte une photo des tables dans leur état actuel** (les modifications non commitées n'étant pas visibles). La transaction va alors travailler sur la base de cette photo, tant que l'on ne lui demande pas d'aller vérifier que les données n'ont pas changé. Donc le SELECT  ne voit pas les changements, et ne se heurte pas au verrou, puisque celui-ci est posé sur les lignes de la table, et non pas sur la photo de cette table que détient la session.

Et comment fait-on pour demander à la session d'actualiser sa photo ?

On lui demande de poser un verrou ! Lorsqu'une session **pose un verrou** sur une table, elle est obligée de travailler vraiment avec la table, et pas sur sa photo. Elle va donc aller chercher les dernières infos disponibles, et actualiser sa photo par la même occasion.  
On le voit bien avec la seconde requête, qui tente de poser un verrou partagé (qui vise donc uniquement la lecture). Elle va d'abord chercher les lignes les plus à jour et tomber sur le verrou posé par la première session ; elle se retrouve alors bloquée jusqu'à ce que la première session ôte le verrou exclusif.

Session 1 :

COMMIT;

En committant les changements de la session 1, le verrou exclusif posé par la requête de modification est relâché. La session 2 est donc libre de poser à son tour un verrou partagé.

On peut également essayer la même manœuvre, avec cette fois-ci un UPDATE  plutôt qu'un SELECT ... LOCK IN SHARE MODE  (donc une requête qui va tenter de poser un verrou exclusif plutôt qu'un verrou partagé).

Session 1 :

START TRANSACTION;

UPDATE Adoption *SET* paye = 0

WHERE client\_id = 11;

Session 2 :

START TRANSACTION;

UPDATE Adoption *SET* paye = 1

WHERE animal\_id = 32;

-- l'animal 32 a été adopté par le client 11

Comme prévu, la seconde session est bloquée, jusqu'à ce que la première session termine sa transaction. Validez la transaction de la première session, puis de la seconde. Le comportement sera le même si la deuxième session fait un DELETE  sur les lignes verrouillées, ou un SELECT ... FOR UPDATE.

##### Verrou posé par une requête d'insertion

Session 1 :

START TRANSACTION;

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, prix)

VALUES (12, 75, NOW(), 10.00);

Session 2 :

SELECT \* FROM Adoption

WHERE client\_id > 13

LOCK IN SHARE MODE;

SELECT \* FROM Adoption

WHERE client\_id < 13

LOCK IN SHARE MODE;

La première session insère une adoption pour le client 12 et pose un verrou exclusif sur cette ligne.  
La seconde session fait deux requêtes SELECT  en posant un verrou partagé : l'une qui sélectionne les adoptions des clients avec un id supérieur à 13 ; l'autre qui sélectionne les adoptions des clients avec un id inférieur à 13.  
Seule la seconde requête SELECT  se heurte au verrou posé par la première session, puisqu'elle tente de récupérer notamment les adoptions du client 12, dont une est verrouillée.

Dès que la session 1 commite l'insertion, la sélection se fait dans la session 2.

Session 1 :

COMMIT;

##### Verrou posé par une requête de sélection

Voyons d'abord le comportement d'un verrou partagé, posé par SELECT ... LOCK IN SHARE MODE.

Session 1 :

START TRANSACTION;

SELECT \* FROM Client

WHERE id < 5

LOCK IN SHARE MODE;

Session 2 :

START TRANSACTION;

SELECT \* FROM Client

WHERE id BETWEEN 3 AND 8;

SELECT \* FROM Client

WHERE id BETWEEN 3 AND 8

LOCK IN SHARE MODE;

SELECT \* FROM Client

WHERE id BETWEEN 3 AND 8

FOR UPDATE;

La première session pose un verrou partagé sur les clients 1, 2, 3 et 4.  
La seconde session fait trois requêtes de sélection. Toutes les trois concernent les clients 3 à 8 (dont les deux premiers sont verrouillés).

* Requête 1 : ne pose aucun verrou (travaille sur une "photo" de la table et pas sur les vraies données) donc s'effectue sans souci.
* Requête 2 : pose un verrou partagé, ce qui est faisable sur une ligne verrouillée par un verrou partagé. Elle s'effectue également.
* Requête 3 : tente de poser un verrou exclusif, ce qui lui est refusé.

Bien entendu, des requêtes UPDATE  ou DELETE  (posant des verrous exclusifs) faites par la deuxième session se verraient, elles aussi, bloquées.

Terminez les transactions des deux sessions (par un rollback ou un commit).

Quant aux requêtes SELECT ... FOR UPDATE  posant un verrou exclusif, elles provoqueront exactement les mêmes effets qu'une requête UPDATE  ou DELETE  (après tout, un verrou exclusif, c'est un verrou exclusif).

Session 1 :

START TRANSACTION;

SELECT \* FROM Client

WHERE id < 5

FOR UPDATE;

Session 2 :

START TRANSACTION;

SELECT \* FROM Client

WHERE id BETWEEN 3 AND 8;

SELECT \* FROM Client

WHERE id BETWEEN 3 AND 8

LOCK IN SHARE MODE;

Cette fois-ci, même la requête SELECT ... LOCK IN SHARE MODE  de la seconde session est bloquée (comme le serait une requête SELECT ... FOR UPDATE, ou une requête UPDATE, ou une requête DELETE).

#### En résumé

* On pose un verrou partagé lorsque l'on fait une requête dans le but de lire des données.
* On pose un verrou exclusif lorsque l'on fait une requête dans le but (immédiat ou non) de modifier des données.
* Un verrou partagé sur les lignes x va permettre aux autres sessions d'obtenir également un verrou partagé sur les lignes x, mais pas d'obtenir un verrou exclusif.
* Un verrou exclusif sur les lignes x va empêcher les autres sessions d'obtenir un verrou sur les lignes x, qu'il soit partagé ou exclusif.

En fait, ils portent plutôt bien leurs noms ces verrous !

#### Rôle des index

Tentons une nouvelle expérience.

Session 1 :

START TRANSACTION;

UPDATE Animal

*SET* commentaires = CONCAT\_WS(' ', 'Animal fondateur.', commentaires) -- On ajoute une phrase de commentaire

WHERE date\_naissance < '2007-01-01'; -- à tous les animaux nés avant 2007

Session 2 :

START TRANSACTION;

UPDATE Animal

*SET* commentaires = 'Aveugle' -- On modifie les commentaires

WHERE date\_naissance = '2008-03-10 13:40:00'; -- De l'animal né le 10 mars 2008 à 13h40

Dans la session 1, on fait un UPDATE  sur les animaux nés avant 2007. On s'attend donc à pouvoir utiliser les animaux nés après dans une autre session, puisque InnoDB pose des verrous sur les lignes et pas sur toute la table.  
Pourtant, la session 2 semble bloquée lorsque l'on fait un UPDATE  sur un animal né en 2008.  
Faites un rollback sur la session 1 ; ceci débloque la session 2. Annulez également la requête de cette session.

Ce comportement est donc en contradiction avec ce que l'on obtenait précédemment. Quelle est la différence ?

Le sous-titre vous a évidemment soufflé la réponse : la différence se trouve au niveau des index. Voyons donc ça !  
Voici une commande qui va vous afficher les index présents sur la table Animal :

SHOW INDEX FROM Animal;

Une partie des colonnes du résultat montré ici a été retirée pour des raisons de clarté.

Nous avons donc des index sur les colonnes suivantes : id, nom, mere\_id, pere\_id, espece\_id et race\_id.  
Mais aucun index sur la colonne date\_naissance.

Il semblerait donc que, lorsque l'on pose un verrou, avec dans la clause WHERE  de la requête une colonne indexée (espece\_id), le verrou soit bien posé uniquement sur les lignes pour lesquelles espece\_id vaut la valeur recherchée.  
Par contre, si, dans la clause WHERE, on utilise une colonne non indexée (date\_naissance), MySQL n'est pas capable de déterminer quelles lignes doivent être bloquées, donc on se retrouve avec toutes les lignes bloquées.

Pourquoi faut-il un index pour pouvoir poser un verrou efficacement ?

C'est très simple ! Vous savez que, lorsqu'une colonne est indexée (que ce soit un index simple, unique, ou une clé primaire ou étrangère), MySQL stocke les valeurs de cette colonne **en les triant**. Donc lors d'une recherche sur l'index, pas besoin de parcourir toutes les lignes, il peut utiliser des algorithmes de recherche performants et trouver facilement les lignes concernées.  
S'il n'y a pas d'index, par contre, toutes les lignes doivent être parcourues chaque fois que la recherche est faite, et il n'y a donc pas moyen de verrouiller simplement une partie de l'index (donc une partie des lignes). Dans ce cas, MySQL verrouille toutes les lignes.

Cela fait une bonne raison de plus de mettre des index sur les colonnes qui servent fréquemment dans vos clauses WHERE  !

Encore une petite expérience pour illustrer le rôle des index dans le verrouillage de lignes :

Session 1 :

START TRANSACTION;

UPDATE Animal -- Modification de tous les rats

*SET* commentaires = CONCAT\_WS(' ', 'Très intelligent.', commentaires)

WHERE espece\_id = 5;

Session 2 :

START TRANSACTION;

UPDATE Animal

*SET* commentaires = 'Aveugle'

WHERE id = 34; -- Modification de l'animal 34 (un chat)

UPDATE Animal

*SET* commentaires = 'Aveugle'

WHERE id = 72; -- Modification de l'animal 72 (un rat)

La session 1 se sert de l'index sur espece\_id pour verrouiller les lignes contenant des rats bruns. Pendant ce temps, la session 2 veut modifier deux animaux : un chat et un rat, en se basant sur leur id.  
La modification du chat se fait sans problème ; par contre, la modification du rat est bloquée tant que la transaction de la session 1 est ouverte.  
Faites un rollback des deux transactions.

On peut conclure de cette expérience que, bien que MySQL utilise les index pour verrouiller les lignes, il n'est pas nécessaire d'utiliser le même index pour avoir des accès concurrents.

#### Lignes fantômes et index de clé suivante

Qu'est-ce qu'une ligne fantôme ?

Dans une session, démarrons une transaction et sélectionnons toutes les adoptions faites par les clients dont l'id dépasse 13, avec un verrou exclusif.

Session 1 :

START TRANSACTION;

SELECT \* FROM Adoption WHERE client\_id > 13 FOR UPDATE;

-- ne pas oublier le FOR UPDATE pour poser le verrou

La requête va poser un verrou exclusif sur toutes les lignes dont client\_id vaut 14 ou plus.

Imaginons maintenant qu'une seconde session démarre une transaction à ce moment-là, insère et commite une ligne dans Adoption pour le client 15.  
Si, par la suite, la première session refait la même requête de sélection avec verrou exclusif, elle va faire apparaître une troisième ligne de résultat : l'adoption nouvellement insérée (étant donné que pour poser le verrou, la session va aller chercher les données les plus à jour, prenant en compte le commit de la seconde session).

Cette ligne nouvellement apparue malgré les verrous est une "ligne fantôme".

Pour pallier ce problème, qui est contraire au principe d'isolation, **les verrous posés par des requêtes de lecture, de modification et de suppression sont des verrous dits "de clé suivante" :** ils empêchent l'insertion d'une ligne dans les espaces entre les lignes verrouillées, ainsi que dans l'espace juste après les lignes verrouillées.

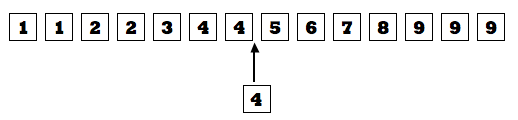
L'espace entre ? L'espace juste après ?

Nous avons vu que les verrous se basent sur les index pour verrouiller uniquement les lignes nécessaires. Voici un petit schéma qui vous expliquera ce qu'est cet "index de clé suivante".

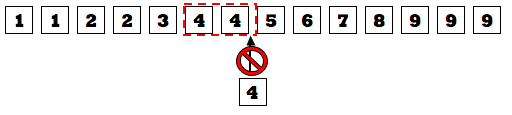
On peut représenter l'index sur client\_id de la table Adoption de la manière suivante (je ne mets que les client\_id < 10) :

Représentation de l'index sur client_idReprésentation de l'index sur client\_id

Si l'on insère une adoption avec 4 pour client\_id, l'index va être réorganisé de la manière suivante :

Insertion d'une nouvelle valeur dans l'index

Mais, si l'on pose un verrou de clé suivante sur l'index, sur les lignes dont client\_id vaut 4, on va alors verrouiller **les lignes, les espaces entre les lignes et les espaces juste après**. Ceci va bloquer l'insertion de la nouvelle ligne :

Verrou de clé suivante

**Démonstration**

On a toujours un verrou exclusif (grâce à notre SELECT ... FOR UPDATE) sur les client\_id supérieurs à 14 dans la session 1 (sinon, reposez-le).

Session 2 :

START TRANSACTION;

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, prix)

VALUES (15, 61, NOW(), 735.00);

L'insertion est bloquée ! Pas de risque de voir apparaître une ligne fantôme.  
Annulez les deux transactions.

##### Exception

Si la clause WHERE  concerne un index UNIQUE  (cela inclut bien sûr les clés primaires) et recherche une seule valeur (exemple : WHERE id = 4), alors seule la ligne concernée (si elle existe) est verrouillée, et pas l'espace juste après dans l'index. Forcément, s'il s'agit d'un index UNIQUE, l'insertion d'une nouvelle valeur ne changera rien : WHERE id = 4  ne renverra jamais qu'une seule ligne.

#### Pourquoi poser un verrou exclusif avec une requête SELECT ?

Après tout, une requête SELECT  ne fait jamais que lire des données. Que personne ne puisse les modifier pendant que l'on est en train de les lire, c'est tout à fait compréhensible. Mais pourquoi carrément interdire aux autres de les lire aussi ?

Tout simplement parce que certaines données sont lues dans le but prévisible et avoué de les modifier immédiatement après.

L'exemple typique est la vérification de stock dans un magasin (ou dans un élevage d'animaux).  
Un client arrive et veut adopter un chat, on vérifie donc les chats disponibles pour l'adoption, en posant un verrou partagé :

Session 1 :

START TRANSACTION;

SELECT Animal.id, Animal.nom, Animal.date\_naissance, Race.nom as race, COALESCE(Race.prix, Espece.prix) as prix

FROM Animal

INNER JOIN Espece ON Animal.espece\_id = Espece.id

LEFT JOIN Race ON Animal.race\_id = Race.id

-- Jointure externe, on ne veut pas que les chats de race

WHERE Espece.nom\_courant = 'Chat'

-- Uniquement les chats...

AND Animal.id NOT IN (SELECT animal\_id FROM Adoption)

-- ... qui n'ont pas encore été adoptés

LOCK IN SHARE MODE;

Je rappelle que la fonction COALESCE()  prend un nombre illimité de paramètres, et renvoie le premier paramètre non NULL  qu'elle rencontre. Donc ici, s'il s'agit d'un chat de race, *Race.prix* ne sera pas NULL  et sera donc renvoyé. Par contre, s'il n'y a pas de race, *Race.prix* sera NULL, mais pas *Espece.prix*, qui sera alors sélectionné.

Si, pendant que le premier client fait son choix, un second client arrive, voulant adopter un chat maine coon, il va également chercher la liste des chats disponibles. Et vu que l'on travaille pour l'instant en verrous partagés, il va pouvoir l'obtenir.

Session 2 :

START TRANSACTION;

SELECT Animal.id, Animal.nom, Animal.date\_naissance, Race.nom as race, COALESCE(Race.prix, Espece.prix) as prix

FROM Animal

INNER JOIN Espece ON Animal.espece\_id = Espece.id

INNER JOIN Race ON Animal.race\_id = Race.id

-- Jointure interne cette fois

WHERE Race.nom = 'Maine Coon'

-- Uniquement les Maine Coon...

AND Animal.id NOT IN (SELECT animal\_id FROM Adoption)

-- ... qui n'ont pas encore été adoptés

LOCK IN SHARE MODE;

C'est alors que le premier client, M. Dupont, décide de craquer pour Bagherra.

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, prix, paye)

SELECT id, 8, NOW(), 735.00, 1

FROM Client

WHERE email = 'jean.dupont@email.com';

COMMIT;

Et M. Durant jette également son dévolu sur Bagherra (qui est décidément très très mignon) !

INSERT INTO Client (nom, prenom, email)

VALUES ('Durant', 'Philippe', 'phidu@email.com');

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, prix, paye)

VALUES (LAST\_INSERT\_ID(), 8, NOW(), 735.00, 0);

L'insertion dans Client fonctionne, mais l'insertion dans Adoption pose problème :

ERROR 1062 (23000): Duplicate entry '8' for key 'ind\_uni\_animal\_id'

Et pour cause : Bagherra vient d'être adopté, à l'instant.  
Furieux, M. Durant s'en va, et l'élevage a perdu un client. Il ne reste plus qu'à annuler sa transaction.

C'est pour éviter ce genre de situation qu'il vaut parfois mieux mettre un verrou exclusif sur une sélection. Si l'on sait que cette sélection sert à déterminer quels changements vont être faits, ce n'est pas la peine de laisser quelqu'un d'autre lire des informations qui cesseront d'être justes incessamment sous peu.

### Niveaux d'isolation

Nous avons vu que, par défaut :

* lorsque l'on démarre une transaction, la session prend une photo des tables, et travaille uniquement sur cette photo (donc sur des données potentiellement périmées) tant qu'elle ne pose pas un verrou ;
* les requêtes SELECT  ne posent pas de verrous si l'on ne le demande pas explicitement ;
* les requêtes SELECT ... LOCK IN SHARE MODE, SELECT ... FOR UPDATE, DELETE  et UPDATE  posent un verrou de clé suivante (sauf dans le cas d'une recherche sur index unique, avec une valeur unique).

Ce comportement est **défini par le niveau d'isolation** des transactions.

#### Syntaxe

Pour définir le niveau d'isolation des transactions, on utilise la requête suivante :

*SET* [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }

* Le mot-clé GLOBAL  définit le niveau d'isolation pour toutes les sessions MySQL qui seront créées dans le futur. Les sessions existantes ne sont pas affectées.
* SESSION  définit le niveau d'isolation pour la session courante.
* Si l'on ne précise ni GLOBAL  ni SESSION, le niveau d'isolation défini ne concernera que la prochaine transaction que l'on ouvrira dans la session courante.

#### Les différents niveaux

##### REPEATABLE READ

Il s'agit du **niveau par défaut**, celui avec lequel vous travaillez depuis le début. Repeatable read signifie "lecture répétable", c'est-à-dire que, si l'on fait plusieurs requêtes de sélection **(non verrouillantes)** à la suite, elles donneront toujours le même résultat, quels que soient les changements effectués par d'autres sessions.  
Si l'on pense à bien utiliser les verrous là où c'est nécessaire, c'est un niveau d'isolation tout à fait suffisant.

##### READ COMMITTED

Avec ce niveau d'isolation, chaque requête SELECT  (non verrouillante) va reprendre une "photo" à jour de la base de données, même si plusieurs SELECT  se font dans la même transaction. Ainsi, un SELECT  verra toujours les derniers changements commités, même s'ils ont été faits dans une autre session, après le début de la transaction.

##### READ UNCOMMITTED

Le niveau READ UNCOMMITTED  fonctionne comme READ COMMITTED, si ce n'est qu'il autorise la "lecture sale". C'est-à-dire qu'une session sera capable de lire des changements encore non commités par d'autres sessions.

**Exemple**

Session 1 :

START TRANSACTION;

UPDATE Race

*SET* prix = 0

WHERE id = 7;

Session 2 :

*SET* TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

START TRANSACTION;

SELECT id, nom, espece\_id, prix FROM Race;

La modification faite par la session 1 n'a pas été commitée. Elle ne sera donc potentiellement jamais validée, auquel cas elle n'affectera jamais les données.  
Pourtant, la session 2 voit ce changement de données non commitées. "Lecture sale" n'a pas une connotation négative par hasard, bien entendu ! Aussi, évitez de travailler avec ce niveau d'isolation.  
Annulez la modification de données réalisée par la session 1 et terminez la transaction de la seconde session.

##### SERIALIZABLE

Ce niveau d'isolation se comporte comme REPEATABLE READ, sauf que, lorsque le mode autocommit est désactivé, tous les SELECT  simples sont implicitement convertis en SELECT ... LOCK IN SHARE MODE.

## Requêtes préparées

Après les verrous et les transactions, voici une troisième notion importante pour la sécurisation des requêtes : les requêtes préparées.

Une requête préparée, c'est en quelque sorte **une requête stockée en mémoire** (pour la session courante), et **que l'on peut exécuter** à loisir. Mais avant d'entrer dans le vif du sujet, nous allons faire un détour par **les variables utilisateur**, qui sont indispensables aux requêtes préparées.

Dans ce chapitre, nous apprendrons donc :

* ce que sont les variables utilisateur et comment les définir ;
* ce qu'est une requête préparée ;
* comment créer, exécuter et supprimer une requête préparée ;
* de quelle manière utiliser les requêtes préparées ;
* en quoi les requêtes préparées sont utiles pour la sécurisation d'une application ;
* comment et dans quel cas on peut gagner en performance en utilisant les requêtes préparées.

### Variables utilisateur

#### Définitions

##### Variable

Une variable est une **information stockée**, constituée d'un **nom**et d'une **valeur**.  
On peut par exemple avoir la variable age (son nom) ayant pour valeur 24.

##### Variable utilisateur

Une variable utilisateur est une variable **définie par l'utilisateur**. Les variables utilisateur MySQL doivent toujours être **précédées du signe @**.

Une variable utilisateur peut contenir quatre types de valeur :

* un entier (ex. : 24) ;
* un réel (ex. : 7,8) ;
* une chaîne de caractères (ex. : 'Hello World !') ;
* une chaîne binaire, auquel cas il faut faire précéder la chaîne du caractère b (ex. : b'011001'). Nous n'en parlerons pas dans ce cours.

Pour les noms de vos variables utilisateur utilisez uniquement des lettres, des chiffres, des \_, des $ et des .. Attention au fait que les noms des variables ne sont pas sensibles à la casse : @A  est la même variable utilisateur que @a.

Il existe également ce que l'on appelle des **variables système**, qui sont des variables prédéfinies par MySQL, et des variables locales, que nous verrons avec les procédures stockées.

#### Créer et modifier une variable utilisateur

##### SET

La manière la plus classique de créer ou de modifier une variable utilisateur est d'utiliser la commande SET.

*SET* @age = 24;

-- Ne pas oublier le @

*SET* @salut = 'Hello World !', @poids = 7.8;

-- On peut créer plusieurs variables en même temps

Si la variable utilisateur existe déjà, sa valeur sera modifiée, sinon, elle sera créée.

SELECT @age, @poids, @salut;

##### Opérateur d'assignation

Il est également possible d'assigner une valeur à une variable utilisateur directement dans une requête, en utilisant l'opérateur d'assignation **:=** (n'oubliez pas les : sinon il s'agit de l'opérateur de comparaison de valeurs).

SELECT @age := 32, @poids := 48.15, @perroquet := 4;

On peut utiliser l'opérateur d'assignation **:=** dans une commande SET  également : SET @chat := 2;

#### Utilisation d'une variable utilisateur

##### Ce que l'on peut faire

Une fois votre variable utilisateur créée, vous pouvez bien sûr l'afficher avec SELECT. Mais vous pouvez également l'utiliser dans des requêtes ou faire des calculs avec.

SELECT id, sexe, nom, commentaires, espece\_id

FROM Animal

WHERE espece\_id = @perroquet;

-- On sélectionne les perroquets

*SET* @conversionDollar = 1.31564;

-- On crée une variable contenant le taux de conversion des euros en dollars

SELECT prix AS prix\_en\_euros,

-- On sélectionne le prix des races, en euros et en dollars.

ROUND(prix \* @conversionDollar, 2) AS prix\_en\_dollars,

-- En arrondissant à deux décimales

nom FROM Race;

Si vous utilisez une variable utilisateur qui n'a pas été définie, vous n'obtiendrez aucune erreur. Simplement, la variable aura comme valeur NULL.

##### Ce que l'on ne peut pas faire

Avec les variables utilisateur, on peut donc dynamiser un peu nos requêtes. Cependant, il n'est pas possible d'utiliser les variables utilisateur pour stocker un nom de table ou de colonne que l'on introduirait ensuite directement dans la requête. Ni pour stocker une partie de commande SQL. **Les variables utilisateur stockent des données**.

**Exemples**

*SET* @table\_clients = 'Client';

SELECT \* FROM @table\_clients;

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '@table\_clients' at line 1

*SET* @colonnes = 'id, nom, description';

SELECT @colonnes FROM Race WHERE espece\_id = 1;

C'est logique, dans une requête, les noms de tables/colonnes et les commandes SQL ne peuvent pas être représentées comme des chaînes de caractères : on ne les entoure pas de guillemets.

#### Portée des variables utilisateur

Une variable utilisateur n'existe que pour la session dans laquelle elle a été définie. Lorsque vous vous déconnectez, toutes vos variables utilisateur sont automatiquement réinitialisées. De plus, deux sessions différentes ne partagent pas leurs variables utilisateur.

**Exemple**

Session 1:

*SET* @essai = 3;

Session 2:

SELECT @essai;

De même, si vous assignez une valeur à @essai dans la session 2, @essai vaudra toujours 3 pour la session 1.

### Principe et syntaxe des requêtes préparées

#### Principe

Une requête préparée, c'est en fait un **modèle de requête** que l'on enregistre et auquel on donne un nom. On va ensuite pouvoir l'exécuter en l'appelant grâce à son nom, et en lui passant éventuellement un ou plusieurs paramètres.

Par exemple, si vous avez régulièrement besoin d'aller chercher des informations sur vos clients à partir de leur adresse e-mail dans une session, plutôt que de faire :

SELECT \* FROM Client WHERE email = 'truc@email.com';

SELECT \* FROM Client WHERE email = 'machin@email.com';

SELECT \* FROM Client WHERE email = 'bazar@email.com';

SELECT \* FROM Client WHERE email = 'brol@email.com';

Vous pouvez préparer une requête modèle :

SELECT \* FROM Client WHERE email = ?

Où le ? représente un paramètre. Ensuite, il suffit de l'appeler par son nom en lui passant 'truc@email.com', ou 'machin@email.com', selon les besoins du moment.

Bien entendu, on parle d'un cas où l'on n'a qu'une seule adresse e-mail à la fois. Typiquement, c'est le cas où l'on s'occupe d'un client à la fois. Sinon, bien entendu, une simple clause email IN ('truc@email.com', 'machin@email.com', 'bazar@email.com', 'brol@email.com')suffirait.

##### Portée

Tout comme les variables utilisateur, **une requête préparée n'existe que pour la session qui la crée**.

#### Syntaxe

Voyons comment faire tout cela !

##### Préparation d'une requête

Pour préparer une requête, il faut renseigner deux éléments :

* le nom que l'on donne à la requête préparée ;
* la requête elle-même, avec un ou plusieurs paramètres (représentés par un ?).

Voici la syntaxe à utiliser :

PREPARE nom\_requete

FROM 'requete\_preparable';

**Exemples**

-- Sans paramètre

PREPARE select\_race

FROM 'SELECT \* FROM Race';

-- Avec un paramètre

PREPARE select\_client

FROM 'SELECT \* FROM Client WHERE email = ?';

-- Avec deux paramètres

PREPARE select\_adoption

FROM 'SELECT \* FROM Adoption WHERE client\_id = ? AND animal\_id = ?';

Plusieurs choses importantes :

* Le nom de la requête préparée ne doit pas être entre guillemets. Par contre, la requête à préparer doit l'être. **La requête à préparer doit être passée comme une chaîne de caractères**.
* Que le paramètre soit un nombre (client\_id = ?) ou une chaîne de caractères (email = ?), cela ne change rien. **On ne met pas de guillemets autour du ?** à l'intérieur de la requête à préparer.
* La chaîne de caractères contenant la requête à préparer ne peut contenir qu'**une seule requête** (et non plusieurs séparées par un ;).
* **Les paramètres ne peuvent représenter que des données, des valeurs**, pas des noms de tables ou de colonnes ni des morceaux de commandes SQL.

Comme la requête à préparer est donnée sous forme de chaîne de caractères, il est également possible d'utiliser une variable utilisateur dans laquelle on enregistre tout ou partie de la requête à préparer.

**Exemples**

*SET* @req = 'SELECT \* FROM Race';

PREPARE select\_race

FROM @req;

*SET* @colonne = 'nom';

*SET* @req\_animal = CONCAT('SELECT ', @colonne, ' FROM Animal WHERE id = ?');

PREPARE select\_col\_animal

FROM @req\_animal;

Par contre, il n'est pas possible de mettre directement la fonction CONCAT()  dans la clause FROM.

Si vous donnez à une requête préparée le nom d'une requête préparée déjà existante, cette dernière sera supprimée et remplacée par la nouvelle.

##### Exécution d'une requête préparée

Pour exécuter une requête préparée, on utilise la commande suivante :

EXECUTE nom\_requete [USING @parametre1, @parametre2, ...];

Si la requête préparée contient un ou plusieurs paramètres, on doit leur donner une valeur avec la clause USING, **en utilisant une variable utilisateur**. Il n'est pas possible de donner directement une valeur. Par ailleurs, il faut donner exactement autant de variables utilisateur qu'il y a de paramètres dans la requête.

**Exemples**

EXECUTE select\_race;

*SET* @id = 3;

EXECUTE select\_col\_animal USING @id;

*SET* @client = 2;

EXECUTE select\_adoption USING @client, @id;

*SET* @email = 'jean.dupont@email.com';

EXECUTE select\_client USING @email;

*SET* @email = 'marie.boudur@email.com';

EXECUTE select\_client USING @email;

*SET* @email = 'fleurtrachon@email.com';

EXECUTE select\_client USING @email;

*SET* @email = 'jeanvp@email.com';

EXECUTE select\_client USING @email;

*SET* @email = 'johanetpirlouit@email.com';

EXECUTE select\_client USING @email;

##### Suppression d'une requête préparée

Pour supprimer une requête préparée, on utilise DEALLOCATE PREPARE, suivi du nom de la requête préparée.

**Exemple**

DEALLOCATE PREPARE select\_race;

### Usage et utilité

#### Usage

La syntaxe que nous venons de voir, avec PREPARE, EXECUTE  et DEALLOCATE  est en fait très rarement utilisée.  
En effet, vous le savez, MySQL est rarement utilisé seul. La plupart du temps, il est utilisé en conjonction avec un langage de programmation, comme Java, PHP, Python, etc. Celui-ci permet de gérer un programme, un site web… et crée des requêtes SQL permettant de gérer la base de données de l'application.  
Or, il existe des API (interfaces de programmation) pour plusieurs langages, qui permettent de faire des requêtes préparées sans exécuter vous-mêmes les commandes SQL PREPARE, EXECUTE  et DEALLOCATE. Exemples : l'API C MySQL pour le langage C, MySQL Connector/J pour le Java, ou MySQL Connector/Net pour le .Net.

Voici un exemple de code C utilisant l'API MySQL pour préparer et exécuter une requête d'insertion :

MYSQL\_STMT \*req\_prep;

MYSQL\_BIND param[3];

*int* client = 2;

*int* animal = 56;

MYSQL\_TIME date\_reserv;

*char* \*req\_texte = "INSERT INTO Adoption (client\_id, animal\_id, date\_reservation) VALUES (?, ?, ?)";

// On prépare la requête

if (mysql\_stmt\_prepare(req\_prep, req\_texte, strlen(req\_texte)) != 0)

{

printf("Impossible de préparer la requête");

exit(0);

}

// On initialise un tableau (param, qui contiendra les paramètres) à 0

memset((*void*\*) param, 0, sizeof(param));

// On définit le premier paramètre (pour client\_id)

param[0].buffer\_type = MYSQL\_TYPE\_INT;

param[0].buffer = (*void*\*) &client;

param[0].is\_unsigned = 0;

param[0].is\_null = 0;

// On définit le deuxième paramètre (pour animal\_id)

param[1].buffer\_type = MYSQL\_TYPE\_INT;

param[1].buffer = (*void*\*) &animal;

param[1].is\_unsigned = 0;

param[1].is\_null = 0;

// On définit le troisième paramètre (pour date\_reservation)

param[2].buffer\_type = MYSQL\_TYPE\_DATE;

param[2].buffer = (*void*\*) &date\_reserv;

param[2].is\_null = 0;

// On lie les paramètres

if (mysql\_stmt\_bind\_param(req\_prep, param) != 0)

{

printf("Impossible de lier les paramètres à la requête");

exit(0);

}

// On définit la date

date\_reserv.year = 2012;

date\_reserv.month = 3;

date\_reserv.day = 20;

// On exécute la requête

if (mysql\_stmt\_execute(req\_prep) != 0)

{

printf("Impossible d'exécuter la requête");

exit(0);

}

Pour d'autres langages, des extensions ont été créées, en général elles-mêmes basées sur l'API C MySQL, comme PDO pour le PHP.

Exemple de code PHP utilisant l'extension PDO pour préparer et exécuter une requête de sélection :

<?php

try

{

$email = 'jean.dupont@email.com';

// On se connecte

$bdd = new *PDO*('mysql:host=localhost;dbname=elevage', 'sdz', '', array(*PDO*::ATTR\_ERRMODE => *PDO*::ERRMODE\_EXCEPTION ));

// On prépare la requête

$requete = $bdd->prepare("SELECT \* FROM Client WHERE email = :email");

// On lie la variable $email définie au-dessus au paramètre :email de la requête préparée

$requete->bindValue(':email', $email, *PDO*::PARAM\_STR);

//On exécute la requête

$requete->execute();

// On récupère le résultat

if ($requete->fetch())

{

echo 'Le client existe !';

}

} catch (Exception $e)

{

die('Erreur : ' . $e->getMessage());

}

Pourquoi nous avoir fait apprendre la syntaxe SQL si l'on ne s'en sert jamais ?

D'abord, parce qu'il est toujours intéressant de savoir comment fonctionnent les requêtes préparées.  
Ensuite, parce qu'il pourrait arriver qu'il n'existe aucune API ni extension permettant de faire des requêtes préparées pour le langage dans lequel vous programmez, auquel cas, bien entendu, il faudrait construire vos requêtes préparées vous-même.  
Ou vous pourriez tomber sur l'un des rares cas où il vous serait nécessaire de préparer une requête directement en SQL.  
Enfin, vous aurez peut-être besoin de faire quelques tests impliquant des requêtes préparées directement dans MySQL.

Cependant, si une API ou une extension existe et répond à vos besoins, utilisez-la ! Elle sera généralement plus performante et plus sécurisée que ce que vous pourriez faire vous-même.

#### Utilité

Les requêtes préparées sont principalement utilisées pour deux raisons :

* protéger son application des injections SQL ;
* gagner en performance dans le cas d'une requête exécutée plusieurs fois par la même session.

##### Empêcher les injections SQL

En général, quand on crée une application, l'utilisateur peut interagir avec celle-ci. L'utilisateur peut créer un membre sur un site web communautaire, un personnage sur un jeu vidéo, etc. Les actions de l'utilisateur vont donc avoir une incidence sur la base de données de l'application. Il va envoyer certaines informations, qui vont être traitées, puis une partie va être envoyée sous forme de requêtes à la base de données.  
Il existe un adage bien connu en programmation : "Never trust user input", traduit en français par "Ne jamais faire confiance aux données fournies par l'utilisateur".

Lorsque l'on traite des données qui viennent de l'extérieur, il est absolument impératif de toujours vérifier celles-ci, et de protéger les requêtes construites à partir de ces données. Ceci évite que l'utilisateur, volontairement ou non, fasse ce que l'on appelle une injection SQL et provoque un comportement inattendu et souvent indésirable, voire dangereux, pour les données.

Les injections SQL sont **un**type de failles exploitables par l'utilisateur. Il existe de nombreux autres types de failles.

Passons maintenant à la question qui doit vous brûler les lèvres.

Mais qu'est-ce qu'une injection SQL ?

On appelle injection SQL le fait qu'un utilisateur fournisse des données contenant des mots-clés SQL ou des caractères particuliers qui vont**détourner ou modifier le comportement des requêtes construites** sur la base de ces données.

Imaginons que vous créiez un site web avec un espace membre.  
Vos membres ont accès à une page "profil" grâce à laquelle ils peuvent gérer leurs informations, ou supprimer leur compte. Pour supprimer leur compte, ils ont simplement à appuyer sur un bouton qui envoie leur numéro d'id.

D'un côté, on a donc une requête incomplète :

DELETE FROM Membre WHERE id =

De l'autre, l'id du client, par exemple 4.

Avec votre langage de programmation web préféré, vous mettez les deux ensemble pour créer la requête complète :

DELETE FROM Membre WHERE id = 4;

Et maintenant, un méchant pirate s'amuse à modifier la donnée envoyée par le bouton "Supprimer compte" (oui oui, c'est faisable, et même plutôt facile).  
À la suite du numéro d'id, il ajoute OR 1 = 1. Après construction de la requête, on obtient donc ceci :

DELETE FROM Membre WHERE id = 4 OR 1 = 1;

Et la seconde condition étant toujours remplie, c'est l'horreur : toute votre table Membre est effacée !  
Et voilà ! Vous avez été victime d'une injection SQL.

Comment une requête préparée peut-elle éviter les injections SQL ?

En utilisant les requêtes préparées, lorsque vous liez une valeur à un paramètre de la requête préparée grâce à la fonction correspondante dans le langage que vous utilisez, le type du paramètre attendu est également donné, explicitement ou implicitement. La plupart du temps, soit une erreur sera générée si la donnée de l'utilisateur ne correspond pas au type attendu, soit la donnée de l'utilisateur sera rendue inoffensive (par l'ajout de guillemets qui en feront une simple chaîne de caractères, par exemple).  
Par ailleurs, lorsque MySQL injecte les valeurs dans la requête, les mots-clés SQL qui s'y trouvent pour une raison ou une autre ne sont pas interprétés (puisque les paramètres n'acceptent que des valeurs, et pas des morceaux de requêtes).

##### Gagner en performance

Lorsque l'on exécute une simple requête (sans la préparer), voici les étapes qui sont effectuées :

1. La requête est envoyée vers le serveur.
2. La requête est compilée par le serveur (traduite du langage SQL compréhensible pour nous, pauvres humains limités, vers un "langage machine" dont se sert le serveur).
3. Le serveur crée un plan d'exécution de la requête (quelles tables utiliser ? quels index ? dans quel ordre ?…).
4. Le serveur exécute la requête.
5. Le résultat de la requête est renvoyé au client.

Voici maintenant les étapes lorsqu'il s'agit d'une requête préparée :

**Préparation**de la requête :

1. La requête est envoyée vers le serveur, avec un identifiant.
2. La requête est compilée par le serveur.
3. Le serveur crée un plan d'exécution de la requête.
4. La requête compilée et son plan d'exécution sont stockés en mémoire par le serveur.
5. Le serveur envoie vers le client une confirmation que la requête est prête (en se servant de l’identifiant de la requête).

**Exécution**de la requête :

1. L'identifiant de la requête à exécuter ainsi que les paramètres à utiliser sont envoyés vers le serveur.
2. Le serveur exécute la requête demandée.
3. Le résultat de la requête est renvoyé au client.

On a donc plus d'étapes pour une requête préparée que pour une requête non préparée (8 contre 5). Du moins, lorsque l'on exécute une seule fois la requête. Car si l'on exécute plusieurs fois une requête, la tendance s'inverse rapidement : dans le cas d'une requête non préparée,**toutes les étapes doivent être répétées à chaque fois** (sachant que la création du plan d'exécution est l'étape la plus longue), alors que, pour une requête préparée, **seules les étapes d'exécution seront répétées**.

Il est donc intéressant, du point de vue des performances, de préparer une requête lorsqu'elle va **être exécutée plusieurs fois pendant la même session** (je vous rappelle que les requêtes préparées sont supprimées à la fin de la session).

Le fait que la requête soit compilée lors de sa préparation, et que le plan d'exécution soit calculé également lors de la préparation et non de l'exécution, explique pourquoi les paramètres peuvent uniquement être des valeurs, et non des parties de requêtes comme des noms de tables ou des mots-clés. En effet, il est impossible de créer le plan si l'on ne sait pas encore sur quelles tables on travaillera ni quelles colonnes (et donc quels index) seront utilisées.

## Procédures stockées

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

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

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

### Création et utilisation d'une procédure

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

CREATE PROCEDURE nom\_procedure ([parametre1 [, parametre2, ...]])

corps de la procédure;

Décodons tout ceci.

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

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

#### Procédure avec une seule requête

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

CREATE PROCEDURE afficher\_races\_requete()

-- pas de paramètres dans les parenthèses

SELECT id, nom, espece\_id, prix FROM Race;

#### Procédure avec un bloc d'instructions

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

BEGIN

-- Série d'instructions

END;

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

CREATE PROCEDURE afficher\_races\_bloc()

-- pas de paramètres dans les parenthèses

BEGIN

SELECT id, nom, espece\_id, prix FROM Race;

END;

Malheureusement…

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '' at line 3

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

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

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

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

Il suffit de changer le délimiteur !

#### Délimiteur

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

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

DELIMITER |

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

SELECT 'test'|

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

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

Les deux délimiteurs suivants sont les plus couramment utilisés :

DELIMITER //

DELIMITER |

Bien ! Ceci étant réglé, reprenons !

#### Création d'une procédure stockée

DELIMITER | -- On change le délimiteur

CREATE PROCEDURE afficher\_races()

-- toujours pas de paramètres, toujours des parenthèses

BEGIN

SELECT id, nom, espece\_id, prix

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

END| -- Et on termine bien sûr la commande CREATE PROCEDURE par notre nouveau délimiteur

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

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

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

#### Utilisation d'une procédure stockée

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

CALL afficher\_races()| -- le délimiteur est toujours | !!!

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

### Les paramètres d'une procédure stockée

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

#### Sens des paramètres

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

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

#### Syntaxe

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

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

#### Exemples

##### Procédure avec un seul paramètre entrant

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

DELIMITER | -- Facultatif si votre délimiteur est toujours |

CREATE PROCEDURE afficher\_race\_selon\_espece (IN p\_espece\_id *INT*)

-- Définition du paramètre p\_espece\_id

BEGIN

SELECT id, nom, espece\_id, prix

FROM Race

WHERE espece\_id = p\_espece\_id; -- Utilisation du paramètre

END |

DELIMITER ; -- On remet le délimiteur par défaut

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

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

CALL afficher\_race\_selon\_espece(1);

*SET* @espece\_id := 2;

CALL afficher\_race\_selon\_espece(@espece\_id);

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

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

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

Voici une procédure assez similaire à la précédente, si ce n'est qu'elle n'affiche pas les races existant pour une espèce, mais compte combien il y en a, puis stocke cette valeur dans un paramètre sortant.

DELIMITER |

CREATE PROCEDURE compter\_races\_selon\_espece (p\_espece\_id *INT*, OUT p\_nb\_races *INT*)

BEGIN

SELECT COUNT(\*) INTO p\_nb\_races

FROM Race

WHERE espece\_id = p\_espece\_id;

END |

DELIMITER ;

Aucun sens n'a été précisé pour p\_espece\_id, il est donc considéré comme un paramètre entrant.

SELECT COUNT(\*) INTO p\_nb\_races, voilà qui est nouveau ! Comme vous l'avez sans doute deviné, le mot-clé INTO  placé après la clause SELECT  permet d'**assigner les valeurs sélectionnées** par ce SELECT  à des variables, au lieu de simplement afficher les valeurs sélectionnées.  
Dans le cas présent, la valeur du COUNT(\*)  est assignée à p\_nb\_races.

Pour pouvoir l'utiliser, il est nécessaire que le SELECT  ne renvoie qu'**une seule ligne**, et il faut que le nombre de valeurs sélectionnées et le nombre de variables à assigner **soient égaux :**

**Exemple 1 :** SELECT ... INTO  correct avec deux valeurs

SELECT id, nom INTO @var1, @var2

FROM Animal

WHERE id = 7;

SELECT @var1, @var2;

Le SELECT ... INTO  n'a rien affiché, mais a assigné la valeur 7  à *@var1*, et la valeur'Caroline'  à *@var2.* Nous les avons ensuite affichées avec un autre SELECT.

**Exemple 2 :** SELECT ... INTO  incorrect, car le nombre de valeurs sélectionnées (deux) n'est pas le même que le nombre de variables à assigner (une).

SELECT id, nom INTO @var1

FROM Animal

WHERE id = 7;

ERROR 1222 (21000): The used SELECT statements have a different number of columns

**Exemple 3 :** SELECT ... INTO  incorrect, car il y a plusieurs lignes de résultats.

SELECT id, nom INTO @var1, @var2

FROM Animal

WHERE espece\_id = 5;

ERROR 1172 (42000): Result consisted of more than one row

Revenons maintenant à notre nouvelle procédure *compter\_races\_selon\_espece()* et exécutons-la. Pour cela, il va falloir lui passer deux paramètres : *p\_espece\_id* et *p\_nb\_races*.  
Le premier ne pose pas de problème, il faut simplement donner un nombre, soit directement, soit par l'intermédiaire d'une variable, comme pour la procédure *afficher\_race\_selon\_espece()*.  
Par contre, pour le second, il s'agit d'un paramètre sortant. Il ne faut donc pas donner une valeur, mais quelque chose dont la valeur sera déterminée par la procédure (grâce au SELECT ... INTO) et que l'on pourra utiliser ensuite : **une variable utilisateur !**

CALL compter\_races\_selon\_espece (2, @nb\_races\_chats);

Et voilà ! La variable *@nb\_races\_chats* contient maintenant le nombre de races de chats. Il suffit de l'afficher pour vérifier.

SELECT @nb\_races\_chats;

**Procédure avec deux paramètres, un entrant et un entrant-sortant**

Nous allons créer une procédure qui va servir à calculer le prix que doit payer un client. Pour cela, deux paramètres sont nécessaires : l'animal acheté (paramètre IN), et le prix à payer (paramètre INOUT).  
La raison pour laquelle le prix est un paramètre à la fois entrant et sortant est que l'on veut pouvoir, avec cette procédure, calculer simplement un prix total dans le cas où un client achèterait plusieurs animaux.  
Le principe est simple : si le client n'a encore acheté aucun animal, le prix est de 0. Pour chaque animal acheté, on appelle la procédure, qui ajoute au prix total le prix de l'animal en question.  
Une fois n'est pas coutume, commençons par voir les requêtes qui nous serviront à tester la procédure. Cela devrait clarifier le principe. Je vous propose d'essayer ensuite d'écrire vous-même la procédure correspondante avant de regarder à quoi elle ressemble.

*SET* @prix = 0; -- On initialise @prix à 0

CALL calculer\_prix (13, @prix); -- Achat de Rouquine

SELECT @prix AS prix\_intermediaire;

CALL calculer\_prix (24, @prix); -- Achat de Cartouche

SELECT @prix AS prix\_intermediaire;

CALL calculer\_prix (42, @prix); -- Achat de Bilba

SELECT @prix AS prix\_intermediaire;

CALL calculer\_prix (75, @prix); -- Achat de Mimi

SELECT @prix AS total;

On passe donc chaque animal acheté tour à tour à la procédure, qui modifie le prix en conséquence. Voici quelques indices et rappels qui devraient vous aider à écrire vous-même la procédure.

* Le prix n'est pas un nombre entier.
* Il est possible de faire des additions directement dans un SELECT.
* Pour déterminer le prix, il faut utiliser la fonction COALESCE().

Réponse :

DELIMITER |

CREATE PROCEDURE calculer\_prix (IN p\_animal\_id *INT*, INOUT p\_prix *DECIMAL*(7,2))

BEGIN

SELECT p\_prix + COALESCE(Race.prix, Espece.prix) INTO p\_prix

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

LEFT JOIN Race ON Race.id = Animal.race\_id

WHERE Animal.id = p\_animal\_id;

END |

DELIMITER ;

### Suppression d'une procédure

Vous commencez à connaître cette commande : pour supprimer une procédure, on utilise DROP  (en précisant qu'il s'agit d'une procédure).

**Exemple :**

DROP PROCEDURE afficher\_races;

Pour rappel, les procédures stockées ne sont pas détruites à la fermeture de la session, mais bien enregistrées comme un élément de la base de données, au même titre qu'une table, par exemple.

Notons encore qu'il n'est pas possible de modifier une procédure directement. La seule façon de modifier une procédure existante est de la supprimer puis de la recréer avec les modifications.

Il existe bien une commande ALTER PROCEDURE, mais elle ne permet de changer ni les paramètres ni le corps de la procédure. Elle permet uniquement de changer certaines caractéristiques de la procédure, et ne sera pas couverte dans ce cours.

### Avantages, inconvénients et usage des procédures stockées

#### Avantages

Les procédures stockées permettent de **réduire les allers-retours entre le client et le serveur MySQL**. En effet, si l'on englobe en une seule procédure un processus demandant l'exécution de plusieurs requêtes, le client ne communique qu'une seule fois avec le serveur (pour demander l'exécution de la procédure) pour exécuter la totalité du traitement. Cela permet donc un certain **gain en performance**.

Elles permettent également de **sécuriser**une base de données. Par exemple, il est possible de **restreindre les droits des utilisateurs** de façon à ce qu'ils puissent **uniquement exécuter des procédures**. Finis les DELETE  dangereux ou les UPDATE  inconsidérés. Chaque requête exécutée par les utilisateurs est créée et contrôlée par l'administrateur de la base de données par l'intermédiaire des procédures stockées.

Cela permet ensuite de**s'assurer qu'un traitement est toujours exécuté de la même manière**, quelle que soit l'application/le client qui le lance. Il arrive par exemple qu'une même base de données soit exploitée par plusieurs applications, lesquelles peuvent être écrites avec différents langages. Si on laisse chaque application avoir son propre code pour un même traitement, il est possible que des différences apparaissent (distraction, mauvaise communication, erreur ou autre). Par contre, si chaque application appelle la même procédure stockée, ce risque disparaît.

#### Inconvénients

Les procédures stockées**ajoutent évidemment à la charge du serveur de données**. Plus on implémente de logique de traitement directement dans la base de données, moins le serveur est disponible pour son but premier : le stockage de données.

Par ailleurs, certains traitements seront toujours plus simples et plus courts à écrire (et donc à maintenir) s'ils sont développés dans un langage informatique adapté. A fortiori lorsqu'il s'agit de traitements complexes. **La logique qu'il est possible d'implémenter avec MySQL permet de nombreuses choses, mais reste assez basique**.

Enfin, **la syntaxe des procédures stockées diffère beaucoup d'un SGBD à un autre**. Par conséquent, si l'on désire en changer, il faudra procéder à un grand nombre de corrections et d'ajustements.

#### Conclusion et usage

Comme souvent, tout est question d'**équilibre**. Il faut savoir utiliser des procédures quand c'est utile, quand on a une bonne raison de le faire. Il ne sert à rien d'en abuser.  
Pour une base contenant des données ultrasensibles, une bonne gestion des droits des utilisateurs couplée à l'usage de procédures stockées peut se révéler salutaire.  
Pour une base de données destinée à être utilisée par plusieurs applications différentes, on choisira de créer des procédures pour les traitements généraux et/ou pour lesquels la moindre erreur peut poser de gros problèmes.  
Pour un traitement long, impliquant de nombreuses requêtes et une logique simple, on peut sérieusement gagner en performance en le faisant dans une procédure stockée (a fortiori si ce traitement est souvent lancé).

À vous de voir quelles procédures sont utiles pour **votre application et vos besoins**.

## Structurez vos instructions

Lorsque l'on écrit une série d'instructions, par exemple dans le corps d'une procédure stockée, il est nécessaire d'être capable de structurer ses instructions. Cela va permettre d’instiller de la **logique dans le traitement :** exécuter telles ou telles instructions en fonction des données que l'on possède, répéter une instruction un certain nombre de fois, etc.

Voici quelques outils indispensables à la structuration des instructions :

* les **variables locales :** elles vont permettre de **stocker et de modifier des valeurs** pendant le déroulement d'une procédure ;
* les **conditions :** elles vont permettre d'exécuter certaines instructions seulement **si une certaine condition est remplie ;**
* les **boucles :** elles vont permettre de **répéter une instruction** plusieurs fois.

Ces structures sont bien sûr utilisables dans les procédures stockées, que nous avons vues au chapitre précédent, mais pas uniquement. Elles sont **utilisables dans tout objet définissant une série d'instructions à exécuter**. C'est le cas des **fonctions stockées** (non couvertes par ce cours et qui forment avec les procédures stockées ce que l'on appelle les "routines"), des événements (non couverts), et également des **triggers**, auxquels un chapitre est consacré à la fin de cette partie.

### Blocs d'instructions et variables locales

#### Blocs d'instructions

Nous avons vu qu'un bloc d'instructions était défini par les mots-clés BEGIN  et END, entre lesquels on met les instructions qui composent le bloc (de zéro à autant d'instructions que l'on veut, séparées bien sûr d'un ;).

Il est possible d'imbriquer plusieurs blocs d'instructions. De même, à l'intérieur d'un bloc d'instructions, plusieurs blocs d'instructions peuvent se suivre. Ceux-ci permettent donc de **structurer les instructions** en plusieurs parties distinctes et sur plusieurs niveaux d'imbrication différents.

BEGIN

SELECT 'Bloc d''instructions principal';

BEGIN

SELECT 'Bloc d''instructions 2, imbriqué dans le bloc principal';

BEGIN

SELECT 'Bloc d''instructions 3, imbriqué dans le bloc d''instructions 2';

END;

END;

BEGIN

SELECT 'Bloc d''instructions 4, imbriqué dans le bloc principal';

END;

END;

Cet exemple montre également l'importance de l'**indentation** pour avoir un code lisible. Ici, toutes les instructions d'un bloc sont au même niveau et décalées vers la droite par rapport à la déclaration du bloc. Cela permet de voir en un coup d’œil où commence et où se termine chaque bloc d'instructions.

#### Variables locales

Nous connaissons déjà les variables utilisateur, qui sont des variables désignées par @. J'ai également mentionné l'existence des variables système, qui sont des variables prédéfinies par MySQL.  
Voyons maintenant les **variables locales**, qui peuvent être définies dans un bloc d'instructions.

##### Déclaration d'une variable locale

La déclaration d'une variable locale se fait avec l'instruction DECLARE  :

DECLARE nom\_variable type\_variable [DEFAULT valeur\_defaut];

Cette instruction doit se trouver tout au début du bloc d'instructions dans lequel la variable locale sera utilisée (donc directement après le BEGIN).

On a donc une structure générale des blocs d'instructions qui se dégage :

BEGIN

-- Déclarations (de variables locales par exemple)

-- Instructions (dont éventuels blocs d'instructions imbriqués)

END;

Tout comme pour les variables utilisateur, le nom des variables locales n'est **pas** sensible à la casse.

Si aucune valeur par défaut n'est précisée, la variable vaudra NULL  tant que sa valeur n'est pas changée.  
Pour changer la valeur d'une variable locale, on peut utiliser SET  ou SELECT ... INTO.

**Exemple :** voici une procédure stockée qui donne la date d'aujourd'hui et de demain.

DELIMITER |

CREATE PROCEDURE aujourdhui\_demain ()

BEGIN

DECLARE v\_date *DATE* DEFAULT CURRENT\_DATE();

-- On déclare une variable locale et on lui met une valeur par défaut

SELECT DATE\_FORMAT(v\_date, '%W %e %M %Y') AS Aujourdhui;

*SET* v\_date = v\_date + INTERVAL 1 DAY;

-- On change la valeur de la variable locale

SELECT DATE\_FORMAT(v\_date, '%W %e %M %Y') AS Demain;

END|

DELIMITER ;

Testons-la :

*SET* lc\_time\_names = 'fr\_FR';

CALL aujourdhui\_demain();

Tout comme pour les paramètres, les variables locales peuvent poser problème si l'on ne fait pas attention au nom qu'on leur donne. En cas de conflit (avec un nom de colonne, par exemple), comme pour les paramètres, le nom sera interprété comme désignant la variable locale en priorité. Par conséquent, toutes mes variables locales seront préfixées par "v\_".

##### Portée des variables locales dans un bloc d'instructions

Les variables locales n'existent que dans le bloc d'instructions dans lequel elles ont été déclarées. Dès que le mot-clé END  est atteint, toutes les variables locales du bloc sont détruites.

**Exemple 1 :**

DELIMITER |

CREATE PROCEDURE test\_portee1()

BEGIN

DECLARE v\_test1 *INT* DEFAULT 1;

BEGIN

DECLARE v\_test2 *INT* DEFAULT 2;

SELECT 'Imbriqué' AS Bloc;

SELECT v\_test1, v\_test2;

END;

SELECT 'Principal' AS Bloc;

SELECT v\_test1, v\_test2;

END|

DELIMITER ;

CALL test\_portee1();

**Exemple 2 :**

DELIMITER |

CREATE PROCEDURE test\_portee2()

BEGIN

DECLARE v\_test1 *INT* DEFAULT 1;

BEGIN

DECLARE v\_test2 *INT* DEFAULT 2;

SELECT 'Imbriqué 1' AS Bloc;

SELECT v\_test1, v\_test2;

END;

BEGIN

SELECT 'imbriqué 2' AS Bloc;

SELECT v\_test1, v\_test2;

END;

END|

DELIMITER ;

CALL test\_portee2();

**Exemple 3 :**

DELIMITER |

CREATE PROCEDURE test\_portee3()

BEGIN

DECLARE v\_test *INT* DEFAULT 1;

SELECT v\_test AS 'Bloc principal';

BEGIN

DECLARE v\_test *INT* DEFAULT 0;

SELECT v\_test AS 'Bloc imbriqué';

*SET* v\_test = 2;

SELECT v\_test AS 'Bloc imbriqué après modification';

END;

SELECT v\_test AS 'Bloc principal';

END |

DELIMITER ;

CALL test\_portee3();

La variable locale v\_test est déclarée dans le bloc principal et dans le bloc imbriqué, avec deux valeurs différentes. Mais lorsque l'on revient dans le bloc principal après exécution du bloc d'instructions imbriqué, v\_test a toujours la valeur qu'elle avait avant l'exécution de ce bloc et sa deuxième déclaration. Il s'agit donc bien de **deux variables locales distinctes**.

### Structures conditionnelles

Les structures conditionnelles permettent de déclencher une action ou une série d'instructions lorsqu'une condition préalable est remplie.

MySQL propose deux structures conditionnelles : IF  et CASE.

#### La structure IF

Voici la syntaxe de la structure IF  :

IF condition THEN instructions

[ELSEIF autre\_condition THEN instructions

[ELSEIF ...]]

[ELSE instructions]

END IF;

##### Le cas le plus simple : si la condition est vraie, alors on exécute ces instructions.

Voici la structure minimale d'un IF  :

IF condition THEN

instructions

END IF;

Soit on exécute les instructions (si la condition est vraie), soit on ne les exécute pas.

**Exemple :** la procédure suivante affiche 'J''ai déjà été adopté !', si c'est le cas, à partir de l'id d'un animal.

DELIMITER |

CREATE PROCEDURE est\_adopte(IN p\_animal\_id *INT*)

BEGIN

DECLARE v\_nb *INT* DEFAULT 0;

-- On crée une variable locale

SELECT COUNT(\*) INTO v\_nb

FROM Adoption

WHERE animal\_id = p\_animal\_id;

-- On met le nombre de lignes correspondant à l'animal dans Adoption dans notre variable locale

IF v\_nb > 0 THEN

-- On teste si v\_nb est supérieur à 0 (donc si l'animal a été adopté)

SELECT 'J''ai déjà été adopté !';

END IF;

-- Et on n'oublie surtout pas le END IF et le ; final

END |

DELIMITER ;

CALL est\_adopte(3);

CALL est\_adopte(28);

Seul le premier appel à la procédure va afficher 'J''ai déjà été adopté !', puisque l'animal 3 est présent dans la table Adoption, contrairement à l'animal 28.

##### Deuxième cas : si ... alors, sinon ...

Grâce au mot-clé ELSE, on peut définir une série d'instructions à exécuter si la condition est fausse.

ELSE  ne doit **pas** être suivi de THEN.

**Exemple :** la procédure suivante affiche 'Je suis né avant 2010'  ou 'Je suis né après 2010', selon la date de naissance de l'animal transmis en paramètre.

DELIMITER |

CREATE PROCEDURE avant\_apres\_2010(IN p\_animal\_id *INT*)

BEGIN

DECLARE v\_annee *INT*;

SELECT YEAR(date\_naissance) INTO v\_annee

FROM Animal

WHERE id = p\_animal\_id;

IF v\_annee < 2010 THEN

SELECT 'Je suis né avant 2010' AS naissance;

ELSE -- Pas de THEN

SELECT 'Je suis né après 2010' AS naissance;

END IF; -- Toujours obligatoire

END |

DELIMITER ;

CALL avant\_apres\_2010(34); -- Né le 20/04/2008

CALL avant\_apres\_2010(69); -- Né le 13/02/2012

##### Troisième et dernier cas : plusieurs conditions alternatives

Enfin, le mot-clé ELSEIF... THEN  permet de vérifier d'autres conditions (en dehors de la condition du IF), chacune ayant une série d'instructions définies à exécuter en cas de véracité. Si plusieurs conditions sont vraies en même temps, seule la première rencontrée verra ses instructions exécutées.  
On peut bien sûr toujours (mais ce n'est pas obligatoire) ajouter un ELSE  pour le cas où aucune condition ne serait vérifiée.

**Exemple :** cette procédure affiche un message différent selon le sexe de l'animal passé en paramètre.

DELIMITER |

CREATE PROCEDURE message\_sexe(IN p\_animal\_id *INT*)

BEGIN

DECLARE v\_sexe *VARCHAR*(10);

SELECT sexe INTO v\_sexe

FROM Animal

WHERE id = p\_animal\_id;

IF (v\_sexe = 'F') THEN -- Première possibilité

SELECT 'Je suis une femelle !' AS sexe;

ELSEIF (v\_sexe = 'M') THEN -- Deuxième possibilité

SELECT 'Je suis un mâle !' AS sexe;

ELSE -- Défaut

SELECT 'Je suis en plein questionnement existentiel...' AS sexe;

END IF;

END|

DELIMITER ;

CALL message\_sexe(8); -- Mâle

CALL message\_sexe(6); -- Femelle

CALL message\_sexe(9); -- Ni l'un ni l'autre

Il peut bien sûr y avoir autant de ELSEIF... THEN  que l'on veut (mais un seul ELSE).

#### La structure CASE

Deux syntaxes sont possibles pour utiliser CASE.

##### Première syntaxe : conditions d'égalité

CASE valeur\_a\_comparer

WHEN possibilite1 THEN instructions

[WHEN possibilite2 THEN instructions] ...

[ELSE instructions]

END CASE;

**Exemple :** on reprend la procédure message\_sexe(), et on l'adapte pour utiliser CASE.

DELIMITER |

CREATE PROCEDURE message\_sexe2(IN p\_animal\_id *INT*)

BEGIN

DECLARE v\_sexe *VARCHAR*(10);

SELECT sexe INTO v\_sexe

FROM Animal

WHERE id = p\_animal\_id;

CASE v\_sexe

WHEN 'F' THEN -- Première possibilité

SELECT 'Je suis une femelle !' AS sexe;

WHEN 'M' THEN -- Deuxième possibilité

SELECT 'Je suis un mâle !' AS sexe;

ELSE -- Défaut

SELECT 'Je suis en plein questionnement existentiel...' AS sexe;

END CASE;

END|

DELIMITER ;

CALL message\_sexe2(8); -- Mâle

CALL message\_sexe2(6); -- Femelle

CALL message\_sexe2(9); -- Ni l'un ni l'autre

On définit donc v\_sexe comme point de comparaison. Chaque WHEN  donne alors un élément auquel v\_sexe doit être comparé. Les instructions exécutées seront celles du WHEN  dont l'élément est égal à v\_sexe. Le ELSE  sera exécuté si aucun WHEN  ne correspond.

Ici, on compare une variable locale (v\_sexe) à des chaînes de caractères ('F'  et 'M'), mais on peut utiliser différents types d'éléments. Voici les principaux :

* des variables locales ;
* des variables utilisateur ;
* des valeurs constantes de tous types (0, 'chaîne', 5.67, '2012-03-23'…) ;
* des expressions (2 + 4, NOW(), CONCAT(nom, ' ', prenom)…) ;
* …

Cette syntaxe ne permet pas de faire des comparaisons avec NULL, puisqu'elle utilise une comparaison de type valeur1 = valeur2. Or cette comparaison est inutilisable dans le cas de NULL. Il faudra donc utiliser la seconde syntaxe, avec le test IS NULL.

##### Seconde syntaxe : toutes conditions

Cette seconde syntaxe ne compare pas un élément à différentes valeurs, mais utilise simplement des conditions classiques et permet donc de faire des comparaisons de type "plus grand que", "différent de", etc. (bien entendu, elle peut également être utilisée pour des égalités).

CASE

WHEN condition THEN instructions

[WHEN condition THEN instructions] ...

[ELSE instructions]

END CASE

**Exemple :** on reprend la procédure avant\_apres\_2010(), que l'on réécrit avec CASE, et en donnant une possibilité en plus. De plus, on passe le message en paramètre OUT  pour changer un peu.

DELIMITER |

CREATE PROCEDURE avant\_apres\_2010\_case (IN p\_animal\_id *INT*, OUT p\_message *VARCHAR*(100))

BEGIN

DECLARE v\_annee *INT*;

SELECT YEAR(date\_naissance) INTO v\_annee

FROM Animal

WHERE id = p\_animal\_id;

CASE

WHEN v\_annee < 2010 THEN

*SET* p\_message = 'Je suis né avant 2010.';

WHEN v\_annee = 2010 THEN

*SET* p\_message = 'Je suis né en 2010.';

ELSE

*SET* p\_message = 'Je suis né après 2010.';

END CASE;

END |

DELIMITER ;

CALL avant\_apres\_2010\_case(59, @message);

SELECT @message;

CALL avant\_apres\_2010\_case(62, @message);

SELECT @message;

CALL avant\_apres\_2010\_case(69, @message);

SELECT @message;

##### Comportement particulier : aucune correspondance trouvée

En l'absence de clause ELSE, si aucune des conditions posées par les différentes clauses WHEN  n'est remplie (quelle que soit la syntaxe utilisée), une erreur est déclenchée.

Par exemple, cette procédure affiche une salutation différente selon la terminaison du nom de l'animal passé en paramètre :

DELIMITER |

CREATE PROCEDURE salut\_nom(IN p\_animal\_id *INT*)

BEGIN

DECLARE v\_terminaison *CHAR*(1);

SELECT SUBSTRING(nom, -1, 1) INTO v\_terminaison

-- Une position négative signifie qu'on recule au lieu d'avancer, -1 est donc la dernière lettre du nom..

FROM Animal

WHERE id = p\_animal\_id;

CASE v\_terminaison

WHEN 'a' THEN

SELECT 'Bonjour !' AS Salutations;

WHEN 'o' THEN

SELECT 'Salut !' AS Salutations;

WHEN 'i' THEN

SELECT 'Coucou !' AS Salutations;

END CASE;

END|

DELIMITER ;

CALL salut\_nom(69); -- Baba

CALL salut\_nom(5); -- Choupi

CALL salut\_nom(29); -- Fiero

CALL salut\_nom(54); -- Bubulle

L'appel de la procédure avec Bubulle présente un cas qui n'est pas couvert par les trois WHEN. Une erreur est donc déclenchée.

Donc, si l'on n'est pas sûr d'avoir couvert tous les cas possibles, il faut toujours ajouter une clause ELSE  pour éviter les erreurs.  
Si l'on veut qu'aucune instruction ne soit exécutée par le ELSE, il suffit de mettre un bloc d'instructions vide (BEGIN END;).

**Exemple :** reprenons la procédure salut\_nom(), et ajoutons-lui une clause ELSE  vide :

DROP PROCEDURE salut\_nom;

DELIMITER |

CREATE PROCEDURE salut\_nom(IN p\_animal\_id *INT*)

BEGIN

DECLARE v\_terminaison *CHAR*(1);

SELECT SUBSTRING(nom, -1, 1) INTO v\_terminaison

FROM Animal

WHERE id = p\_animal\_id;

CASE v\_terminaison

WHEN 'a' THEN

SELECT 'Bonjour !' AS Salutations;

WHEN 'o' THEN

SELECT 'Salut !' AS Salutations;

WHEN 'i' THEN

SELECT 'Coucou !' AS Salutations;

ELSE

BEGIN -- Bloc d'instructions vide

END;

END CASE;

END|

DELIMITER ;

CALL salut\_nom(69); -- Baba

CALL salut\_nom(5); -- Choupi

CALL salut\_nom(29); -- Fiero

CALL salut\_nom(54); -- Bubulle

Cette fois, pas d'erreur. Le dernier appel (avec Bubulle) n'affiche simplement rien.

Il faut au minimum une instruction ou un bloc d'instructions par clause WHEN  et par clause ELSE. Un bloc vide BEGIN END;  est donc nécessaire si l'on ne veut rien exécuter.

#### Utiliser une structure conditionnelle directement dans une requête

Jusqu'ici, on a vu l'usage des structures conditionnelles dans des procédures stockées. Il est cependant possible d'utiliser une structure CASE  dans une simple requête.

Par exemple, écrivons une requête SELECT  suivant le même principe que la procédure message\_sexe() :

SELECT id, nom, CASE

WHEN sexe = 'M' THEN 'Je suis un mâle !'

WHEN sexe = 'F' THEN 'Je suis une femelle !'

ELSE 'Je suis en plein questionnement existentiel...'

END AS message

FROM Animal

WHERE id IN (9, 8, 6);

Quelques remarques :

* On peut utiliser les deux syntaxes de CASE.
* Il faut clôturer le CASE  par END, et non par END CASE  (et bien sûr ne pas mettre de ; si la requête n'est pas finie).
* Ce n'est pas limité aux clauses SELECT, on peut tout à fait utiliser un CASE  dans une clause WHERE, par exemple.
* Ce n'est par conséquent pas non plus limité aux requêtes SELECT, on peut l'utiliser dans n'importe quelle requête.

Il n'est par contre pas possible d'utiliser une structure IF  dans une requête. Cependant, il existe une **fonction**IF(), beaucoup plus limitée, dont la syntaxe est la suivante :

IF(condition, valeur\_si\_vrai, valeur\_si\_faux)

**Exemple :**

SELECT nom, IF(sexe = 'M', 'Je suis un mâle', 'Je ne suis pas un mâle') AS sexe

FROM Animal

WHERE espece\_id = 5;

### Boucles

Une boucle est une structure qui permet de répéter plusieurs fois une série d'instructions. Il existe trois types de boucles en MySQL : WHILE, LOOP  et REPEAT.

#### La boucle WHILE

La boucle WHILE  permet de répéter une série d'instructions **tant que la condition donnée reste vraie**.

WHILE condition DO

-- Attention de ne pas oublier le DO, erreur classique

instructions

END WHILE;

**Exemple :** la procédure suivante affiche les nombres entiers de 1 à p\_nombre (passé en paramètre).

DELIMITER |

CREATE PROCEDURE compter\_jusque\_while(IN p\_nombre *INT*)

BEGIN

DECLARE v\_i *INT* DEFAULT 1;

WHILE v\_i <= p\_nombre DO

SELECT v\_i AS nombre;

*SET* v\_i = v\_i + 1;

-- À ne surtout pas oublier, sinon la condition restera vraie

END WHILE;

END |

DELIMITER ;

CALL compter\_jusque\_while(3);

Vérifiez que votre condition devient bien fausse après un certain nombre d'itérations de la boucle. Sinon, vous vous retrouvez avec une boucle infinie (qui ne s'arrête jamais).

#### La boucle REPEAT

La boucle REPEAT  travaille en quelque sorte de manière opposée à WHILE, puisqu'elle exécute des instructions de la boucle **jusqu'à ce que la condition donnée devienne vraie**.

**Exemple :** voici la même procédure écrite avec une boucle REPEAT.

DELIMITER |

CREATE PROCEDURE compter\_jusque\_repeat(IN p\_nombre *INT*)

BEGIN

DECLARE v\_i *INT* DEFAULT 1;

REPEAT

SELECT v\_i AS nombre;

*SET* v\_i = v\_i + 1;

-- À ne surtout pas oublier, sinon la condition restera vraie

UNTIL v\_i > p\_nombre END REPEAT;

END |

DELIMITER ;

CALL compter\_jusque\_repeat(3);

Attention, comme la condition d'une boucle REPEAT  est vérifiée après le bloc d'instructions de la boucle, **on passe au moins une fois dans la boucle**, même si la condition est tout de suite fausse !

**Test**

-- Condition fausse dès le départ, on ne rentre pas dans la boucle

CALL compter\_jusque\_while(0);

-- Condition fausse dès le départ, on rentre quand même une fois dans la boucle

CALL compter\_jusque\_repeat(0);

#### Donner un label à une boucle

Il est possible de donner un label (un nom) à une boucle, ou à un bloc d'instructions défini par BEGIN... END. Il suffit pour cela de faire précéder l'ouverture de la boucle/du bloc par ce label, suivi de :.

La fermeture de la boucle/du bloc peut alors faire référence à ce label (mais ce n'est pas obligatoire).

Un label ne peut pas dépasser 16 caractères.

**Exemples**

-- Boucle WHILE

-- ------------

super\_while: WHILE condition DO

-- La boucle a pour label "super\_while"

instructions

END WHILE super\_while;

-- On ferme en donnant le label de la boucle (facultatif)

-- Boucle REPEAT

-- -------------

repeat\_genial: REPEAT -- La boucle s'appelle "repeat\_genial"

instructions

UNTIL condition END REPEAT;

-- Cette fois, on choisit de ne pas faire référence au label lors de la fermeture

-- Bloc d'instructions

-- -------------------

bloc\_extra: BEGIN -- Le bloc a pour label "bloc\_extra"

instructions

END bloc\_extra;

Mais en quoi cela peut-il être utile ?

D'une part, cela peut permettre de **clarifier le code** lorsqu'il y a beaucoup de boucles et de blocs d'instructions imbriqués. D'autre part, il est nécessaire de donner un label aux boucles et aux blocs d'instructions pour lesquels on veut pouvoir **utiliser les instructions ITERATE  et LEAVE**.

#### Les instructions LEAVE  et ITERATE

##### LEAVE  : quitter la boucle ou le bloc d'instructions

L'instruction LEAVE  peut s'utiliser**dans une boucle ou un bloc d'instructions** et **déclenche la sortie immédiate de la structure dont le label est donné**.

LEAVE label\_structure;

**Exemple :** cette procédure incrémente de 1 et affiche un nombre entier passé en paramètre. Et cela 4 fois maximum. Mais si l'on trouve un multiple de 10, la boucle s'arrête.

DELIMITER |

CREATE PROCEDURE test\_leave1(IN p\_nombre *INT*)

BEGIN

DECLARE v\_i *INT* DEFAULT 4;

SELECT 'Avant la boucle WHILE';

while1: WHILE v\_i > 0 DO

*SET* p\_nombre = p\_nombre + 1; -- On incrémente le nombre de 1

IF p\_nombre%10 = 0 THEN -- Si p\_nombre est divisible par 10,

SELECT 'Stop !' AS 'Multiple de 10';

LEAVE while1; -- On quitte la boucle WHILE.

END IF;

SELECT p\_nombre; -- On affiche p\_nombre

*SET* v\_i = v\_i - 1; -- Attention de ne pas l'oublier

END WHILE while1;

SELECT 'Après la boucle WHILE';

END|

DELIMITER ;

CALL test\_leave1(3); -- La boucle s'exécutera 4 fois

**Exemple :** voici la même procédure. Cette fois-ci, un multiple de 10 provoque l'arrêt de toute la procédure, pas seulement de la boucle WHILE.

DELIMITER |

CREATE PROCEDURE test\_leave2(IN p\_nombre *INT*)

corps\_procedure: BEGIN

-- On donne un label au bloc d'instructions principal

DECLARE v\_i *INT* DEFAULT 4;

SELECT 'Avant la boucle WHILE';

while1: WHILE v\_i > 0 DO

*SET* p\_nombre = p\_nombre + 1; -- On incrémente le nombre de 1

IF p\_nombre%10 = 0 THEN -- Si p\_nombre est divisible par 10,

SELECT 'Stop !' AS 'Multiple de 10';

LEAVE corps\_procedure; -- je quitte la procédure.

END IF;

SELECT p\_nombre; -- On affiche p\_nombre

*SET* v\_i = v\_i - 1; -- Attention de ne pas l'oublier

END WHILE while1;

SELECT 'Après la boucle WHILE';

END|

DELIMITER ;

CALL test\_leave2(8);

'Après la boucle WHILE'  ne s'affiche plus lorsque l'instruction LEAVE  est déclenchée, puisque l'on quitte la procédure stockée avant d'arriver à l'instruction SELECT  qui suit la boucle WHILE.

En revanche, LEAVE  ne permet pas de quitter directement une structure conditionnelle (IF  ou CASE). Il n'est d'ailleurs pas non plus possible de donner un label à ces structures. Cette restriction est cependant aisément contournable en utilisant les blocs d'instructions.

**Exemple :** la procédure suivante affiche les nombres de 4 à 1 en précisant s'ils sont pairs. Sauf pour le nombre 2, pour lequel une instruction LEAVE  empêche l'affichage habituel.

DELIMITER |

CREATE PROCEDURE test\_leave3()

BEGIN

DECLARE v\_i *INT* DEFAULT 4;

WHILE v\_i > 0 DO

IF v\_i%2 = 0 THEN

if\_pair: BEGIN

IF v\_i = 2 THEN -- Si v\_i vaut 2

LEAVE if\_pair;

-- On quitte le bloc "if\_pair", ce qui revient à quitter la structure IF v\_i%2 = 0

END IF;

SELECT CONCAT(v\_i, ' est pair') AS message;

END if\_pair;

ELSE

if\_impair: BEGIN

SELECT CONCAT(v\_i, ' est impair') AS message;

END if\_impair;

END IF;

*SET* v\_i = v\_i - 1;

END WHILE;

END|

DELIMITER ;

CALL test\_leave3();

**Exemple :** la procédure suivante affiche les nombres de 1 à 3, avec un message avant le IF  et après le IF. Sauf pour le nombre 2, qui relance une itération de la boucle dans le IF.

DELIMITER |

CREATE PROCEDURE test\_iterate()

BEGIN

DECLARE v\_i *INT* DEFAULT 0;

boucle\_while: WHILE v\_i < 3 DO

*SET* v\_i = v\_i + 1;

SELECT v\_i, 'Avant IF' AS message;

IF v\_i = 2 THEN

ITERATE boucle\_while;

END IF;

SELECT v\_i, 'Après IF' AS message;

-- Ne sera pas exécuté pour v\_i = 2

END WHILE;

END |

DELIMITER ;

CALL test\_iterate();

#### La boucle LOOP

On a gardé la boucle LOOP  pour la fin, parce qu'elle est un peu particulière. En effet, voici sa syntaxe :

[label:] LOOP

instructions

END LOOP [label]

Vous voyez bien : il n'est question de condition nulle part. En fait, une boucle LOOP  doit intégrer dans ses instructions un élément qui va la faire s'arrêter : typiquement, une instruction LEAVE. Sinon, c'est une boucle infinie.

**Exemple :** à nouveau une procédure qui affiche les nombres entiers de 1 à p\_nombre.

DELIMITER |

CREATE PROCEDURE compter\_jusque\_loop(IN p\_nombre *INT*)

BEGIN

DECLARE v\_i *INT* DEFAULT 1;

boucle\_loop: LOOP

SELECT v\_i AS nombre;

*SET* v\_i = v\_i + 1;

IF v\_i > p\_nombre THEN

LEAVE boucle\_loop;

END IF;

END LOOP;

END |

DELIMITER ;

CALL compter\_jusque\_loop(3);

## Gestionnaires d'erreurs, curseurs et utilisation avancée

Dans ce chapitre, nous verrons tout d'abord deux structures utilisables dans les blocs d'instructions qui vont vous ouvrir d'énormes possibilités :

* les **gestionnaires d'erreurs**, qui permettent de gérer les cas où une erreur se produirait pendant l'exécution d'une série d'instructions ;
* les **curseurs**, qui permettent de parcourir les lignes de résultat d'une requête SELECT.

Ensuite, nous verrons quelques cas d'utilisation avancée des blocs d'instructions, utilisant non seulement les structures décrites dans ce chapitre et le précédent, mais également d'autres notions et objets (transaction, variables utilisateur, etc.).

### Gestion des erreurs

Il arrive régulièrement qu'un traitement soit susceptible de générer une erreur SQL. Prenons la procédure suivante, qui enregistre une adoption :

DELIMITER |

CREATE PROCEDURE ajouter\_adoption(IN p\_client\_id *INT*, IN p\_animal\_id *INT*, IN p\_date *DATE*, IN p\_paye TINYINT)

BEGIN

DECLARE v\_prix *DECIMAL*(7,2);

SELECT COALESCE(Race.prix, Espece.prix) INTO v\_prix

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

LEFT JOIN Race ON Race.id = Animal.race\_id

WHERE Animal.id = p\_animal\_id;

INSERT INTO Adoption (animal\_id, client\_id, date\_reservation, date\_adoption, prix, paye)

VALUES (p\_animal\_id, p\_client\_id, CURRENT\_DATE(), p\_date, v\_prix, p\_paye);

SELECT 'Adoption correctement ajoutée' AS message;

END|

DELIMITER ;

Plusieurs erreurs sont susceptibles de se déclencher selon les paramètres passés à cette procédure.

**Exemple 1 :** le client n'existe pas.

*SET* @date\_adoption = CURRENT\_DATE() + INTERVAL 7 DAY;

CALL ajouter\_adoption(18, 6, @date\_adoption, 1);

ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails (`elevage`.`Adoption`, CONSTRAINT `Adoption\_ibfk\_1` FOREIGN KEY (`client\_id`) REFERENCES `Client` (`id`))

**Exemple 2 :** l'animal a déjà été adopté.

CALL ajouter\_adoption(12, 21, @date\_adoption, 1);

ERROR 1062 (23000): Duplicate entry '21' for key 'ind\_uni\_animal\_id'

**Exemple 3 :** l'animal n'existe pas, v\_prix est donc NULL.

CALL ajouter\_adoption(12, 102, @date\_adoption, 1);

ERROR 1048 (23000): Column 'prix' cannot be null

Pour empêcher ces erreurs intempestives, deux solutions :

* vérifier chaque paramètre pouvant poser problème (p\_animal\_id et p\_client\_id ne sont pas NULL  et correspondent à quelque chose dans les tables Animal et Client, p\_animal\_id ne correspond pas à un animal déjà adopté, etc.) ;
* utiliser un gestionnaire d'erreurs : c'est ce que nous allons apprendre à faire ici.

#### Création d'un gestionnaire d'erreurs

Voici la syntaxe à utiliser pour créer un gestionnaire d'erreurs :

DECLARE { EXIT | CONTINUE } HANDLER FOR { numero\_erreur | { SQLSTATE identifiant\_erreur } | condition }

instruction ou bloc d'instructions

* Un gestionnaire d'erreurs définit une instruction (une seule !), ou un bloc d'instructions (BEGIN ... END;), qui va être**exécuté en cas d'erreur** correspondant au gestionnaire.
* Tous les gestionnaires d'erreurs doivent être déclarés au même endroit : **après la déclaration des variables locales, mais avant les instructions de la procédure**.
* Un gestionnaire peut soit provoquer l'**arrêt de la procédure** (EXIT), soit **faire reprendre la procédure** après avoir géré l'erreur (CONTINUE).
* On peut identifier le type d'erreur que le gestionnaire va reconnaître de trois manières différentes : **un numéro d'erreur, un identifiant, ou une CONDITION**.
* Un gestionnaire étant défini grâce au mot-clé DECLARE, comme les variables locales, il a exactement **la même portée** que celles-ci.

**Exemples :** ces deux procédures enregistrent une adoption en gérant les erreurs, l'une arrêtant la procédure, l'autre relançant celle-ci.

DELIMITER |

CREATE PROCEDURE ajouter\_adoption\_exit(IN p\_client\_id *INT*, IN p\_animal\_id *INT*, IN p\_date *DATE*, IN p\_paye TINYINT)

BEGIN

DECLARE v\_prix *DECIMAL*(7,2);

DECLARE EXIT HANDLER FOR SQLSTATE '23000'

BEGIN

SELECT 'Une erreur est survenue...';

SELECT 'Arrêt prématuré de la procédure';

END;

SELECT 'Début procédure';

SELECT COALESCE(Race.prix, Espece.prix) INTO v\_prix

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

LEFT JOIN Race ON Race.id = Animal.race\_id

WHERE Animal.id = p\_animal\_id;

INSERT INTO Adoption (animal\_id, client\_id, date\_reservation, date\_adoption, prix, paye)

VALUES (p\_animal\_id, p\_client\_id, CURRENT\_DATE(), p\_date, v\_prix, p\_paye);

SELECT 'Fin procédure' AS message;

END|

CREATE PROCEDURE ajouter\_adoption\_continue(IN p\_client\_id *INT*, IN p\_animal\_id *INT*, IN p\_date *DATE*, IN p\_paye TINYINT)

BEGIN

DECLARE v\_prix *DECIMAL*(7,2);

DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SELECT 'Une erreur est survenue...';

SELECT 'Début procédure';

SELECT COALESCE(Race.prix, Espece.prix) INTO v\_prix

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

LEFT JOIN Race ON Race.id = Animal.race\_id

WHERE Animal.id = p\_animal\_id;

INSERT INTO Adoption (animal\_id, client\_id, date\_reservation, date\_adoption, prix, paye)

VALUES (p\_animal\_id, p\_client\_id, CURRENT\_DATE(), p\_date, v\_prix, p\_paye);

SELECT 'Fin procédure';

END|

DELIMITER ;

*SET* @date\_adoption = CURRENT\_DATE() + INTERVAL 7 DAY;

CALL ajouter\_adoption\_exit(18, 6, @date\_adoption, 1);

CALL ajouter\_adoption\_continue(18, 6, @date\_adoption, 1);

Les instructions définies par le gestionnaire sont bien exécutées, mais 'Fin procédure'  n'est affiché que dans le cas de ajouter\_adoption\_continue(), qui fait reprendre la procédure une fois l'erreur gérée. La procédure ajouter\_adoption\_exit() utilise un bloc d'instructions et peut donc exécuter plusieurs instructions.

#### Définition de l'erreur gérée

##### Identifiant ou numéro MySQL de l'erreur

Voici la déclaration du gestionnaire dans la procédure ajouter\_adoption\_continue() :

DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SELECT 'Une erreur est survenue...';

Et voici une des erreurs qui peut être interceptée par ce gestionnaire :

ERROR 1062 (23000): Duplicate entry '21' for key 'ind\_uni\_animal\_id'

Le message d'erreur est constitué de trois éléments importants :

* 1062 : le **numéro d'erreur MySQL** (un nombre entier) ;
* 23000 : l'**identifiant de l'état SQL** (une chaîne de 5 caractères) ;
* Duplicate entry '21' for key 'ind\_uni\_animal\_id' : un message donnant le détail de l'erreur.

**Identifiant de l'état SQL**

Dans la procédure ajouter\_adoption\_continue(), c'est l'identifiant de l'état SQL ('23000') qui a été utilisé.  
Il s'agit d'une chaîne de 5 caractères, renvoyée par le serveur au client pour **informer de la réussite ou de l'échec d'une instruction**. Un identifiant commençant par '00', par exemple, signifie que l'instruction a réussi. '23000'  est l'identifiant renvoyé lorsqu'une erreur concernant une contrainte (NOT NULL, unicité, clé primaire ou secondaire…) a été déclenchée.  
Pour utiliser cet identifiant dans un gestionnaire d'erreurs, il faut le faire précéder de SQLSTATE.

**Numéro d'erreur MySQL**

Pour utiliser le numéro d'erreur SQL, par contre, il suffit de l'indiquer, comme un nombre entier :

DECLARE CONTINUE HANDLER FOR 1062 SELECT 'Une erreur est survenue...';

Ce sont des codes qui, contrairement aux identifiants SQLSTATE, sont **propres à MySQL**. Ils sont aussi en général **plus précis**. L'identifiant SQL '23000', par exemple, correspond à une dizaine de codes d'erreur MySQL différents.

**Quelques exemples de codes souvent rencontrés**

| **Code MySQL** | **SQLSTATE** | **Description** |
| --- | --- | --- |
| 1048 | 23000 | La colonne x ne peut pas être NULL |
| 1169 | 23000 | Violation de contrainte d'unicité |
| 1216 | 23000 | Violation de clé secondaire : insertion ou modification impossible (table avec la clé secondaire) |
| 1217 | 23000 | Violation de clé secondaire : suppression ou modification impossible (table avec la **référence**de la clé secondaire) |
| 1172 | 42000 | Plusieurs lignes de résultats alors que l'on ne peut en avoir qu'une seule. |
| 1242 | 21000 | La sous-requête retourne plusieurs lignes de résultats alors que l'on ne peut en avoir qu'une seule. |

Pour une liste plus complète, il suffit d'aller sur la [documentation officielle](https://dev.mysql.com/doc/refman/5.5/en/error-messages-server.html).

Pour finir, notez qu'il est tout à fait possible d'intercepter des avertissements avec un gestionnaire d'erreurs. Ils sont également représentés par un identifiant SQL et un code d'erreur MySQL. Un avertissement, contrairement à une erreur, ne fait pas échouer l'instruction par défaut, mais en l'interceptant dans une requête stockée avec un gestionnaire, vous pouvez décider du comportement à adopter à la suite de cet avertissement.

Vous aurez, par exemple, un avertissement si vous insérez un DATETIME  dans une colonne DATE, puisque la donnée sera tronquée pour correspondre au type de la colonne (l'heure sera supprimée pour ne garder que la date) : code Mysql 1265, SQLSTATE '01000'.

##### Utilisation d'une CONDITION

Avec un numéro d'erreur MySQL et un identifiant d'état SQL, il existe une troisième manière d'identifier les erreurs reconnues par un gestionnaire : une CONDITION.  
Une CONDITION  est simplement un nom donné à un numéro d'erreur MySQL ou à un identifiant d'état SQL. Cela vous permet de travailler avec des erreurs plus claires.

Voici la syntaxe à utiliser pour nommer une erreur. Il s'agit à nouveau d'un DECLARE. Les déclarations de CONDITION  doivent se trouver avant les déclarations de gestionnaires.

DECLARE nom\_erreur CONDITION FOR { SQLSTATE identifiant\_SQL | numero\_erreur\_MySQL };

**Exemple :** réécrivons la procédure ajouter\_adoption\_exit() en nommant l'erreur.

DROP PROCEDURE ajouter\_adoption\_exit;

DELIMITER |

CREATE PROCEDURE ajouter\_adoption\_exit(IN p\_client\_id *INT*, IN p\_animal\_id *INT*, IN p\_date *DATE*, IN p\_paye TINYINT)

BEGIN

DECLARE v\_prix *DECIMAL*(7,2);

DECLARE violation\_contrainte

CONDITION FOR SQLSTATE '23000';

-- On nomme l'erreur dont l'identifiant est 23000 "violation\_contrainte"

DECLARE EXIT HANDLER FOR violation\_contrainte

-- Le gestionnaire sert donc à intercepter les erreurs de type "violation\_contrainte"

BEGIN

SELECT 'Une erreur est survenue...';

SELECT 'Arrêt prématuré de la procédure';

END;

SELECT 'Début procédure';

SELECT COALESCE(Race.prix, Espece.prix) INTO v\_prix

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

LEFT JOIN Race ON Race.id = Animal.race\_id

WHERE Animal.id = p\_animal\_id;

INSERT INTO Adoption (animal\_id, client\_id, date\_reservation, date\_adoption, prix, paye)

VALUES (p\_animal\_id, p\_client\_id, CURRENT\_DATE(), p\_date, v\_prix, p\_paye);

SELECT 'Fin procédure';

END|

DELIMITER ;

##### Conditions prédéfinies

Il existe trois conditions prédéfinies dans MySQL :

* SQLWARNING  : tous les identifiants SQL commençant par '01', c'est-à-dire les avertissements et les notes ;
* NOT FOUND  : tous les identifiants SQL commençant par '02', et que nous verrons plus en détail avec les curseurs ;
* SQLEXCEPTION  : tous les identifiants SQL ne commençant ni par '00', ni par '01', ni par '02', donc les erreurs.

**Exemple :** réécriture de la procédure ajouter\_adoption\_exit(), de façon à ce que le gestionnaire intercepte toutes les erreurs SQL.

DROP PROCEDURE ajouter\_adoption\_exit;

DELIMITER |

CREATE PROCEDURE ajouter\_adoption\_exit(IN p\_client\_id *INT*, IN p\_animal\_id *INT*, IN p\_date *DATE*, IN p\_paye TINYINT)

BEGIN

DECLARE v\_prix *DECIMAL*(7,2);

DECLARE EXIT HANDLER FOR SQLEXCEPTION

BEGIN

SELECT 'Une erreur est survenue...';

SELECT 'Arrêt prématuré de la procédure';

END;

SELECT 'Début procédure';

SELECT COALESCE(Race.prix, Espece.prix) INTO v\_prix

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

LEFT JOIN Race ON Race.id = Animal.race\_id

WHERE Animal.id = p\_animal\_id;

INSERT INTO Adoption (animal\_id, client\_id, date\_reservation, date\_adoption, prix, paye)

VALUES (p\_animal\_id, p\_client\_id, CURRENT\_DATE(), p\_date, v\_prix, p\_paye);

SELECT 'Fin procédure';

END|

DELIMITER ;

#### Déclarer plusieurs gestionnaires, gérer plusieurs erreurs par gestionnaire

Un gestionnaire peut reconnaître plusieurs types d'erreurs différents. Par ailleurs, il est possible de déclarer plusieurs gestionnaires dans un même bloc d'instructions.

**Exemple :** toujours avec la procédure ajouter\_adoption\_exit(). On peut l'écrire en détaillant différentes erreurs possibles, puis en ajoutant un gestionnaire général qui reconnaîtra les SQLEXCEPTION  et les SQLWARNING, pour tous les cas que l'on ne traite pas dans les autres gestionnaires. Ce qui donne :

DROP PROCEDURE ajouter\_adoption\_exit;

DELIMITER |

CREATE PROCEDURE ajouter\_adoption\_exit(IN p\_client\_id *INT*, IN p\_animal\_id *INT*, IN p\_date *DATE*, IN p\_paye TINYINT)

BEGIN

DECLARE v\_prix *DECIMAL*(7,2);

-- Déclaration des CONDITIONS

DECLARE violation\_cle\_etrangere CONDITION FOR 1452;

DECLARE violation\_unicite CONDITION FOR 1062;

-- Déclaration du gestionnaire pour les erreurs de clés étrangères

DECLARE EXIT HANDLER FOR violation\_cle\_etrangere

BEGIN

SELECT 'Erreur : violation de clé étrangère.';

END;

-- Déclaration du gestionnaire pour les erreurs d'index unique

DECLARE EXIT HANDLER FOR violation\_unicite

BEGIN

SELECT 'Erreur : violation de contrainte d''unicité.';

END;

-- Déclaration du gestionnaire pour toutes les autres erreurs ou avertissements

DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING

BEGIN

SELECT 'Une erreur est survenue...';

END;

SELECT 'Début procédure';

SELECT COALESCE(Race.prix, Espece.prix) INTO v\_prix

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

LEFT JOIN Race ON Race.id = Animal.race\_id

WHERE Animal.id = p\_animal\_id;

INSERT INTO Adoption (animal\_id, client\_id, date\_reservation, date\_adoption, prix, paye)

VALUES (p\_animal\_id, p\_client\_id, CURRENT\_DATE(), p\_date, v\_prix, p\_paye);

SELECT 'Fin procédure';

END|

DELIMITER ;

*SET* @date\_adoption = CURRENT\_DATE() + INTERVAL 7 DAY;

CALL ajouter\_adoption\_exit(12, 3, @date\_adoption, 1);

-- Violation unicité (animal 3 est déjà adopté)

CALL ajouter\_adoption\_exit(133, 6, @date\_adoption, 1);

-- Violation clé étrangère (client 133 n'existe pas)

CALL ajouter\_adoption\_exit(NULL, 6, @date\_adoption, 1);

-- Violation de contrainte NOT NULL

Cette procédure montre également que, lorsque plusieurs gestionnaires d'erreurs peuvent correspondre à l'erreur déclenchée (ou à l'avertissement), c'est le plus précis qui est utilisé. C'est la raison pour laquelle une violation de clé étrangère déclenche le gestionnaire FOR violation\_cle\_etrangere  (numéro MySQL 1062), et non le gestionnaire FOR SQLEXCEPTION.

### Curseurs

Nous avons vu qu'il était possible d'exploiter le résultat d'un SELECT  dans un bloc d'instructions en utilisant la commande SELECT colonne(s) INTO variable(s), qui assigne les valeurs sélectionnées à des variables.  
Cependant, SELECT ... INTO  ne peut être utilisé que pour des requêtes qui ne ramènent qu'une seule ligne de résultats.

Les curseurs permettent de **parcourir un jeu de résultats d'une requête SELECT**, quel que soit le nombre de lignes récupérées, et d'en exploiter les valeurs.

Quatre étapes sont nécessaires pour utiliser un curseur.

* Déclaration du curseur : avec une instruction DECLARE.
* Ouverture du curseur : on exécute la requête SELECT  du curseur et on stocke le résultat dans celui-ci.
* Parcours du curseur : on parcourt une à une les lignes.
* Fermeture du curseur.

#### Syntaxe

##### Déclaration du curseur

Comme toutes les instructions DECLARE, la déclaration d'un curseur doit se faire au début du bloc d'instructions pour lequel celui-ci est défini. Plus précisément, on déclare les curseurs **après les variables locales et les conditions**, mais **avant les gestionnaires d'erreurs**.

DECLARE nom\_curseur CURSOR FOR requete\_select;

Un curseur est donc composé d'**un nom**, et d'**une requête SELECT**.

**Exemple :**

DECLARE curseur\_client CURSOR

FOR SELECT \*

FROM Client;

##### Ouverture et fermeture du curseur

En déclarant le curseur, on a donc associé un nom et une requête SELECT. L'ouverture du curseur va provoquer l'exécution de la requête SELECT, ce qui va produire un jeu de résultats.

Une fois que l'on aura parcouru les résultats, il n'y aura plus qu'à fermer le curseur. Si l'on ne le fait pas explicitement, le curseur sera fermé à la fin du bloc d'instructions.

OPEN nom\_curseur;

-- Parcours du curseur et instructions diverses

CLOSE nom\_curseur;

##### Parcours du curseur

Une fois que le curseur a été ouvert et le jeu de résultats récupéré, le curseur place un pointeur sur la première ligne de résultats. Avec la commande FETCH, on récupère la ligne sur laquelle pointe le curseur, et on fait avancer le pointeur vers la ligne de résultats suivante.

FETCH nom\_curseur INTO variable(s);

Bien entendu, comme pour SELECT ... INTO, il faut donner autant de variables dans la clause INTO  que l'on a récupéré de colonnes dans la clause SELECT  du curseur.

**Exemple :** la procédure suivante parcourt les deux premières lignes de la table Client avec un curseur.

DELIMITER |

CREATE PROCEDURE parcours\_deux\_clients()

BEGIN

DECLARE v\_nom, v\_prenom *VARCHAR*(100);

DECLARE curs\_clients CURSOR

FOR SELECT nom, prenom -- Le SELECT récupère deux colonnes

FROM Client

ORDER BY nom, prenom;

-- On trie les clients par ordre alphabétique

OPEN curs\_clients; -- Ouverture du curseur

FETCH curs\_clients INTO v\_nom, v\_prenom;

-- On récupère la première ligne et on assigne les valeurs récupérées à nos variables locales

SELECT CONCAT(v\_prenom, ' ', v\_nom) AS 'Premier client';

FETCH curs\_clients INTO v\_nom, v\_prenom;

-- On récupère la seconde ligne et on assigne les valeurs récupérées à nos variables locales

SELECT CONCAT(v\_prenom, ' ', v\_nom) AS 'Second client';

CLOSE curs\_clients; -- Fermeture du curseur

END|

DELIMITER ;

CALL parcours\_deux\_clients();

#### Restrictions

FETCH  est la seule commande permettant de récupérer une partie d'un jeu de résultats d'un curseur, et elle ne permet qu'une chose : récupérer la ligne de résultats suivante.  
Il n'est pas possible de sauter une ou plusieurs lignes ni d'aller rechercher une ligne précédente. **On ne peut que parcourir les lignes une à une**, de la première à la dernière.

Ensuite, il n'est pas possible de modifier une ligne directement à partir d'un curseur. Il s'agit d'une restriction particulière à MySQL. D'autres SGBD vous permettent des requêtes d'UPDATE  directement sur les curseurs.

Avec Oracle, la requête suivante modifiera la dernière ligne récupérée avec FETCH  :

UPDATE nom\_table

*SET* colonne = valeur

WHERE CURRENT OF nom\_curseur;

Ce n'est, du moins actuellement, absolument **pas possible** avec MySQL !!!

Il vaut d'ailleurs mieux éviter tout UPDATE  sur une table sur laquelle un curseur est ouvert, même sans faire de référence à celui-ci. Le résultat d'un tel UPDATE  serait imprévisible.

Ces restrictions sur les requêtes UPDATE  sont bien entendu également valables pour les suppressions (DELETE).

#### Parcourir intelligemment tous les résultats d'un curseur

Pour récupérer une ligne de résultats, on utilise donc FETCH. Dans la procédure parcours\_deux\_clients(), on voulait récupérer les deux premières lignes, on a donc utilisé deux FETCH.  
Cependant, la plupart du temps, on ne veut pas seulement utiliser les deux premières lignes, mais toutes ! Or, sauf exception, on ne sait pas combien de lignes seront sélectionnées.

On veut donc parcourir une à une les lignes de résultats et leur appliquer un traitement, sans savoir à l'avance combien de fois ce traitement devra être répété.  
Pour cela, on utilise une boucle ! WHILE, REPEAT  ou LOOP. Il n'y a plus qu'à trouver une condition pour arrêter la boucle une fois tous les résultats parcourus.

##### Condition d'arrêt

Voyons ce qui se passe lorsque l'on fait un FETCH  alors qu'il n'y a plus, ou pas, de résultats.  
Voici une procédure qui sélectionne les clients selon une ville donnée en paramètre. Les lignes sont récupérées et affichées grâce au FETCH, placé dans une boucle LOOP. Je rappelle que cette boucle ne définit pas de condition d'arrêt : il est nécessaire d'ajouter une instruction LEAVE  pour l'arrêter. Ici, pour tester, on ne mettra pas d'instruction LEAVE.

DELIMITER |

CREATE PROCEDURE test\_condition(IN p\_ville *VARCHAR*(100))

BEGIN

DECLARE v\_nom, v\_prenom *VARCHAR*(100);

DECLARE curs\_clients CURSOR

FOR SELECT nom, prenom

FROM Client

WHERE ville = p\_ville;

OPEN curs\_clients;

LOOP

FETCH curs\_clients INTO v\_nom, v\_prenom;

SELECT CONCAT(v\_prenom, ' ', v\_nom) AS 'Client';

END LOOP;

CLOSE curs\_clients;

END|

DELIMITER ;

Voyons donc ce que cela donne pour une ville dans laquelle quelques clients habitent.

CALL test\_condition('Houtsiplou');

ERROR 1329 (02000): No data - zero rows fetched, selected, or processed

Tentons ensuite l'expérience avec une ville qui ne donnera aucun résultat.

CALL test\_condition('Bruxelles');

ERROR 1329 (02000): No data - zero rows fetched, selected, or processed

On a la même erreur dans les deux cas, lorsque l'on essaye de faire un FETCH  alors qu'il n'y a pas ou plus de ligne à récupérer. Or, vous vous souvenez peut-être d'une condition prédéfinie pour les gestionnaires d'erreurs, NOT FOUND, qui représente les erreurs dont l'identifiant SQL commence par '02', ce qui est le cas ici.

On va donc utiliser cette condition pour arrêter la boucle : on définit un gestionnaire pour la condition NOT FOUND, qui change la valeur d'une variable locale. Cette variable locale vaut 0 au départ, et passe à 1 quand le gestionnaire est déclenché (donc quand il n'y a plus de ligne). Il suffit alors d'ajouter une structure IF  qui vérifie la valeur de la variable locale une fois le FETCH  exécuté. Si elle vaut 1, on quitte la boucle.

**Exemple :** la procédure test\_condition2() ci-dessous fait la même chose que test\_condition(), mais inclut le gestionnaire d'erreurs et le IF  nécessaires pour stopper la boucle dès que toutes les lignes sélectionnées ont été parcourues :

DELIMITER |

CREATE PROCEDURE test\_condition2(IN p\_ville *VARCHAR*(100))

BEGIN

DECLARE v\_nom, v\_prenom *VARCHAR*(100);

DECLARE fin TINYINT DEFAULT 0;

-- Variable locale utilisée pour stopper la boucle

DECLARE curs\_clients CURSOR

FOR SELECT nom, prenom

FROM Client

WHERE ville = p\_ville;

-- Gestionnaire d'erreur pour la condition NOT FOUND

DECLARE CONTINUE HANDLER FOR NOT FOUND *SET* fin = 1;

OPEN curs\_clients;

loop\_curseur: LOOP

FETCH curs\_clients INTO v\_nom, v\_prenom;

-- Structure IF pour quitter la boucle à la fin des résultats

IF fin = 1 THEN

LEAVE loop\_curseur;

END IF;

SELECT CONCAT(v\_prenom, ' ', v\_nom) AS 'Client';

END LOOP;

CLOSE curs\_clients;

END|

DELIMITER ;

CALL test\_condition2('Houtsiplou');

CALL test\_condition2('Bruxelles');

Je vous laisse le soin, si vous le désirez, de réécrire cette procédure en utilisant une boucle WHILE  ou une boucle REPEAT. C'est un excellent exercice ! :)

Notez que ce n'est pas la seule solution pour arrêter la boucle. On pourrait par exemple compter le nombre de lignes récupérées, grâce à la fonction FOUND\_ROWS()  (dont on a déjà parlé dans le [chapitre sur les fonctions](http://www.siteduzero.com/tutoriel-3-509482-1-rappels-et-introduction.html#ss_part_3)). Cependant, cette utilisation du gestionnaire d'erreurs est la solution la plus utilisée, et je la trouve personnellement très élégante.

##### Le cas des booléens chez MySQL

Un booléen, en informatique, est un type de donnée pouvant prendre deux états : vrai ou faux.  
MySQL ne propose pas ce type de données. Avec MySQL, on représente la valeur "vrai" par 1, et la valeur "faux" par 0.

**Démonstration :**

SELECT 1 = 1, 1 = 2;

-- 1 = 1 est vrai, bien sûr. Contrairement à 1 = 2

Par conséquent, pour représenter un booléen, on utilise en général un TINYINT, valant soit 0, soit 1. C'est ce que l'on a fait pour la colonne paye de la table Client, par exemple.

Une autre conséquence est que la structure IF, que l'on utilise dans la procédure test\_condition2(), peut être réécrite de la manière suivante :

IF fin THEN

LEAVE loop\_curseur;

END IF;

Et pour améliorer encore la lisibilité pour les données de ce genre, MySQL a créé plusieurs synonymes que l'on peut utiliser dans ces situations.

* BOOL  et BOOLEAN  sont synonymes de TINYINT(1).
* TRUE  est synonyme de 1.
* FALSE  est synonyme de 0.

On peut donc réécrire la procédure test\_condition2() en utilisant ces synonymes.

DROP PROCEDURE test\_condition2;

DELIMITER |

CREATE PROCEDURE test\_condition2(IN p\_ville *VARCHAR*(100))

BEGIN

DECLARE v\_nom, v\_prenom *VARCHAR*(100);

-- On déclare fin comme un BOOLEAN, avec FALSE pour défaut

DECLARE fin BOOLEAN DEFAULT FALSE;

DECLARE curs\_clients CURSOR

FOR SELECT nom, prenom

FROM Client

WHERE ville = p\_ville;

-- On utilise TRUE au lieu de 1

DECLARE CONTINUE HANDLER FOR NOT FOUND *SET* fin = TRUE;

OPEN curs\_clients;

loop\_curseur: LOOP

FETCH curs\_clients INTO v\_nom, v\_prenom;

IF fin THEN -- Plus besoin de "= 1"

LEAVE loop\_curseur;

END IF;

SELECT CONCAT(v\_prenom, ' ', v\_nom) AS 'Client';

END LOOP;

CLOSE curs\_clients;

END|

DELIMITER ;

### Utilisation avancée des blocs d'instructions

Vous avez maintenant vu les principales structures qu'il est possible d'utiliser dans un bloc d'instructions. Nous allons ici combiner ces structures avec des objets ou notions vues précédemment.

La puissance du langage SQL (et de tous les langages informatiques) réside dans le fait que l'on peut **combiner différentes notions pour réaliser des traitements complexes**. Voici quelques exemples de telles combinaisons.  
Notez que ces exemples utilisent tous des procédures stockées, mais la plupart sont adaptables à d'autres objets, comme les triggers, les fonctions stockées ou les événements.

#### Utiliser des variables utilisateur dans un bloc d'instructions

En plus des variables locales, il est tout à fait possible d'utiliser des variables utilisateur dans un bloc d'instructions. Mais n'oubliez pas qu'**une variable utilisateur est définie pour toute la session, pas uniquement le bloc**, même si elle est créée à l'intérieur de celui-ci.

**Exemple :** procédure stockée utilisant une variable utilisateur.

DELIMITER |

CREATE PROCEDURE test\_vu()

BEGIN

*SET* @var = 15;

END|

DELIMITER ;

SELECT @var; -- @var n'existe pas encore, on ne l'a pas définie

CALL test\_vu(); -- On exécute la procédure

SELECT @var;

-- @var vaut maintenant 15, même en dehors de la procédure, puisqu'elle est définie partout dans la session

Voyant cela, on pourrait être tenté d'utiliser des variables utilisateur à la place des paramètres OUT  et INOUT  des procédures stockées.

Cependant, il convient d'être extrêmement prudent lorsque l'on utilise des variables utilisateur dans un bloc d'instructions.  
Si l'on reste par exemple dans le contexte des procédures stockées, un des intérêts de celles-ci est d'avoir une interface entre la base de données et l'utilisateur. L'utilisateur n'est donc pas nécessairement conscient des variables utilisateur qui sont définies ou modifiées dans la procédure. Il pourrait donc définir des variables utilisateur, puis exécuter une procédure et constater que certaines de ses variables ont été écrasées par la procédure.

Ainsi dans le code suivant :

*SET* @var = 'Bonjour';

CALL test\_vu();

SELECT @var; -- Donne 15 !

Un utilisateur inconscient du fait que test\_vu() modifie @var  pourrait bien y perdre des cheveux !

Un raisonnement similaire peut être tenu pour les autres objets utilisant les blocs d'instructions.  
Évitez donc les variables utilisateur dans les blocs quand c'est possible (et nous allons bientôt voir un cas où ce n'est pas possible).

#### Utiliser une procédure dans un bloc

CALL nom\_procedure();  est une instruction. On peut donc parfaitement exécuter une procédure dans un bloc d'instructions.

**Exemple :** la procédure surface\_cercle() calcule la surface d'un cercle à partir de son rayon. Elle exécute pour cela la procédure carre(), qui élève un nombre au carré.  
Pour rappel, on calcule la surface d'un cercle avec la formule suivante :  S = π×r2π×r2

DELIMITER |

CREATE PROCEDURE carre(INOUT p\_nb *FLOAT*) *SET* p\_nb = p\_nb \* p\_nb|

CREATE PROCEDURE surface\_cercle(IN p\_rayon *FLOAT*, OUT p\_surface *FLOAT*)

BEGIN

CALL carre(p\_rayon);

*SET* p\_surface = p\_rayon \* PI();

END|

DELIMITER ;

CALL surface\_cercle(1, @surface); -- Donne environ pi (3,14...)

SELECT @surface;

CALL surface\_cercle(2, @surface); -- Donne environ 12,57...

SELECT @surface;

Un FLOAT  stockant **une valeur approchée**, il est tout à fait possible (voire probable) que vous obteniez un résultat différent de celui donné par une calculette.

#### Transactions et gestion d'erreurs

Un usage classique et utile des gestionnaires d'erreurs est l'annulation des transactions en cas d'erreur.

**Exemple :** la procédure suivante prend en paramètre l'id d'un client et de deux animaux que le client veut adopter :

DELIMITER |

CREATE PROCEDURE adoption\_deux\_ou\_rien(p\_client\_id *INT*, p\_animal\_id\_1 *INT*, p\_animal\_id\_2 *INT*)

BEGIN

DECLARE v\_prix *DECIMAL*(7,2);

-- Gestionnaire qui annule la transaction et termine la procédure

DECLARE EXIT HANDLER FOR SQLEXCEPTION ROLLBACK;

START TRANSACTION;

SELECT COALESCE(Race.prix, Espece.prix) INTO v\_prix

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

LEFT JOIN Race ON Race.id = Animal.race\_id

WHERE Animal.id = p\_animal\_id\_1;

INSERT INTO Adoption (animal\_id, client\_id, date\_reservation, date\_adoption, prix, paye)

VALUES (p\_animal\_id\_1, p\_client\_id, CURRENT\_DATE(), CURRENT\_DATE(), v\_prix, TRUE);

SELECT 'Adoption animal 1 réussie' AS message;

SELECT COALESCE(Race.prix, Espece.prix) INTO v\_prix

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

LEFT JOIN Race ON Race.id = Animal.race\_id

WHERE Animal.id = p\_animal\_id\_2;

INSERT INTO Adoption (animal\_id, client\_id, date\_reservation, date\_adoption, prix, paye)

VALUES (p\_animal\_id\_2, p\_client\_id, CURRENT\_DATE(), CURRENT\_DATE(), v\_prix, TRUE);

SELECT 'Adoption animal 2 réussie' AS message;

COMMIT;

END|

DELIMITER ;

CALL adoption\_deux\_ou\_rien(2, 43, 55); -- L'animal 55 a déjà été adopté

La procédure s'interrompt, puisque la seconde insertion échoue. On n'exécute donc pas le second SELECT. Ici, grâce à la transaction et au ROLLBACK  du gestionnaire, la première insertion a été annulée.

#### Préparer une requête dans un bloc d'instructions

Pour finir, on peut créer et exécuter une requête préparée dans un bloc d'instructions. Ceci permet de créer des requêtes dynamiques, puisque l'on prépare une requête à partir d'une chaîne de caractères.

**Exemple :** la procédure suivante ajoute la ou les clauses que l'on veut à une simple requête SELECT.

DELIMITER |

CREATE PROCEDURE select\_race\_dynamique(p\_clause *VARCHAR*(255))

BEGIN

*SET* @sql = CONCAT('SELECT nom, description FROM Race ', p\_clause);

PREPARE requete FROM @sql;

EXECUTE requete;

END|

DELIMITER ;

-- Affichera les races de chats

CALL select\_race\_dynamique('WHERE espece\_id = 2');

-- Affichera les deux premières races par ordre alphabétique de leur nom

CALL select\_race\_dynamique('ORDER BY nom LIMIT 2');

Il va sans dire que ce genre de construction dynamique de requêtes peut poser d'**énormes problèmes de sécurité** si l'on ne prend pas de précaution.  
Par ailleurs, il n'est pas possible de construire une requête préparée à partir d'une variable locale. Il est donc nécessaire d'utiliser une variable utilisateur.

## Triggers

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

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

### Principe et usage

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

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

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

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

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

##### Que fait un trigger ?

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

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

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

#### À quoi sert un trigger ?

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

##### Contraintes et vérifications de données

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

##### Intégrité des données

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

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

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

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

##### Historisation des actions

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

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

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

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

### Création des triggers

#### Syntaxe

Pour créer un trigger, on utilise la commande suivante :

CREATE TRIGGER nom\_trigger moment\_trigger evenement\_trigger

ON nom\_table FOR EACH ROW

corps\_trigger;

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

##### Événement déclencheur

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

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

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

##### Avant ou après

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

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

##### Exemple

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

CREATE TRIGGER after\_insert\_animal AFTER INSERT

ON Animal FOR EACH ROW

corps\_trigger;

#### Règle et convention

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

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

#### OLD et NEW

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

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

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

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

Exécutons la commande suivante :

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, prix, paye)

VALUES (12, 15, NOW(), 200.00, FALSE);

Pendant le traitement de cette ligne par le trigger correspondant,

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

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

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

UPDATE Adoption

*SET* paye = TRUE

WHERE client\_id = 12 AND animal\_id = 15;

Pendant le traitement de cette ligne par le trigger correspondant,

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

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

#### Erreur déclenchée pendant un trigger

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

### Suppression des triggers

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

DROP TRIGGER nom\_trigger;

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

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

### Exemples

#### Contraintes et vérification des données

##### Vérification du sexe des animaux

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

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

-- Trigger déclenché par l'insertion

DELIMITER |

CREATE TRIGGER before\_insert\_animal BEFORE INSERT

ON Animal FOR EACH ROW

BEGIN

-- Instructions

END |

-- Trigger déclenché par la modification

CREATE TRIGGER before\_update\_animal BEFORE UPDATE

ON Animal FOR EACH ROW

BEGIN

-- Instructions

END |

DELIMITER ;

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

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

Quel comportement adopter en cas de valeur erronée ?

Deux possibilités :

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

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

DELIMITER |

CREATE TRIGGER before\_update\_animal BEFORE UPDATE

ON Animal FOR EACH ROW

BEGIN

IF NEW.sexe IS NOT NULL -- le sexe n'est ni NULL

AND NEW.sexe != 'M' -- ni "M"âle

AND NEW.sexe != 'F' -- ni "F"emelle

THEN

*SET* NEW.sexe = NULL;

END IF;

END |

DELIMITER ;

**Test :**

UPDATE Animal

*SET* sexe = 'A'

WHERE id = 20; -- l'animal 20 est Balou, un mâle

SELECT id, sexe, date\_naissance, nom

FROM Animal

WHERE id = 20;

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

Mais comment déclencher une erreur ?

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

**Exemple :**

SELECT 1, 2 INTO @a;

ERROR 1222 (21000): The used SELECT statements have a different number of columns

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

-- Création de la table Erreur

CREATE TABLE Erreur (

id TINYINT UNSIGNED AUTO\_INCREMENT PRIMARY KEY,

erreur *VARCHAR*(255) UNIQUE);

-- Insertion de l'erreur qui nous intéresse

INSERT INTO Erreur (erreur) VALUES ('Erreur : sexe doit valoir "M", "F" ou NULL.');

-- Création du trigger

DELIMITER |

CREATE TRIGGER before\_insert\_animal BEFORE INSERT

ON Animal FOR EACH ROW

BEGIN

IF NEW.sexe IS NOT NULL -- le sexe n'est ni NULL

AND NEW.sexe != 'M' -- ni "M"

AND NEW.sexe != 'F' -- ni "F"

THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : sexe doit valoir "M", "F" ou NULL.');

END IF;

END |

DELIMITER ;

**Test :**

INSERT INTO Animal (nom, sexe, date\_naissance, espece\_id)

VALUES ('Babar', 'A', '2011-08-04 12:34', 3);

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

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

**Vérification du booléen dans *Adoption***

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

INSERT INTO Erreur (erreur) VALUES ('Erreur : paye doit valoir TRUE (1) ou FALSE (0).');

DELIMITER |

CREATE TRIGGER before\_insert\_adoption BEFORE INSERT

ON Adoption FOR EACH ROW

BEGIN

IF NEW.paye != TRUE -- ni TRUE

AND NEW.paye != FALSE -- ni FALSE

THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : paye doit valoir TRUE (1) ou FALSE (0).');

END IF;

END |

CREATE TRIGGER before\_update\_adoption BEFORE UPDATE

ON Adoption FOR EACH ROW

BEGIN

IF NEW.paye != TRUE -- ni TRUE

AND NEW.paye != FALSE -- ni FALSE

THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : paye doit valoir TRUE (1) ou FALSE (0).');

END IF;

END |

DELIMITER ;

**Test :**

UPDATE Adoption

*SET* paye = 3

WHERE client\_id = 9;

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

**Vérification de la date d'adoption**

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

INSERT INTO Erreur (erreur) VALUES ('Erreur : date\_adoption doit être >= à date\_reservation.');

DELIMITER |

DROP TRIGGER before\_insert\_adoption|

CREATE TRIGGER before\_insert\_adoption BEFORE INSERT

ON Adoption FOR EACH ROW

BEGIN

IF NEW.paye != TRUE -- On remet la vérification sur paye

AND NEW.paye != FALSE

THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : paye doit valoir TRUE (1) ou FALSE (0).');

ELSEIF NEW.date\_adoption < NEW.date\_reservation THEN

-- Adoption avant réservation

INSERT INTO Erreur (erreur) VALUES ('Erreur : date\_adoption doit être >= à date\_reservation.');

END IF;

END |

DROP TRIGGER before\_update\_adoption|

CREATE TRIGGER before\_update\_adoption BEFORE UPDATE

ON Adoption FOR EACH ROW

BEGIN

IF NEW.paye != TRUE -- On remet la vérification sur paye

AND NEW.paye != FALSE

THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : paye doit valoir TRUE (1) ou FALSE (0).');

ELSEIF NEW.date\_adoption < NEW.date\_reservation THEN

-- Adoption avant réservation

INSERT INTO Erreur (erreur) VALUES ('Erreur : date\_adoption doit être >= à date\_reservation.');

END IF;

END |

DELIMITER ;

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

**Test :**

INSERT INTO Adoption (animal\_id, client\_id, date\_reservation, date\_adoption, prix, paye)

VALUES (10, 10, NOW(), NOW() - INTERVAL 2 DAY, 200.00, 0);

INSERT INTO Adoption (animal\_id, client\_id, date\_reservation, date\_adoption, prix, paye)

VALUES (10, 10, NOW(), NOW(), 200.00, 4);

ERROR 1062 (23000): Duplicate entry 'Erreur : date\_adoption doit être >= à date\_reservation.' for key 'erreur'

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

Les deux vérifications fonctionnent !

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

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

SELECT id, nom, sexe, date\_naissance, commentaires

FROM Animal

WHERE NOT EXISTS (

SELECT \*

FROM Adoption

WHERE Animal.id = Adoption.animal\_id

);

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

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

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

-- Ajout de la colonne disponible

ALTER TABLE Animal ADD COLUMN disponible BOOLEAN DEFAULT TRUE;

-- À l'insertion, un animal est forcément disponible

-- Remplissage de la colonne

UPDATE Animal

*SET* disponible = FALSE

WHERE EXISTS (

SELECT \*

FROM Adoption

WHERE Animal.id = Adoption.animal\_id

);

-- Création des trois triggers

DELIMITER |

CREATE TRIGGER after\_insert\_adoption AFTER INSERT

ON Adoption FOR EACH ROW

BEGIN

UPDATE Animal

*SET* disponible = FALSE

WHERE id = NEW.animal\_id;

END |

CREATE TRIGGER after\_delete\_adoption AFTER DELETE

ON Adoption FOR EACH ROW

BEGIN

UPDATE Animal

*SET* disponible = TRUE

WHERE id = OLD.animal\_id;

END |

CREATE TRIGGER after\_update\_adoption AFTER UPDATE

ON Adoption FOR EACH ROW

BEGIN

IF OLD.animal\_id <> NEW.animal\_id THEN

UPDATE Animal

*SET* disponible = TRUE

WHERE id = OLD.animal\_id;

UPDATE Animal

*SET* disponible = FALSE

WHERE id = NEW.animal\_id;

END IF;

END |

DELIMITER ;

**Test :**

SELECT animal\_id, nom, sexe, disponible, client\_id

FROM Animal

INNER JOIN Adoption ON Adoption.animal\_id = Animal.id

WHERE client\_id = 9;

DELETE FROM Adoption -- 54 doit redevenir disponible

WHERE animal\_id = 54;

UPDATE Adoption

*SET* animal\_id = 38, prix = 985.00 -- 38 doit devenir indisponible

WHERE animal\_id = 33; -- et 33 redevenir disponible

INSERT INTO Adoption (client\_id, animal\_id, date\_reservation, prix, paye)

VALUES (9, 59, NOW(), 700.00, FALSE); -- 59 doit devenir indisponible

SELECT Animal.id AS animal\_id, nom, sexe, disponible, client\_id

FROM Animal

LEFT JOIN Adoption ON Animal.id = Adoption.animal\_id

WHERE Animal.id IN (33, 54, 55, 38, 59);

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

SELECT \*

FROM Animal

WHERE disponible = TRUE;

-- Ou même

SELECT \*

FROM Animal

WHERE disponible;

**Historisation**

Voici deux exemples de systèmes d'historisation :

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

**Historisation basique**

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

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

-- On modifie la table Race

ALTER TABLE Race

ADD COLUMN date\_insertion DATETIME, -- date d'insertion

ADD COLUMN utilisateur\_insertion *VARCHAR*(20), -- utilisateur ayant inséré la ligne

ADD COLUMN date\_modification DATETIME, -- date de dernière modification

ADD COLUMN utilisateur\_modification *VARCHAR*(20); -- utilisateur ayant fait la dernière modification

-- On remplit les colonnes

UPDATE Race

*SET* date\_insertion = NOW() - INTERVAL 1 DAY,

utilisateur\_insertion = 'Test',

date\_modification = NOW()- INTERVAL 1 DAY,

utilisateur\_modification = 'Test';

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

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

DELIMITER |

CREATE TRIGGER before\_insert\_race BEFORE INSERT

ON Race FOR EACH ROW

BEGIN

*SET* NEW.date\_insertion = NOW();

*SET* NEW.utilisateur\_insertion = CURRENT\_USER();

*SET* NEW.date\_modification = NOW();

*SET* NEW.utilisateur\_modification = CURRENT\_USER();

END |

CREATE TRIGGER before\_update\_race BEFORE UPDATE

ON Race FOR EACH ROW

BEGIN

*SET* NEW.date\_modification = NOW();

*SET* NEW.utilisateur\_modification = CURRENT\_USER();

END |

DELIMITER ;

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

**Test** :

INSERT INTO Race (nom, description, espece\_id, prix)

VALUES ('Yorkshire terrier', 'Chien de petite taille au pelage long et soyeux de couleur bleu et feu.', 1, 700.00);

UPDATE Race

*SET* prix = 630.00

WHERE nom = 'Rottweiller' AND espece\_id = 1;

SELECT nom, *DATE*(date\_insertion) AS date\_ins, utilisateur\_insertion AS utilisateur\_ins, *DATE*(date\_modification) AS date\_mod, utilisateur\_modification AS utilisateur\_mod

FROM Race

WHERE espece\_id = 1;

##### Historisation complète

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

CREATE TABLE Animal\_histo (

id SMALLINT(6) UNSIGNED NOT NULL, -- Colonnes historisées

sexe *CHAR*(1),

date\_naissance DATETIME NOT NULL,

nom *VARCHAR*(30),

commentaires *TEXT*,

espece\_id SMALLINT(6) UNSIGNED NOT NULL,

race\_id SMALLINT(6) UNSIGNED DEFAULT NULL,

mere\_id SMALLINT(6) UNSIGNED DEFAULT NULL,

pere\_id SMALLINT(6) UNSIGNED DEFAULT NULL,

disponible BOOLEAN DEFAULT TRUE,

date\_histo DATETIME NOT NULL, -- Colonnes techniques

utilisateur\_histo *VARCHAR*(20) NOT NULL,

evenement\_histo *CHAR*(6) NOT NULL,

PRIMARY KEY (id, date\_histo)

) ENGINE=InnoDB;

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

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

DELIMITER |

CREATE TRIGGER after\_update\_animal AFTER UPDATE

ON Animal FOR EACH ROW

BEGIN

INSERT INTO Animal\_histo (

id,

sexe,

date\_naissance,

nom,

commentaires,

espece\_id,

race\_id,

mere\_id,

pere\_id,

disponible,

date\_histo,

utilisateur\_histo,

evenement\_histo)

VALUES (

OLD.id,

OLD.sexe,

OLD.date\_naissance,

OLD.nom,

OLD.commentaires,

OLD.espece\_id,

OLD.race\_id,

OLD.mere\_id,

OLD.pere\_id,

OLD.disponible,

NOW(),

CURRENT\_USER(),

'UPDATE');

END |

CREATE TRIGGER after\_delete\_animal AFTER DELETE

ON Animal FOR EACH ROW

BEGIN

INSERT INTO Animal\_histo (

id,

sexe,

date\_naissance,

nom,

commentaires,

espece\_id,

race\_id,

mere\_id,

pere\_id,

disponible,

date\_histo,

utilisateur\_histo,

evenement\_histo)

VALUES (

OLD.id,

OLD.sexe,

OLD.date\_naissance,

OLD.nom,

OLD.commentaires,

OLD.espece\_id,

OLD.race\_id,

OLD.mere\_id,

OLD.pere\_id,

OLD.disponible,

NOW(),

CURRENT\_USER(),

'DELETE');

END |

DELIMITER ;

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

**Test :**

UPDATE Animal

*SET* commentaires = 'Petit pour son âge'

WHERE id = 10;

DELETE FROM Animal

WHERE id = 47;

SELECT id, sexe, date\_naissance, nom, commentaires, espece\_id

FROM Animal

WHERE id IN (10, 47);

SELECT id, nom, date\_histo, utilisateur\_histo, evenement\_histo

FROM Animal\_histo;

##### Quelques remarques sur l'historisation

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

### Restrictions

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

##### Commandes interdites

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

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

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

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

##### Tables utilisées par la requête

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

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

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

INSERT INTO Adoption (animal\_id, client\_id, date\_reservation, prix, paye)

SELECT Animal.id, 4, NOW(), COALESCE(Race.prix, Espece.prix), FALSE

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id

LEFT JOIN Race ON Race.id = Animal.race\_id

WHERE Animal.nom = 'Boucan' AND Animal.espece\_id = 2;

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

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

##### Clés étrangères

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

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

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

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

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

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

Voici les commandes :

-- On supprime les clés

ALTER TABLE Race DROP FOREIGN KEY fk\_race\_espece\_id;

ALTER TABLE Animal DROP FOREIGN KEY fk\_race\_id,

DROP FOREIGN KEY fk\_mere\_id,

DROP FOREIGN KEY fk\_pere\_id;

-- On les recrée sans option

ALTER TABLE Race

ADD CONSTRAINT fk\_race\_espece\_id FOREIGN KEY (espece\_id) REFERENCES Espece (id);

ALTER TABLE Animal

ADD CONSTRAINT fk\_race\_id FOREIGN KEY (race\_id) REFERENCES Race (id),

ADD CONSTRAINT fk\_mere\_id FOREIGN KEY (mere\_id) REFERENCES Animal (id),

ADD CONSTRAINT fk\_pere\_id FOREIGN KEY (pere\_id) REFERENCES Animal (id);

-- Trigger sur Race

DELIMITER |

CREATE TRIGGER before\_delete\_race BEFORE DELETE

ON Race FOR EACH ROW

BEGIN

UPDATE Animal

*SET* race\_id = NULL

WHERE race\_id = OLD.id;

END|

-- Trigger sur Espece

CREATE TRIGGER before\_delete\_espece BEFORE DELETE

ON Espece FOR EACH ROW

BEGIN

DELETE FROM Race

WHERE espece\_id = OLD.id;

END |

DELIMITER ;

## Vues

Les vues sont des objets de la base de données, constitués d'**un nom** et d'une **requête de sélection**.

Une fois qu'une vue est définie, on peut l'utiliser comme on le ferait avec une table qui serait constituée des données sélectionnées par la requête définissant la vue.

Nous verrons dans ce chapitre :

* comment créer, modifier, supprimer une vue ;
* à quoi peut servir une vue ;
* deux algorithmes différents pouvant être utilisés par les vues ;
* comment modifier les données à partir d'une vue.

### État actuel de la base de données

**Note :** les tables de test et les procédures stockées ne sont pas reprises.

*SET* NAMES utf8;

DROP TABLE IF EXISTS Erreur;

DROP TABLE IF EXISTS Animal\_histo;

DROP TABLE IF EXISTS Adoption;

DROP TABLE IF EXISTS Animal;

DROP TABLE IF EXISTS Race;

DROP TABLE IF EXISTS Espece;

DROP TABLE IF EXISTS Client;

CREATE TABLE Client (

id smallint(5) unsigned NOT NULL AUTO\_INCREMENT,

nom *varchar*(100) NOT NULL,

prenom *varchar*(60) NOT NULL,

adresse *varchar*(200) DEFAULT NULL,

code\_postal *varchar*(6) DEFAULT NULL,

ville *varchar*(60) DEFAULT NULL,

pays *varchar*(60) DEFAULT NULL,

email varbinary(100) DEFAULT NULL,

PRIMARY KEY (id),

UNIQUE KEY ind\_uni\_email (email)

) ENGINE=InnoDB AUTO\_INCREMENT=17;

LOCK TABLES Client WRITE;

INSERT INTO Client VALUES (1,'Dupont','Jean','Rue du Centre, 5','45810','Houtsiplou','France','jean.dupont@email.com'),(2,'Boudur','Marie','Place de la Gare, 2','35840','Troudumonde','France','marie.boudur@email.com'),(3,'Trachon','Fleur','Rue haute, 54b','3250','Belville','Belgique','fleurtrachon@email.com'),

(4,'Van Piperseel','Julien',NULL,NULL,NULL,NULL,'jeanvp@email.com'),(5,'Nouvel','Johan',NULL,NULL,NULL,'Suisse','johanetpirlouit@email.com'),(6,'Germain','Frank',NULL,NULL,NULL,NULL,'francoisgermain@email.com'),

(7,'Antoine','Maximilien','Rue Moineau, 123','4580','Trocoul','Belgique','max.antoine@email.com'),(8,'Di Paolo','Hector',NULL,NULL,NULL,'Suisse','hectordipao@email.com'),(9,'Corduro','Anaelle',NULL,NULL,NULL,NULL,'ana.corduro@email.com'),

(10,'Faluche','Eline','Avenue circulaire, 7','45870','Garduche','France','elinefaluche@email.com'),(11,'Penni','Carine','Boulevard Haussman, 85','1514','Plasse','Suisse','cpenni@email.com'),(12,'Broussaille','Virginie','Rue du Fleuve, 18','45810','Houtsiplou','France','vibrousaille@email.com'),

(13,'Durant','Hannah','Rue des Pendus, 66','1514','Plasse','Suisse','hhdurant@email.com'),(14,'Delfour','Elodie','Rue de Flore, 1','3250','Belville','Belgique','e.delfour@email.com'),(15,'Kestau','Joel',NULL,NULL,NULL,NULL,'joel.kestau@email.com');

UNLOCK TABLES;

CREATE TABLE Espece (

id smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

nom\_courant *varchar*(40) NOT NULL,

nom\_latin *varchar*(40) NOT NULL,

description *text*,

prix *decimal*(7,2) unsigned DEFAULT NULL,

PRIMARY KEY (id),

UNIQUE KEY nom\_latin (nom\_latin)

) ENGINE=InnoDB AUTO\_INCREMENT=6;

LOCK TABLES Espece WRITE;

INSERT INTO Espece VALUES (1,'Chien','Canis canis','Bestiole à quatre pattes qui aime les caresses et tire souvent la langue',200.00),(2,'Chat','Felis silvestris','Bestiole à quatre pattes qui saute très haut et grimpe aux arbres',150.00),(3,'Tortue d''Hermann','Testudo hermanni','Bestiole avec une carapace très dure',140.00),

(4,'Perroquet amazone','Alipiopsitta xanthops','Joli oiseau parleur vert et jaune',700.00),(5,'Rat brun','Rattus norvegicus','Petite bestiole avec de longues moustaches et une longue queue sans poils',10.00);

UNLOCK TABLES;

CREATE TABLE Race (

id smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

nom *varchar*(40) NOT NULL,

espece\_id smallint(6) unsigned NOT NULL,

description *text*,

prix *decimal*(7,2) unsigned DEFAULT NULL,

date\_insertion datetime DEFAULT NULL,

utilisateur\_insertion *varchar*(20) DEFAULT NULL,

date\_modification datetime DEFAULT NULL,

utilisateur\_modification *varchar*(20) DEFAULT NULL,

PRIMARY KEY (id)

) ENGINE=InnoDB AUTO\_INCREMENT=11;

LOCK TABLES Race WRITE;

INSERT INTO Race VALUES (1,'Berger allemand',1,'Chien sportif et élégant au pelage dense, noir-marron-fauve, noir ou gris.',485.00,'2012-05-21 00:53:36','Test','2012-05-21 00:53:36','Test'),(2,'Berger blanc suisse',1,'Petit chien au corps compact, avec des pattes courtes mais bien proportionnées et au pelage tricolore ou bicolore.',935.00,'2012-05-21 00:53:36','Test','2012-05-21 00:53:36','Test'),(3,'Singapura',2,'Chat de petite taille aux grands yeux en amandes.',985.00,'2012-05-21 00:53:36','Test','2012-05-21 00:53:36','Test'),

(4,'Bleu russe',2,'Chat aux yeux verts et à la robe épaisse et argentée.',835.00,'2012-05-21 00:53:36','Test','2012-05-21 00:53:36','Test'),(5,'Maine coon',2,'Chat de grande taille, à poils mi-longs.',735.00,'2012-05-21 00:53:36','Test','2012-05-21 00:53:36','Test'),(7,'Sphynx',2,'Chat sans poils.',1235.00,'2012-05-21 00:53:36','Test','2012-05-21 00:53:36','Test'),

(8,'Nebelung',2,'Chat bleu russe, mais avec des poils longs...',985.00,'2012-05-21 00:53:36','Test','2012-05-21 00:53:36','Test'),(9,'Rottweiller',1,'Chien d''apparence solide, bien musclé, à la robe noire avec des taches feu bien délimitées.',630.00,'2012-05-21 00:53:36','Test','2012-05-22 00:54:13','sdz@localhost'),(10,'Yorkshire terrier',1,'Chien de petite taille au pelage long et soyeux de couleur bleu et feu.',700.00,'2012-05-22 00:58:25','sdz@localhost','2012-05-22 00:58:25','sdz@localhost');

UNLOCK TABLES;

CREATE TABLE Animal (

id smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

sexe *char*(1) DEFAULT NULL,

date\_naissance datetime NOT NULL,

nom *varchar*(30) DEFAULT NULL,

commentaires *text*,

espece\_id smallint(6) unsigned NOT NULL,

race\_id smallint(6) unsigned DEFAULT NULL,

mere\_id smallint(6) unsigned DEFAULT NULL,

pere\_id smallint(6) unsigned DEFAULT NULL,

disponible tinyint(1) DEFAULT 1,

PRIMARY KEY (id),

UNIQUE KEY ind\_uni\_nom\_espece\_id (nom,espece\_id)

) ENGINE=InnoDB AUTO\_INCREMENT=76;

LOCK TABLES Animal WRITE;

INSERT INTO Animal VALUES (1,'M','2010-04-05 13:43:00','Rox','Mordille beaucoup',1,1,18,22,1),(2,NULL,'2010-03-24 02:23:00','Roucky',NULL,2,NULL,40,30,1),(3,'F','2010-09-13 15:02:00','Schtroumpfette',NULL,2,4,41,31,0),

(4,'F','2009-08-03 05:12:00',NULL,'Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),(5,NULL,'2010-10-03 16:44:00','Choupi','Né sans oreille gauche',2,NULL,NULL,NULL,0),(6,'F','2009-06-13 08:17:00','Bobosse','Carapace bizarre',3,NULL,NULL,NULL,1),

(7,'F','2008-12-06 05:18:00','Caroline',NULL,1,2,NULL,NULL,1),(8,'M','2008-09-11 15:38:00','Bagherra',NULL,2,5,NULL,NULL,0),(9,NULL,'2010-08-23 05:18:00',NULL,'Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),

(10,'M','2010-07-21 15:41:00','Bobo','Petit pour son âge',1,NULL,7,21,1),(11,'F','2008-02-20 15:45:00','Canaille',NULL,1,NULL,NULL,NULL,1),(12,'F','2009-05-26 08:54:00','Cali',NULL,1,2,NULL,NULL,1),

(13,'F','2007-04-24 12:54:00','Rouquine',NULL,1,1,NULL,NULL,1),(14,'F','2009-05-26 08:56:00','Fila',NULL,1,2,NULL,NULL,1),(15,'F','2008-02-20 15:47:00','Anya',NULL,1,NULL,NULL,NULL,0),

(16,'F','2009-05-26 08:50:00','Louya',NULL,1,NULL,NULL,NULL,0),(17,'F','2008-03-10 13:45:00','Welva',NULL,1,NULL,NULL,NULL,1),(18,'F','2007-04-24 12:59:00','Zira',NULL,1,1,NULL,NULL,0),

(19,'F','2009-05-26 09:02:00','Java',NULL,1,2,NULL,NULL,1),(20,NULL,'2007-04-24 12:45:00','Balou',NULL,1,1,NULL,NULL,1),(21,'F','2008-03-10 13:43:00','Pataude',NULL,1,NULL,NULL,NULL,0),

(22,'M','2007-04-24 12:42:00','Bouli',NULL,1,1,NULL,NULL,1),(24,'M','2007-04-12 05:23:00','Cartouche',NULL,1,NULL,NULL,NULL,1),(25,'M','2006-05-14 15:50:00','Zambo',NULL,1,1,NULL,NULL,1),

(26,'M','2006-05-14 15:48:00','Samba',NULL,1,1,NULL,NULL,0),(27,'M','2008-03-10 13:40:00','Moka',NULL,1,NULL,NULL,NULL,0),(28,'M','2006-05-14 15:40:00','Pilou',NULL,1,1,NULL,NULL,1),

(29,'M','2009-05-14 06:30:00','Fiero',NULL,2,3,NULL,NULL,1),(30,'M','2007-03-12 12:05:00','Zonko',NULL,2,5,NULL,NULL,0),(31,'M','2008-02-20 15:45:00','Filou',NULL,2,4,NULL,NULL,1),

(32,'M','2009-07-26 11:52:00','Spoutnik',NULL,3,NULL,52,NULL,0),(33,'M','2006-05-19 16:17:00','Caribou',NULL,2,4,NULL,NULL,1),(34,'M','2008-04-20 03:22:00','Capou',NULL,2,5,NULL,NULL,1),

(35,'M','2006-05-19 16:56:00','Raccou','Pas de queue depuis la naissance',2,4,NULL,NULL,1),(36,'M','2009-05-14 06:42:00','Boucan',NULL,2,3,NULL,NULL,1),(37,'F','2006-05-19 16:06:00','Callune',NULL,2,8,NULL,NULL,1),

(38,'F','2009-05-14 06:45:00','Boule',NULL,2,3,NULL,NULL,0),(39,'F','2008-04-20 03:26:00','Zara',NULL,2,5,NULL,NULL,0),(40,'F','2007-03-12 12:00:00','Milla',NULL,2,5,NULL,NULL,0),

(41,'F','2006-05-19 15:59:00','Feta',NULL,2,4,NULL,NULL,0),(42,'F','2008-04-20 03:20:00','Bilba','Sourde de l''oreille droite à 80%',2,5,NULL,NULL,0),(43,'F','2007-03-12 11:54:00','Cracotte',NULL,2,5,NULL,NULL,1),

(44,'F','2006-05-19 16:16:00','Cawette',NULL,2,8,NULL,NULL,1),(45,'F','2007-04-01 18:17:00','Nikki','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),(46,'F','2009-03-24 08:23:00','Tortilla','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),

(48,'F','2006-03-15 14:56:00','Lulla','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),(49,'F','2008-03-15 12:02:00','Dana','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,0),(50,'F','2009-05-25 19:57:00','Cheli','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),

(51,'F','2007-04-01 03:54:00','Chicaca','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),(52,'F','2006-03-15 14:26:00','Redbul','Insomniaque',3,NULL,NULL,NULL,1),(54,'M','2008-03-16 08:20:00','Bubulle','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),

(55,'M','2008-03-15 18:45:00','Relou','Surpoids',3,NULL,NULL,NULL,0),(56,'M','2009-05-25 18:54:00','Bulbizard','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),(57,'M','2007-03-04 19:36:00','Safran','Coco veut un gâteau !',4,NULL,NULL,NULL,0),

(58,'M','2008-02-20 02:50:00','Gingko','Coco veut un gâteau !',4,NULL,NULL,NULL,0),(59,'M','2009-03-26 08:28:00','Bavard','Coco veut un gâteau !',4,NULL,NULL,NULL,0),(60,'F','2009-03-26 07:55:00','Parlotte','Coco veut un gâteau !',4,NULL,NULL,NULL,1),

(61,'M','2010-11-09 00:00:00','Yoda',NULL,2,5,NULL,NULL,1),(62,'M','2010-11-05 00:00:00','Pipo',NULL,1,9,NULL,NULL,0),(69,'F','2012-02-13 15:45:00','Baba',NULL,5,NULL,NULL,NULL,0),

(70,'M','2012-02-13 15:48:00','Bibo','Agressif',5,NULL,72,73,1),(72,'F','2008-02-01 02:25:00','Momy',NULL,5,NULL,NULL,NULL,1),(73,'M','2007-03-11 12:45:00','Popi',NULL,5,NULL,NULL,NULL,1),

(75,'F','2007-03-12 22:03:00','Mimi',NULL,5,NULL,NULL,NULL,0);

UNLOCK TABLES;

CREATE TABLE Adoption (

client\_id smallint(5) unsigned NOT NULL,

animal\_id smallint(5) unsigned NOT NULL,

date\_reservation *date* NOT NULL,

date\_adoption *date* DEFAULT NULL,

prix *decimal*(7,2) unsigned NOT NULL,

paye tinyint(1) NOT NULL DEFAULT '0',

PRIMARY KEY (client\_id,animal\_id),

UNIQUE KEY ind\_uni\_animal\_id (animal\_id)

) ENGINE=InnoDB;

LOCK TABLES Adoption WRITE;

INSERT INTO Adoption VALUES (1,8,'2012-05-21',NULL,735.00,1),(1,39,'2008-08-17','2008-08-17',735.00,1),(1,40,'2008-08-17','2008-08-17',735.00,1),

(2,3,'2011-03-12','2011-03-12',835.00,1),(2,18,'2008-06-04','2008-06-04',485.00,1),(3,27,'2009-11-17','2009-11-17',200.00,1),

(4,26,'2007-02-21','2007-02-21',485.00,1),(4,41,'2007-02-21','2007-02-21',835.00,1),(5,21,'2009-03-08','2009-03-08',200.00,1),

(6,16,'2010-01-27','2010-01-27',200.00,1),(7,5,'2011-04-05','2011-04-05',150.00,1),(8,42,'2008-08-16','2008-08-16',735.00,1),

(9,38,'2007-02-11','2007-02-11',985.00,1),(9,55,'2011-02-13','2011-02-13',140.00,1),(9,59,'2012-05-22',NULL,700.00,0),

(10,49,'2010-08-17','2010-08-17',140.00,0),(11,32,'2008-08-17','2010-03-09',140.00,1),(11,62,'2011-03-01','2011-03-01',630.00,0),

(12,15,'2012-05-22',NULL,200.00,1),(12,69,'2007-09-20','2007-09-20',10.00,1),(12,75,'2012-05-21',NULL,10.00,0),

(13,57,'2012-01-10','2012-01-10',700.00,1),(14,58,'2012-02-25','2012-02-25',700.00,1),(15,30,'2008-08-17','2008-08-17',735.00,1);

UNLOCK TABLES;

ALTER TABLE Race ADD CONSTRAINT fk\_race\_espece\_id FOREIGN KEY (espece\_id) REFERENCES Espece (id);

ALTER TABLE Animal ADD CONSTRAINT fk\_race\_id FOREIGN KEY (race\_id) REFERENCES Race (id);

ALTER TABLE Animal ADD CONSTRAINT fk\_mere\_id FOREIGN KEY (mere\_id) REFERENCES Animal (id);

ALTER TABLE Animal ADD CONSTRAINT fk\_pere\_id FOREIGN KEY (pere\_id) REFERENCES Animal (id);

ALTER TABLE Animal ADD CONSTRAINT fk\_espece\_id FOREIGN KEY (espece\_id) REFERENCES Espece (id);

ALTER TABLE Adoption ADD CONSTRAINT fk\_client\_id FOREIGN KEY (client\_id) REFERENCES Client (id);

ALTER TABLE Adoption ADD CONSTRAINT fk\_adoption\_animal\_id FOREIGN KEY (animal\_id) REFERENCES Animal (id);

CREATE TABLE Animal\_histo (

id smallint(6) unsigned NOT NULL,

sexe *char*(1) DEFAULT NULL,

date\_naissance datetime NOT NULL,

nom *varchar*(30) DEFAULT NULL,

commentaires *text*,

espece\_id smallint(6) unsigned NOT NULL,

race\_id smallint(6) unsigned DEFAULT NULL,

mere\_id smallint(6) unsigned DEFAULT NULL,

pere\_id smallint(6) unsigned DEFAULT NULL,

disponible tinyint(1) DEFAULT 1,

date\_histo datetime NOT NULL,

utilisateur\_histo *varchar*(20) NOT NULL,

evenement\_histo *char*(6) NOT NULL,

PRIMARY KEY (id,date\_histo)

) ENGINE=InnoDB;

LOCK TABLES Animal\_histo WRITE;

INSERT INTO Animal\_histo VALUES (10,'M','2010-07-21 15:41:00','Bobo',NULL,1,NULL,7,21,1,'2012-05-22 01:00:34','sdz@localhost','UPDATE'),(47,'F','2009-03-26 01:24:00','Scroupy','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1,'2012-05-22 01:00:34','sdz@localhost','DELETE');

UNLOCK TABLES;

CREATE TABLE Erreur (

id tinyint(3) unsigned NOT NULL AUTO\_INCREMENT,

erreur *varchar*(255) DEFAULT NULL,

PRIMARY KEY (id),

UNIQUE KEY erreur (erreur)

) ENGINE=InnoDB AUTO\_INCREMENT=8;

LOCK TABLES Erreur WRITE;

INSERT INTO Erreur VALUES (5,'Erreur : date\_adoption doit être >= à date\_reservation.'),(3,'Erreur : paye doit valoir TRUE (1) ou FALSE (0).'),(1,'Erreur : sexe doit valoir \"M\", \"F\", \"A\" ou NULL.');

UNLOCK TABLES;

-- -------- --

-- TRIGGERS --

-- -------- --

DELIMITER |

CREATE TRIGGER before\_insert\_adoption BEFORE INSERT

ON Adoption FOR EACH ROW

BEGIN

IF NEW.paye != TRUE

AND NEW.paye != FALSE

THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : paye doit valoir TRUE (1) ou FALSE (0).');

ELSEIF NEW.date\_adoption < NEW.date\_reservation THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : date\_adoption doit être >= à date\_reservation.');

END IF;

END |

CREATE TRIGGER after\_insert\_adoption AFTER INSERT

ON Adoption FOR EACH ROW

BEGIN

UPDATE Animal

*SET* disponible = FALSE

WHERE id = NEW.animal\_id;

END |

CREATE TRIGGER before\_update\_adoption BEFORE UPDATE

ON Adoption FOR EACH ROW

BEGIN

IF NEW.paye != TRUE

AND NEW.paye != FALSE

THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : paye doit valoir TRUE (1) ou FALSE (0).');

ELSEIF NEW.date\_adoption < NEW.date\_reservation THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : date\_adoption doit être >= à date\_reservation.');

END IF;

END |

CREATE TRIGGER after\_update\_adoption AFTER UPDATE

ON Adoption FOR EACH ROW

BEGIN

IF OLD.animal\_id <> NEW.animal\_id THEN

UPDATE Animal

*SET* disponible = TRUE

WHERE id = OLD.animal\_id;

UPDATE Animal

*SET* disponible = FALSE

WHERE id = NEW.animal\_id;

END IF;

END |

CREATE TRIGGER after\_delete\_adoption AFTER DELETE

ON Adoption FOR EACH ROW

BEGIN

UPDATE Animal

*SET* disponible = TRUE

WHERE id = OLD.animal\_id;

END |

CREATE TRIGGER before\_insert\_animal BEFORE INSERT

ON Animal FOR EACH ROW

BEGIN

IF NEW.sexe IS NOT NULL

AND NEW.sexe != 'M'

AND NEW.sexe != 'F'

AND NEW.sexe != 'A'

THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : sexe doit valoir "M", "F", "A" ou NULL.');

END IF;

END |

CREATE TRIGGER before\_update\_animal BEFORE UPDATE

ON Animal FOR EACH ROW

BEGIN

IF NEW.sexe IS NOT NULL

AND NEW.sexe != 'M'

AND NEW.sexe != 'F'

AND NEW.sexe != 'A'

THEN

*SET* NEW.sexe = NULL;

END IF;

END |

CREATE TRIGGER after\_update\_animal AFTER UPDATE

ON Animal FOR EACH ROW

BEGIN

INSERT INTO Animal\_histo (

id, sexe, date\_naissance, nom, commentaires, espece\_id, race\_id,

mere\_id, pere\_id, disponible, date\_histo, utilisateur\_histo, evenement\_histo)

VALUES (

OLD.id, OLD.sexe, OLD.date\_naissance, OLD.nom, OLD.commentaires, OLD.espece\_id, OLD.race\_id,

OLD.mere\_id, OLD.pere\_id, OLD.disponible, NOW(), CURRENT\_USER(), 'UPDATE');

END |

CREATE TRIGGER after\_delete\_animal AFTER DELETE

ON Animal FOR EACH ROW

BEGIN

INSERT INTO Animal\_histo (

id, sexe, date\_naissance, nom, commentaires, espece\_id, race\_id,

mere\_id, pere\_id, disponible, date\_histo, utilisateur\_histo, evenement\_histo)

VALUES (

OLD.id, OLD.sexe, OLD.date\_naissance, OLD.nom, OLD.commentaires, OLD.espece\_id, OLD.race\_id,

OLD.mere\_id, OLD.pere\_id, OLD.disponible, NOW(), CURRENT\_USER(), 'DELETE');

END |

CREATE TRIGGER before\_delete\_espece BEFORE DELETE

ON Espece FOR EACH ROW

BEGIN

DELETE FROM Race

WHERE espece\_id = OLD.id;

END |

CREATE TRIGGER before\_insert\_race BEFORE INSERT

ON Race FOR EACH ROW

BEGIN

*SET* NEW.date\_insertion = NOW();

*SET* NEW.utilisateur\_insertion = CURRENT\_USER();

*SET* NEW.date\_modification = NOW();

*SET* NEW.utilisateur\_modification = CURRENT\_USER();

END |

CREATE TRIGGER before\_update\_race BEFORE UPDATE

ON Race FOR EACH ROW

BEGIN

*SET* NEW.date\_modification = NOW();

*SET* NEW.utilisateur\_modification = CURRENT\_USER();

END |

CREATE TRIGGER before\_delete\_race BEFORE DELETE

ON Race FOR EACH ROW

BEGIN

UPDATE Animal

*SET* race\_id = NULL

WHERE race\_id = OLD.id;

END |

DELIMITER ;

### Création d'une vue

#### Le principe

Pour notre élevage, la requête suivante est très utile.

SELECT Animal.id, Animal.sexe, Animal.date\_naissance, Animal.nom, Animal.commentaires,

Animal.espece\_id, Animal.race\_id, Animal.mere\_id, Animal.pere\_id, Animal.disponible,

Espece.nom\_courant AS espece\_nom, Race.nom AS race\_nom

FROM Animal

INNER JOIN Espece ON Animal.espece\_id = Espece.id

LEFT JOIN Race ON Animal.race\_id = Race.id;

Avec ou sans clause WHERE, il arrive régulièrement que l'on veuille trouver des renseignements sur nos animaux, y compris leur race et leur espèce (et seul, l'id contenu dans Animal n'est pas une information très explicite). Il serait donc bien pratique de pouvoir stocker cette requête plutôt que de devoir la retaper en entier à chaque fois.

C'est très exactement le principe d'une vue : **on stocke une requête SELECT  en lui donnant un nom**, et on peut ensuite appeler directement la vue par son nom.

Quelques remarques importantes :

* Il s'agit bien d'objets de la base de données, **stockés de manière durable**, comme le sont les tables ou les procédures stockées.
* C'est donc bien différent des requêtes préparées, qui ne sont définies que le temps d'une session, et qui ont un tout autre but.
* Ce qui est stocké est la **requête**, et non pas les résultats de celle-ci. **On ne gagne absolument rien en termes de performances** en utilisant une vue plutôt qu'en faisant une requête directement sur les tables.

#### Création

Pour créer une vue, on utilise tout simplement la commande CREATE VIEW, dont voici la syntaxe :

CREATE [OR REPLACE] VIEW nom\_vue

AS requete\_select;

La clause OR REPLACE  est facultative. Si elle est fournie, la vue nom\_vue sera soit créée, si elle n'existe pas, soit remplacée, si elle existait déjà. Si OR REPLACE  est omise et qu'une vue portant le même nom a été précédemment définie, cela déclenchera une erreur.

Donc, si l'on reprend la requête précédente, voici comment créer une vue pour stocker celle-ci :

CREATE VIEW V\_Animal\_details

AS SELECT Animal.id, Animal.sexe, Animal.date\_naissance, Animal.nom, Animal.commentaires,

Animal.espece\_id, Animal.race\_id, Animal.mere\_id, Animal.pere\_id, Animal.disponible,

Espece.nom\_courant AS espece\_nom, Race.nom AS race\_nom

FROM Animal

INNER JOIN Espece ON Animal.espece\_id = Espece.id

LEFT JOIN Race ON Animal.race\_id = Race.id;

Dorénavant, plus besoin de retaper cette requête, il suffit de travailler à partir de la vue, comme s'il s'agissait d'une table :

SELECT \* FROM V\_Animal\_details;

J'ai préfixé le nom de la vue par "V\_". Il s'agit d'une convention que je vous conseille fortement de respecter. Cela permet de savoir au premier coup d’œil si l'on travaille avec une vraie table, ou avec une vue.

D'ailleurs, si l'on demande la liste des tables de la base de données, on peut voir que la vue est reprise (bien qu'il ne s'agisse pas d'une table, mais d'une requête SELECT  stockée, j'insiste).

SHOW TABLES;

#### Les colonnes de la vue

Comme pour les tables, on peut utiliser la commande DESCRIBE  pour voir les différentes colonnes de notre vue.

DESCRIBE V\_Animal\_details;

Comme on pouvait s'y attendre, les noms de colonnes ont été déterminés par la clause SELECT  de la requête définissant la vue. S'il n'y avait pas d'alias pour la colonne, c'est simplement le même nom que dans la table d'origine (*date\_naissance,*par exemple), et si un alias a été donné, il est utilisé (*espece\_nom,*par exemple).

**Lister les colonnes dans  CREATE VIEW**

Il existe une autre possibilité que le recours aux alias dans la clause SELECT  pour nommer les colonnes d'une vue. Il suffit de lister les colonnes juste après le nom de la vue, lors de la création de celle-ci. La commande suivante est l'équivalent de la précédente pour la création de *V\_Animal\_details*, mais cette fois sans alias dans la requête SELECT.

CREATE VIEW V\_Animal\_details (id, sexe, date\_naissance, nom, commentaires, espece\_id, race\_id, mere\_id, pere\_id, disponible, espece\_nom, race\_nom)

AS SELECT Animal.id, Animal.sexe, Animal.date\_naissance, Animal.nom, Animal.commentaires,

Animal.espece\_id, Animal.race\_id, Animal.mere\_id, Animal.pere\_id, Animal.disponible,

Espece.nom\_courant, Race.nom

FROM Animal

INNER JOIN Espece ON Animal.espece\_id = Espece.id

LEFT JOIN Race ON Animal.race\_id = Race.id;

Il faut toujours vérifier que les colonnes sélectionnées sont dans le bon ordre par rapport à la liste des noms, car la correspondance se fait uniquement sur la base de la position. MySQL ne lèvera pas le petit doigt si l'on nomme "espece" une colonne contenant le sexe.

**Doublons interdits**

Comme pour une table, il est impossible que deux colonnes ayant le même nom cohabitent dans une même vue.

CREATE VIEW V\_test

AS SELECT Animal.id, Espece.id

FROM Animal

INNER JOIN Espece ON Animal.espece\_id = Espece.id;

ERROR 1060 (42S21): Duplicate column name 'id'

Pour pouvoir créer cette vue, il est nécessaire de renommer une des deux colonnes dans la vue (soit avec un alias, soit en listant les colonnes juste après CREATE VIEW nom\_vue).

**Requête SELECT  stockée dans la vue**

Que peut-on mettre dans la requête SELECT  qui sert à définir la vue ?

La requête définissant une vue peut être n'importe quelle requête SELECT, à quelques exceptions près.

* Il n'est pas possible d'utiliser une requête SELECT  dont la clause FROM  contient une sous-requête SELECT.
* La requête ne peut pas faire référence à des variables utilisateur, à des variables système ni même à des variables locales (dans le cas d'une vue définie par une procédure stockée).
* Toutes les tables (ou vues) mentionnées dans la requête doivent exister (au moment de la création du moins).

En dehors de cela, carte blanche ! La requête peut contenir une clause WHERE, une clause GROUP BY, des fonctions (scalaires ou d’agrégation), des opérations mathématiques, une autre vue, des jointures, etc.

**Exemple 1 :** une vue pour les chiens.

L'employé de l'élevage préposé aux chiens peut créer une vue sur ceux-ci afin de ne pas devoir ajouter une clause WHERE espece\_id = 1  à chacune de ses requêtes.

CREATE VIEW V\_Chien

AS SELECT id, sexe, date\_naissance, nom, commentaires, espece\_id, race\_id, mere\_id, pere\_id, disponible

FROM Animal

WHERE espece\_id = 1;

**Exemple 2 :** combien de chats possédons-nous ?

Il est tout à fait possible de créer une vue définie par une requête qui compte le nombre d'animaux de chaque espèce que l'on possède.

CREATE OR REPLACE VIEW V\_Nombre\_espece

AS SELECT Espece.id, COUNT(Animal.id) AS nb

FROM Espece

LEFT JOIN Animal ON Animal.espece\_id = Espece.id

GROUP BY Espece.id;

**Exemple 3 :** vue sur une vue.

La vue *V\_Chien*sélectionne les chiens. Créons la vue *V\_Chien\_race*qui sélectionne les chiens dont on connaît la race. On peut bien sûr créer cette vue à partir de la table *Animal*, mais on peut aussi repartir simplement de la vue *V\_Chien*.

CREATE OR REPLACE VIEW V\_Chien\_race

AS SELECT id, sexe, date\_naissance, nom, commentaires, espece\_id, race\_id, mere\_id, pere\_id, disponible

FROM V\_Chien

WHERE race\_id IS NOT NULL;

**Exemple 4 :** expression dans une vue.

Voici un exemple de vue définie par une requête contenant une expression, qui sélectionne les espèces, avec leur prix en dollars.

CREATE VIEW V\_Espece\_dollars

AS SELECT id, nom\_courant, nom\_latin, description, ROUND(prix\*1.31564, 2) AS prix\_dollars

FROM Espece;

**La requête SELECT  est "figée"**

La requête SELECT  définissant une vue est figée : les changements de structure faits par la suite sur la ou les tables sous-jacentes n'influent pas sur la vue.

Par exemple, si l'on crée une vue *V\_Client* toute simple.

CREATE VIEW V\_Client

AS SELECT \*

FROM Client;

Cette vue sélectionne les huit colonnes de *Client(id, nom, prenom, adresse, code\_postal, ville, pays, email)*. Une fois cette vue créée, elle est figée, et l'ajout d'une colonne dans la table *Client***ne changera pas** les résultats d'une requête sur *V\_Client*. Cette vue ne sélectionnera jamais que *id, nom, prenom, adresse, code\_postal, ville, pays* et *email*, malgré le caractère \* dans la clauseSELECT. Pour que *V\_Client* sélectionne la nouvelle colonne de *Client*, il faudrait recréer la vue pour l'inclure.

**Exemple :** ajout d'une colonne *date\_naissance* à la table *Client*.

ALTER TABLE Client ADD COLUMN date\_naissance *DATE*;

DESCRIBE V\_Client;

##### Tri des données directement dans la vue

Si l'on met un ORDER BY  dans la requête définissant une vue, celui-ci prendra effet, sauf si l'on fait une requête sur la vue avec un ORDER BY. Dans ce cas, c'est ce dernier qui prime, et l'ORDER BY  originel est ignoré.

**Exemple**

CREATE OR REPLACE VIEW V\_Race

AS SELECT Race.id, nom, Espece.nom\_courant AS espece

FROM Race

INNER JOIN Espece ON Espece.id = Race.espece\_id

ORDER BY nom;

SELECT \*

FROM V\_Race;

-- Sélection sans ORDER BY, on prend l'ORDER BY de la définition

SELECT \*

FROM V\_Race

ORDER BY espece;

-- Sélection avec ORDER BY, c'est celui-là qui sera pris en compte

La première requête donnera bien les races par ordre alphabétique ; les races de chiens et les races de chats seront mélangées. Par contre, dans la deuxième, on a d'abord toutes les races de chats, puis toutes les races de chiens.

##### Comportement d'autres clauses de SELECT

En ce qui concerne notamment les clauses LIMIT  et DISTINCT  (et son opposé, ALL), le comportement est indéfini. Dans le cas où l'on fait une requête avec un LIMIT  sur une vue dont la requête possède aussi un LIMIT, on ne sait pas quelle clause, de la définition ou de la sélection, sera appliquée. Par conséquent, il est déconseillé d'inclure ces clauses dans la définition d'une vue.

### Sélection des données d'une vue

Une fois la vue créée, on peut bien entendu faire plus qu'un simple SELECT \* FROM la\_vue;  : on peut tout simplement traiter cette vue comme une table, et donc ajouter des clauses WHERE, GROUP BY, des fonctions, des jointures et tout ce que l'on veut !

**Exemple 1 :** on sélectionne les rats bruns à partir de la vue V\_Animal\_details.

SELECT id, nom, espece\_nom, date\_naissance, commentaires, disponible

FROM V\_Animal\_details

WHERE espece\_nom = 'Rat brun';

**Exemple 2 :** on sélectionne le nombre d'animaux par espèce avec la vue *V\_Nombre\_espece*, en ajoutant une jointure sur la table *Espece* pour avoir le nom des espèces en plus de leur *id*.

SELECT V\_Nombre\_espece.id, Espece.nom\_courant, V\_Nombre\_espece.nb

FROM V\_Nombre\_espece

INNER JOIN Espece ON Espece.id = V\_Nombre\_espece.id;

**Exemple 3 :** on sélectionne le nombre de chiens que l'on possède pour chaque race en utilisant un regroupement sur la vue *V\_Chien\_race*, avec jointure sur *Race* pour avoir le nom de la race.

SELECT Race.nom, COUNT(V\_Chien\_race.id)

FROM Race

INNER JOIN V\_Chien\_race ON Race.id = V\_Chien\_race.race\_id

GROUP BY Race.nom;

### Modification et suppression d'une vue

#### Modification

##### CREATE OR REPLACE

Comme déjà mentionné, pour modifier une vue, on peut tout simplement ajouter la clause OR REPLACE  à la requête de création de la vue, et lui donner le nom de la vue à modifier. L'ancienne vue sera alors remplacée.

Si le nom de votre vue est incorrect et ne correspond à aucune vue existante, aucune erreur ne sera renvoyée, et vous créerez tout simplement une nouvelle vue.

##### ALTER

Il existe aussi la commande ALTER VIEW, qui aura le même effet que CREATE OR REPLACE VIEW, si la vue existe bel et bien. Dans le cas contraire, ALTER VIEW  générera une erreur.

Syntaxe :

ALTER VIEW nom\_vue [(liste\_colonnes)]

AS requete\_select

**Exemples :** les deux requêtes suivantes ont exactement le même effet ; on modifie la vue V\_espece\_dollars pour mettre à jour le taux de change du dollar.

CREATE OR REPLACE VIEW V\_Espece\_dollars

AS SELECT id, nom\_courant, nom\_latin, description, ROUND(prix\*1.30813, 2) AS prix\_dollars

FROM Espece;

ALTER VIEW V\_Espece\_dollars

AS SELECT id, nom\_courant, nom\_latin, description, ROUND(prix\*1.30813, 2) AS prix\_dollars

FROM Espece;

#### Suppression

Pour supprimer une vue, on utilise simplement DROP VIEW [IF EXISTS] nom\_vue;.

**Exemple :** suppression de V\_Race.

DROP VIEW V\_Race;

### Utilité des vues

Au-delà de la légendaire paresse des informaticiens, bien contents de ne pas devoir retaper la même requête encore et encore, les vues sont utilisées pour différentes raisons, dont voici les principales.

#### Clarification et facilitation des requêtes

Lorsqu'une requête implique un grand nombre de tables ou nécessite des fonctions, des regroupements, etc., même une bonne indentation ne suffit pas toujours à rendre la requête claire. Qui plus est, plus la requête est longue et complexe, plus l'ajout de la moindre condition peut se révéler pénible.  
Avoir une vue pour des requêtes complexes permet de simplifier et de clarifier les requêtes.

**Exemple :** on veut savoir quelles espèces rapportent le plus, année après année. Comme c'est une question importante pour le développement de l'élevage et qu'elle reviendra souvent, on crée une vue que l'on pourra interroger facilement.

CREATE OR REPLACE VIEW V\_Revenus\_annee\_espece

AS SELECT YEAR(date\_reservation) AS annee, Espece.id AS espece\_id, SUM(Adoption.prix) AS somme, COUNT(Adoption.animal\_id) AS nb

FROM Adoption

INNER JOIN Animal ON Animal.id = Adoption.animal\_id

INNER JOIN Espece ON Animal.espece\_id = Espece.id

GROUP BY annee, Espece.id;

Avec des requêtes toutes simples, on peut maintenant obtenir des informations résultant de requêtes complexes, par exemple :

**1.** Les revenus obtenus par année

SELECT annee, SUM(somme) AS total

FROM V\_Revenus\_annee\_espece

GROUP BY annee;

**2.** Les revenus obtenus pour chaque espèce, toutes années confondues

SELECT Espece.nom\_courant AS espece, SUM(somme) AS total

FROM V\_Revenus\_annee\_espece

INNER JOIN Espece ON V\_Revenus\_annee\_espece.espece\_id = Espece.id

GROUP BY espece;

**3.** Les revenus moyens générés par la vente d'un individu de l'espèce

SELECT Espece.nom\_courant AS espece, SUM(somme)/SUM(nb) AS moyenne

FROM V\_Revenus\_annee\_espece

INNER JOIN Espece ON V\_Revenus\_annee\_espece.espece\_id = Espece.id

GROUP BY espece;

#### Création d'une interface entre l'application et la base de données

Lorsque l'on a une base de données exploitée par une application (écrite dans un langage de programmation comme Java ou PHP, par exemple), c'est souvent dans cette application que sont construites les requêtes qui vont insérer, modifier, et sélectionner les données de la base.  
Si, pour une raison ou une autre (mauvaise conception de la base de données au départ, modèle de données qui s’étend fortement…), la structure des tables de la base change, il faut réécrire également l'application pour prendre en compte les modifications nécessaires pour les requêtes.

Cependant, si l'on a utilisé des vues, on peut éviter de réécrire toutes ces requêtes, ou du moins limiter le nombre de requêtes à réécrire. Si les requêtes sont faites sur des vues, il suffit en effet de modifier la définition de ces vues pour qu'elles fonctionnent avec la nouvelle structure.

**Exemple**

On a créé une vue V\_Client qui permet de voir le contenu de notre table Client (sauf la date de naissance, ajoutée après la définition de la vue).

Si, un beau jour, on décidait de stocker les adresses postales dans une table à part, il faudrait modifier la structure de notre base. Au lieu d'une seule table Client, on en aurait deux : Client(id, nom, prenom, date\_naissance, email, adresse\_id) et Adresse(id, rue\_numero, code\_postal, ville, pays).

Pour que l'application puisse continuer à sélectionner les personnes et leur adresse sans que l'on doive la modifier, il suffit de changer la requête définissant la vue :

CREATE OR REPLACE VIEW V\_Client

-- le OR REPLACE indispensable (ou on utilise ALTER VIEW)

AS SELECT Client.id, nom, prenom, rue\_numero AS adresse, code\_postal, ville, pays, email, date\_naissance

FROM Client

LEFT JOIN Adresse ON Client.adresse\_id = Adresse.id

-- LEFT JOIN au cas où certains clients n'auraient pas d'adresse définie

Le changement de structure de la base de données serait ainsi transparent pour l'application (du moins en ce qui concerne la sélection des clients) !

#### Restriction des données visibles par les utilisateurs

La gestion des utilisateurs et de leurs droits fera l'objet d'un prochain chapitre. Sachez simplement que, pour chaque utilisateur, il est possible de définir des droits particuliers pour chaque table et chaque vue (entre autres choses).  
On peut par exemple autoriser l'utilisateur A à faire des requêtes d'insertion, de modification, de suppression et de sélection (le fameux CRUD) sur la table T1, mais n'autoriser l'utilisateur B qu'à faire des sélections sur la table T1.  
Et imaginons qu'il existe aussi un utilisateur C auquel on veut donner l'autorisation de faire des requêtes de sélection sur la table T1, mais auquel il faudrait cacher certaines colonnes qui contiennent des données sensibles, ou certaines lignes auxquelles il ne devrait pas accéder. Il suffit de créer la vue V\_T1, n'ayant accès qu'aux colonnes/lignes "publiques" de la table T1, et de donner à C les droits sur la vue V\_T1, mais pas sur la table T1.

**Exemple**

Le stagiaire travaillant dans notre élevage s'occupe exclusivement des chats et ne doit pas avoir accès aux commentaires. On ne lui donne donc pas accès à la table Animal, mais à une vue V\_Animal\_stagiaire créée de la manière suivante :

CREATE VIEW V\_Animal\_stagiaire

AS SELECT id, nom, sexe, date\_naissance, espece\_id, race\_id, mere\_id, pere\_id, disponible

FROM Animal

WHERE espece\_id = 2;

### Algorithmes

Lors de la création d'une vue, on peut définir quel algorithme sera utilisé par MySQL lors d'une sélection sur celle-ci.

CREATE [OR REPLACE]

[ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]

VIEW nom\_vue

AS requete\_select

Il s'agit d'une **clause non standard**, donc valable uniquement pour MySQL. Deux algorithmes différents peuvent être utilisés : MERGE  et TEMPTABLE.

Les algorithmes interviennent au moment où l'on **sélectionne des données de la vue**, et pas directement à la création de celle-ci.

#### MERGE

Si l'algorithme MERGE  (fusion en anglais) a été choisi, lorsque l'on sélectionne des données de la vue, MySQL va **fusionner la requête SELECT  qui définit la vue avec les clauses de sélection**. Faire une sélection sur une vue qui utilise cet algorithme revient donc à faire une requête directement sur les tables sous-jacentes.

Comme ce n'est pas nécessairement très clair, voici deux exemples.

**Exemple 1**

On a créé la vue V\_Animal\_details à partir de la requête suivante :

SELECT Animal.id, Animal.sexe, Animal.date\_naissance, Animal.nom, Animal.commentaires,

Animal.espece\_id, Animal.race\_id, Animal.mere\_id, Animal.pere\_id, Animal.disponible,

Espece.nom\_courant AS espece\_nom, Race.nom AS race\_nom

FROM Animal

INNER JOIN Espece ON Animal.espece\_id = Espece.id

LEFT JOIN Race ON Animal.race\_id = Race.id;

Ensuite, on fait une sélection sur cette vue :

SELECT \*

FROM V\_Animal\_details

WHERE MONTH(date\_naissance) = 6;

Si c'est l'algorithme MERGE  qui est utilisé, MySQL va fusionner :

* la requête définissant la vue :
* SELECT Animal.id, Animal.sexe, Animal.date\_naissance, Animal.nom, Animal.commentaires,
* Animal.espece\_id, Animal.race\_id, Animal.mere\_id, Animal.pere\_id, Animal.disponible,
* Espece.nom\_courant AS espece\_nom, Race.nom AS race\_nom
* FROM Animal
* INNER JOIN Espece ON Animal.espece\_id = Espece.id
* LEFT JOIN Race ON Animal.race\_id = Race.id;
* les clauses de la requête de sélection :
* WHERE MONTH(date\_naissance) = 6;

Finalement, faire cette requête sur la vue provoquera l'exécution de la requête suivante :

SELECT Animal.id, Animal.sexe, Animal.date\_naissance, Animal.nom, Animal.commentaires,

Animal.espece\_id, Animal.race\_id, Animal.mere\_id, Animal.pere\_id, Animal.disponible,

Espece.nom\_courant AS espece\_nom, Race.nom AS race\_nom

FROM Animal

INNER JOIN Espece ON Animal.espece\_id = Espece.id

LEFT JOIN Race ON Animal.race\_id = Race.id

WHERE MONTH(date\_naissance) = 6;

**Exemple 2**

Si l'on exécute la requête suivante sur la vue V\_Chien, et que l'algorithme MERGE  est utilisé :

SELECT nom, date\_naissance

FROM V\_Chien

WHERE pere\_id IS NOT NULL;

MySQL va fusionner les deux requêtes, et exécuter ceci :

SELECT nom, date\_naissance

FROM Animal

WHERE espece\_id = 1

AND pere\_id IS NOT NULL;

#### TEMPTABLE

L'algorithme TEMPTABLE, par contre, **crée une table temporaire** contenant les résultats de la requête définissant la vue, puis, par la suite, exécute la requête de sélection sur cette table temporaire.

Donc, si l'on exécute la requête suivante sur la vue V\_Chien :

SELECT nom, date\_naissance

FROM V\_Chien

WHERE pere\_id IS NOT NULL;

Avec l'algorithme TEMPTABLE, la requête définissant la vue va être exécutée et ses résultats stockés dans une table temporaire.

SELECT id, sexe, date\_naissance, nom, commentaires, espece\_id, race\_id, mere\_id, pere\_id, disponible

FROM Animal

WHERE espece\_id = 1;

Ensuite, sur cette table temporaire, va être exécutée la requête finale :

SELECT nom, date\_naissance

FROM table\_temporaire

WHERE pere\_id IS NOT NULL;

#### Algorithme par défaut et conditions

Il existe une troisième option possible pour la clause ALGORITHM  dans la requête de création des vues : UNDEFINED.

Par défaut, si l'on ne précise pas d'algorithme pour la vue, c'est l'option UNDEFINED  qui est utilisée. Cette option laisse MySQL décider lui-même de l'algorithme qu'il appliquera.  
Si c'est possible, MERGE  sera utilisé, car cet algorithme est plus performant que TEMPTABLE. Cependant, toutes les vues ne peuvent pas utiliser l'algorithme MERGE. En effet, une vue utilisant un ou plusieurs des éléments suivants **ne pourra pas** utiliser MERGE  :

* DISTINCT  ;
* LIMIT  ;
* une fonction d'agrégation (SUM(), COUNT(), MAX(), etc.) ;
* GROUP BY  ;
* HAVING  ;
* UNION  ;
* une sous-requête dans la clause SELECT.

### Modification des données d'une vue

On a tendance à penser que les vues ne servent que pour la sélection de données. En réalité, il est possible de modifier, d'insérer et de supprimer des données par l'intermédiaire d'une vue.  
Les requêtes sont les mêmes que pour insérer, modifier et supprimer des données à partir d'une table (si ce n'est que l'on met le nom de la vue au lieu du nom de la table, bien sûr).

Cependant, pour qu'une vue ne soit pas en "lecture seule", elle doit répondre à une série de conditions.

#### Conditions pour qu'une vue permette de modifier des données (requêtes UPDATE)

##### Jointures

Il est possible de modifier des données à partir d'une vue définie avec une jointure, à condition de ne modifier qu'**une seule table**.

**Exemple**

-- Modifie Animal

UPDATE V\_Animal\_details

*SET* commentaires = 'Rhume chronique'

WHERE id = 21;

-- Modifie Race

UPDATE V\_Animal\_details

*SET* race\_nom = 'Maine Coon'

WHERE race\_nom = 'Maine coon';

-- Erreur

UPDATE V\_Animal\_details

*SET* commentaires = 'Vilain oiseau', espece\_nom = 'Perroquet pas beau'

-- commentaires vient de Animal, et espece\_nom vient de Espece

WHERE espece\_id = 4;

Les deux premières modifications ne posent aucun problème, mais la troisième échouera, car elle modifie des colonnes appartenant à deux tables différentes.

ERROR 1393 (HY000): Can not modify more than one base table through a join view 'elevage.v\_animal\_details'

##### Algorithme

La vue doit utiliser l'algorithme MERGE  (que l'algorithme ait été spécifié à la création, ou choisi par MySQL si l'algorithme n'a pas été défini). Par conséquent, les mêmes conditions que pour utiliser l'algorithme MERGE  s'appliquent. Les éléments suivants sont ainsi interdits dans une vue si l'on veut pouvoir modifier les données à partir de celle-ci :

* DISTINCT  ;
* LIMIT  ;
* une fonction d’agrégation (SUM(), COUNT(), MAX(), etc.) ;
* GROUP BY  ;
* HAVING  ;
* UNION  ;
* une sous-requête dans la clause SELECT.

**Exemple :** la vue V\_Nombre\_espece, qui utilise une fonction d'agrégation, ne permet pas de modifier les données.

UPDATE V\_Nombre\_espece

*SET* nb = 6

WHERE id = 4;

1288 (HY000): The target table V\_Nombre\_espece of the UPDATE is not updatable

##### Autres conditions

* On ne peut pas modifier les données à partir d'une vue qui est elle-même définie à partir d'une vue qui ne permet pas la modification des données.
* Ce n'est pas non plus possible à partir d'une vue dont la clause WHERE  contient une sous-requête faisant référence à une des tables de la clause FROM.

#### Conditions pour qu'une vue permette d'insérer des données (requêtes INSERT)

On peut insérer des données dans une vue si celle-ci respecte **les mêmes conditions que pour la modification de données**, ainsi que les conditions supplémentaires suivantes.

##### Valeurs par défaut

Toutes les colonnes n'ayant pas de valeur par défaut (et ne pouvant pas être NULL), de la table dans laquelle on veut faire l'insertion, doivent être référencées par la vue.

**Exemple**

INSERT INTO V\_Animal\_stagiaire (nom, sexe, date\_naissance, espece\_id, race\_id)

VALUES ('Rocco', 'M', '2012-03-12', 1, 9);

Ceci fonctionne. La colonne commentaires n'est pas référencée par V\_Animal\_stagiaire, mais peut être NULL  (qui est donc sa valeur par défaut).

CREATE VIEW V\_Animal\_mini

AS SELECT id, nom, sexe, espece\_id

FROM Animal;

INSERT INTO V\_Animal\_mini(nom, sexe, espece\_id)

VALUES ('Toxi', 'F', 1);

Par contre, l'insertion dans V\_Animal\_mini échoue, puisque date\_naissance n'est pas référencée, ne peut pas être NULLet n'a pas de valeur par défaut.

ERROR 1423 (HY000): Field of view 'elevage.v\_animal\_mini' underlying table doesn't have a default value

##### Jointures

Les vues avec jointure peuvent supporter l'insertion si :

* il n'y a que des jointures internes ;
* l'insertion se fait sur une seule table (comme pour la modification).

**Exemple**

INSERT INTO V\_Animal\_details (espece\_nom, espece\_nom\_latin)

VALUES ('Perruche terrestre', 'Pezoporus wallicus');

Il y a une jointure externe dans V\_Animal\_details, donc l'insertion ne fonctionnera pas.

ERROR 1471 (HY000): The target table V\_Animal\_details of the INSERT is not insertable-into

Par contre, si l'on crée une table V\_Animal\_espece avec uniquement une jointure interne, il n'y a aucun problème.

CREATE OR REPLACE VIEW V\_Animal\_espece

AS SELECT Animal.id, Animal.sexe, Animal.date\_naissance, Animal.nom, Animal.commentaires,

Animal.espece\_id, Animal.race\_id, Animal.mere\_id, Animal.pere\_id, Animal.disponible,

Espece.nom\_courant AS espece\_nom, Espece.nom\_latin AS espece\_nom\_latin

FROM Animal

INNER JOIN Espece ON Espece.id = Animal.espece\_id;

INSERT INTO V\_Animal\_espece (espece\_nom, espece\_nom\_latin)

VALUES ('Perruche terrestre', 'Pezoporus wallicus');

##### Expressions

Les colonnes de la vue doivent être de simples références à des colonnes, et non pas des expressions.

**Exemple**

Dans la vue V\_Espece\_dollars, la colonne prix\_dollars correspond à ROUND(prix\*1.30813, 2). Il n'est donc pas possible d'insérer des données à partir de cette vue.

INSERT INTO V\_Espece\_dollars (nom\_courant, nom\_latin, prix\_dollars)

VALUES ('Perruche terrestre', 'Pezoporus wallicus', 30);

ERROR 1471 (HY000): The target table V\_Espece\_dollars of the INSERT is not insertable-into

##### Colonnes dupliquées

La même colonne ne peut pas être référencée deux fois dans la vue.

**Exemple**

Si l'on crée une vue avec deux fois la même colonne référencée, il est possible de modifier des données à partir de celle-ci, mais pas d'en insérer.

CREATE VIEW V\_Espece\_2noms -- on référence nom\_courant deux fois

AS SELECT id, nom\_courant, nom\_latin, description, prix, nom\_courant AS nom2

FROM Espece;

-- Modification, pas de problème

UPDATE V\_Espece\_2noms

*SET* description= 'Joli oiseau aux plumes majoritairement vert brillant', prix = 20.00

WHERE nom\_courant = 'Perruche terrestre';

-- Insertion, impossible

INSERT INTO V\_Espece\_2noms (nom\_courant, nom\_latin, prix)

VALUES ('Perruche turquoisine', 'Neophema pulchella', 40);

ERROR 1471 (HY000): The target table V\_Espece\_2noms of the INSERT is not insertable-into

#### Conditions pour qu'une vue permette de supprimer des données (requêtes DELETE)

Il est possible de supprimer des données à partir d'une vue si et seulement si :

* il est possible de modifier des données à partir de cette vue ;
* cette vue est "mono-table" (une seule table sous-jacente).

#### Option de la vue pour la modification des données

Lors de la création d'une vue, on peut ajouter une option : WITH [LOCAL | CASCADED] CHECK OPTION.

CREATE [OR REPLACE]

[ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]

VIEW nom\_vue [(liste\_colonnes)]

AS requete\_select

[WITH [CASCADED | LOCAL] CHECK OPTION]

Lorsque cette option est spécifiée, les modifications et insertions ne sont acceptées que si les données répondent aux conditions de la vue (c'est-à-dire aux conditions données par la clause WHERE  de la requête définissant la vue).

**Exemples**

Si la vue V\_Animal\_stagiaire (qui, pour rappel, sélectionne les chats uniquement) est définie avec WITH CHECK OPTION, on ne peut pas modifier l'espece\_id à partir de cette vue.

CREATE OR REPLACE VIEW V\_Animal\_stagiaire

AS SELECT id, nom, sexe, date\_naissance, espece\_id, race\_id, mere\_id, pere\_id, disponible

FROM Animal

WHERE espece\_id = 2

WITH CHECK OPTION;

En effet, cette vue est définie avec la condition WHERE espece\_id = 2. Les modifications faites sur les données de cette vue doivent respecter cette condition.

UPDATE V\_Animal\_stagiaire

*SET* espece\_id = 1

WHERE nom = 'Cracotte';

ERROR 1369 (HY000): CHECK OPTION failed 'elevage.v\_animal\_stagiaire'

De même, l'insertion d'un animal dont l'espece\_id n'est pas 2 sera refusée aussi.

INSERT INTO V\_Animal\_stagiaire (sexe, date\_naissance, espece\_id, nom)

VALUES ('F', '2011-09-21 15:14:00', 2, 'Bambi');

-- c'est un chat, pas de problème

INSERT INTO V\_Animal\_stagiaire (sexe, date\_naissance, espece\_id, nom)

VALUES ('M', '2011-03-11 05:54:00', 6, 'Tiroli');

-- c'est une perruche, impossible

La première insertion ne pose aucun problème, mais la seconde échoue.

##### LOCAL ou CASCADED

* LOCAL  : seules les conditions de la vue même sont vérifiées.
* CASCADED  : les conditions des vues sous-jacentes éventuelles sont également vérifiées. C'est l'option par défaut.

La vue V\_Chien\_race, par exemple, est définie à partir de la vue V\_Chien. Si l'on ajoute WITH LOCAL CHECK OPTION  à V\_Chien\_race, les modifications et insertions dans cette vue devront vérifier uniquement les conditions de la vue elle-même, c'est-à-dire race\_id IS NOT NULL. Si, par contre, c'est WITH CASCADED CHECK OPTION  qui est choisi, alors les modifications et insertions devront toujours vérifier les conditions de V\_Chien\_race, mais également celles de V\_Chien (espece\_id = 1).

**Exemple 1 :** LOCAL

CREATE OR REPLACE VIEW V\_Chien\_race

AS SELECT id, sexe, date\_naissance, nom, commentaires, espece\_id, race\_id, mere\_id, pere\_id, disponible

FROM V\_Chien

WHERE race\_id IS NOT NULL

WITH LOCAL CHECK OPTION;

-- Modification --

-- ------------ --

UPDATE V\_Chien\_race

*SET* race\_id = NULL -- Ne respecte pas la condition de V\_Chien\_race

WHERE nom = 'Zambo';-- => Impossible

UPDATE V\_Chien\_race

*SET* espece\_id = 2, race\_id = 4-- Ne respecte pas la condition de V\_Chien

WHERE nom = 'Java'; -- => possible puisque LOCAL CHECK OPTION

-- Insertion --

-- --------- --

INSERT INTO V\_Chien\_race (sexe, date\_naissance, nom, commentaires, espece\_id, race\_id)

VALUES ('M', '2012-02-28 03:05:00', 'Pumba', 'Prématuré, à surveiller', 1, 9);

-- Respecte toutes les conditions => Pas de problème

INSERT INTO V\_Chien\_race (sexe, date\_naissance, nom, commentaires, espece\_id, race\_id)

VALUES ('M', '2011-05-24 23:51:00', 'Lion', NULL, 2, 5);

-- La race n'est pas NULL, mais c'est un chat => pas de problème puisque LOCAL

INSERT INTO V\_Chien\_race (sexe, date\_naissance, nom, commentaires, espece\_id, race\_id)

VALUES ('F', '2010-04-28 13:01:00', 'Mouchou', NULL, 1, NULL);

-- La colonne race\_id est NULL => impossible

La première modification et la dernière insertion échouent donc, avec l'erreur suivante :

ERROR 1369 (HY000): CHECK OPTION failed 'elevage.v\_chien\_race'

**Exemple 2 :** CASCADED

CREATE OR REPLACE VIEW V\_Chien\_race

AS SELECT id, sexe, date\_naissance, nom, commentaires, espece\_id, race\_id, mere\_id, pere\_id, disponible

FROM V\_Chien

WHERE race\_id IS NOT NULL

WITH CASCADED CHECK OPTION;

UPDATE V\_Chien\_race

*SET* race\_id = NULL -- Ne respecte pas la condition de V\_Chien\_race

WHERE nom = 'Zambo'; -- => impossible

UPDATE V\_Chien\_race

*SET* espece\_id = 2, race\_id = 4 -- Ne respecte pas la condition de V\_Chien

WHERE nom = 'Fila'; -- => impossible aussi puisque CASCADED

Cette fois, les deux modifications échouent.

## Tables temporaires

Ce chapitre est consacré aux tables temporaires qui, comme leur nom l'indique, sont des tables ayant une durée de vie très limitée. En effet, tout comme les variables utilisateur ou les requêtes préparées, les tables temporaires **n'existent que dans la session qui les a créées**. Dès que la session se termine (déconnexion volontaire ou accidentelle), les tables temporaires sont supprimées.

Nous verrons en détail dans ce chapitre comment se comportent ces tables temporaires et à quoi elles peuvent servir.  
De plus, nous verrons deux nouvelles manières de créer une table (temporaire ou non) en partant de la structure et/ou des données d'une ou plusieurs autres tables.

### Principe, règles et comportement

On l'a dit, une table temporaire est une table qui n'existe que dans la session qui l'a créée. En dehors de cela, c'est une table presque normale. On peut exécuter sur ces tables toutes les opérations que l'on exécute sur une table classique : insérer des données, les modifier, les supprimer, et bien sûr les sélectionner.

#### Création, modification, suppression d'une table temporaire

Pour créer, modifier et supprimer une table temporaire, on utilise les mêmes commandes que pour les tables normales.

##### Création

Pour créer une table temporaire, on peut utiliser tout simplement CREATE TABLE  en ajoutant TEMPORARY, pour préciser qu'il s'agit d'une table temporaire.

**Exemple :** création d'une table temporaire TMP\_Animal.

CREATE TEMPORARY TABLE TMP\_Animal (

id *INT* UNSIGNED PRIMARY KEY,

nom *VARCHAR*(30),

espece\_id *INT* UNSIGNED,

sexe *CHAR*(1)

);

DESCRIBE TMP\_Animal;

Visiblement, la table a bien été créée. Par contre, si l'on tente de vérifier cela en utilisant SHOW TABLES  plutôt que DESCRIBE TMP\_Animal, on ne trouvera pas la nouvelle table temporaire dans la liste des tables. Et pour cause, seules les tables permanentes (et les vues) sont listées par cette commande.

##### Modification

Pour la modification, nul besoin du mot-clé TEMPORARY, on utilise directement ALTER TABLE, comme s'il s'agissait d'une table normale.

**Exemple :** ajout d'une colonne à TMP\_Animal.

ALTER TABLE TMP\_Animal

ADD COLUMN date\_naissance DATETIME;

##### Suppression

En ce qui concerne la suppression, on a le choix d'ajouter TEMPORARY  ou non.  
Si TEMPORARY  est précisé, la table mentionnée ne sera supprimée que s'il s'agit bien d'une table temporaire. Sans ce mot-clé, on pourrait supprimer par erreur une table non temporaire, en cas de confusion des noms des tables, par exemple.

**Exemple :** suppression de TMP\_Animal, qui n'aura pas fait long feu.

DROP TEMPORARY TABLE TMP\_Animal;

#### Utilisation des tables temporaires

Une table temporaire s'utilise comme une table normale. Les commandes d'insertion, modification et suppression de données sont exactement les mêmes.

Notez que l'immense majorité des tables temporaires étant créées pour stocker des données venant d'autres tables, on utilise souvent INSERT INTO ... SELECT  pour l'insertion des données.

#### Cache-cache table

Une table temporaire existe donc uniquement pour la session qui la crée. Par conséquent, deux sessions différentes peuvent parfaitement utiliser chacune une table temporaire ayant le même nom, mais ni la même structure ni les mêmes données.

Il est également possible de créer une table temporaire ayant le même nom qu'une table normale. Toutes les autres sessions éventuelles continueront à travailler sur la table normale comme si de rien n'était.

Et la session qui crée cette fameuse table ?

En cas de conflit de noms entre une table temporaire et une table permanente, la table temporaire va masquer la table permanente.  
Donc tant que cette table temporaire existera (c'est-à-dire jusqu'à ce qu'on la supprime explicitement, ou jusqu'à déconnexion de la session), toutes les requêtes faites en utilisant son nom seront exécutées sur la table temporaire, et non sur la table permanente de même nom.

**Exemple :** création d'une table temporaire nommée Animal.

On commence par sélectionner les perroquets de la table Animal (la table normale, puisque l'on n'a pas encore créé la table temporaire). On obtient quatre résultats.

SELECT id, sexe, nom, commentaires, espece\_id

FROM Animal

WHERE espece\_id = 4;

On crée ensuite la table temporaire, et on y insère un animal avec *espece\_id = 4*. On refait la même requête de sélection, et on obtient uniquement l'animal que l'on vient d'insérer, avec la structure de la table temporaire.

CREATE TEMPORARY TABLE Animal (

id *INT* UNSIGNED PRIMARY KEY,

nom *VARCHAR*(30),

espece\_id *INT* UNSIGNED,

sexe *CHAR*(1)

);

INSERT INTO Animal

VALUES (1, 'Tempo', 4, 'M');

SELECT \*

FROM Animal

WHERE espece\_id = 4;

Pour terminer, suppression de la table temporaire. La même requête de sélection nous affiche à nouveau les quatre perroquets de départ.

DROP TEMPORARY TABLE Animal;

SELECT id, sexe, nom, commentaires, espece\_id

FROM Animal

WHERE espece\_id = 4;

Dans ce cas de figure, il vaut mieux utiliser le mot-clé TEMPORARY  dans la requête de suppression de la table, afin d'être sûr que l'on détruit bien la table temporaire et non la table permanente.

Gardez bien en tête le fait que les tables temporaires sont détruites dès que la connexion prend fin. Dans de nombreuses applications, une nouvelle connexion est créée à chaque nouvelle action. Par exemple, pour un site internet codé en PHP, on crée une nouvelle connexion (donc une nouvelle session) à chaque rechargement de page.

**Restrictions des tables temporaires**

Deux restrictions importantes sont à noter lorsque l'on travaille avec des tables temporaires :

* on ne peut pas mettre de clé étrangère sur une table temporaire ;
* on ne peut pas faire référence à une table temporaire deux fois dans la même requête.

Quelques explications sur la deuxième restriction : lorsque l'on parle de "faire référence deux fois à la table", il s'agit d'aller **chercher des données** deux fois dans la table. Le nom de la table temporaire peut apparaître plusieurs fois dans la requête, par exemple pour préfixer le nom des colonnes.

**Exemples :** on recrée une table *TMP\_Animal,* un peu plus complexe cette fois, que l'on remplit avec les chats de la table *Animal*.

CREATE TEMPORARY TABLE TMP\_Animal (

id *INT* UNSIGNED PRIMARY KEY,

nom *VARCHAR*(30),

espece\_id *INT* UNSIGNED,

sexe *CHAR*(1),

mere\_id *INT* UNSIGNED,

pere\_id *INT* UNSIGNED

);

INSERT INTO TMP\_Animal

SELECT id, nom, espece\_id, sexe, mere\_id, pere\_id

FROM Animal

WHERE espece\_id = 2;

**1.** Référence n'est pas occurrence

SELECT TMP\_Animal.nom, TMP\_Animal.sexe

FROM TMP\_Animal

WHERE nom LIKE 'B%';

Ici, TMP\_Animal apparaît trois fois dans la requête. Cela ne pose aucun problème, puisque la table n'est utilisée qu'une fois pour en extraire des données : dans le FROM.

**2.** Auto-jointure

On essaye d'afficher le nom des chats, ainsi que le nom de leur père quand on le connaît.

SELECT TMP\_Animal.nom, TMP\_Pere.nom AS pere

FROM TMP\_Animal

INNER JOIN TMP\_Animal AS TMP\_Pere

ON TMP\_Animal.pere\_id = TMP\_Pere.id;

ERROR 1137 (HY000): Can't reopen table: 'TMP\_Animal'

Cette fois, cela ne fonctionne pas : on extrait en effet des données de TMP\_Animal, et de TMP\_Animal  AS  TMP\_Pere.

**3.** Sous-requête

On affiche le nom des animaux référencés comme pères.

SELECT nom

FROM TMP\_Animal

WHERE id IN (SELECT pere\_id FROM TMP\_Animal);

ERROR 1137 (HY000): Can't reopen table: 'TMP\_Animal'

À nouveau, on extrait des données de TMP\_Animal à deux endroits différents : dans la requête et dans la sous-requête. Cela ne peut pas fonctionner.

#### Interaction avec les transactions

Un comportement assez intéressant lié aux tables temporaires concerne les transactions.  
On a vu que les commandes de création d'objets en tout genre provoquaient la validation implicite de la transaction dans laquelle la commande était faite. Ainsi, CREATE TABLE  provoque une validation implicite.

Mais ce n'est pas le cas pour les tables temporaires : CREATE TEMPORARY TABLE  et DROP TEMPORARY TABLE  ne valident pas les transactions.

Cependant, ces deux commandes ne peuvent pas être annulées par un ROLLBACK.

**Exemple**

START TRANSACTION;

INSERT INTO Espece (nom\_courant, nom\_latin)

VALUES ('Gerbille de Mongolie', 'Meriones unguiculatus');

CREATE TEMPORARY TABLE TMP\_Test (id *INT*);

ROLLBACK;

SELECT id, nom\_courant, nom\_latin, prix FROM Espece;

SELECT \* FROM TMP\_Test;

### Méthodes alternatives de création des tables

Les deux méthodes de création de tables présentées ici sont valables pour créer **des tables normales ou des tables temporaires**. Elles se basent toutes les deux sur des tables et/ou des données préexistantes.  
Le choix de présenter ces méthodes dans le cadre des tables temporaires s'explique par le fait qu'en général, celles-ci sont créées pour stocker des données provenant d'autres tables, et par conséquent, sont souvent créées en utilisant une de ces deux méthodes.

#### Créer une table à partir de la structure d'une autre

Il est possible de créer la copie exacte d'une table en utilisant la commande suivante :

CREATE [TEMPORARY] TABLE nouvelle\_table

LIKE ancienne\_table;

Les types de données, les index, les contraintes, les valeurs par défaut, le moteur de stockage, toutes les caractéristiques de ancienne\_table seront reproduites dans nouvelle\_table. Par contre, les données ne seront pas copiées : nouvelle\_table sera vide.

**Exemple :** reproduction de la table Espece.

CREATE TABLE Espece\_copy

LIKE Espece;

DESCRIBE Espece;

DESCRIBE Espece\_copy;

**Exemple**

INSERT INTO Espece\_copy

SELECT \* FROM Espece

WHERE prix < 100;

SELECT id, nom\_courant, prix

FROM Espece\_copy;

**Tables temporaires**

Si l'on crée une table temporaire avec cette commande, tous les attributs de la table d'origine seront conservés, sauf les clés étrangères, puisque l'on ne peut avoir de clé étrangère dans une table temporaire.

**Exemple :** copie de la table *Animal*

CREATE TEMPORARY TABLE Animal\_copy

LIKE Animal;

INSERT INTO Animal (nom, sexe, date\_naissance, espece\_id)

VALUES ('Mutant', 'M', NOW(), 12);

INSERT INTO Animal\_copy (nom, sexe, date\_naissance, espece\_id)

VALUES ('Mutant', 'M', NOW(), 12);

Aucune espèce n'a 12 pour *id*, l'insertion dans *Animal* échoue donc à cause de la clé étrangère. Par contre, dans la table temporaire *Animal\_copy*, l'insertion réussit.

Si l'on crée une table sur la base d'une table temporaire, la nouvelle table n'est pas temporaire par défaut. Pour qu'elle le soit, il faut obligatoirement le préciser à la création avec CREATE TEMPORARY TABLE.

**Créer une table à partir de données sélectionnées**

Cette seconde méthode ne se base pas sur la structure d'une table, mais sur les données récupérées par une requête SELECT  pour construire une table et y insérer des données. On fait donc ici d'une pierre deux coups : création de la table et insertion de données dans celle-ci.

CREATE [TEMPORARY] TABLE nouvelle\_table

SELECT ...

Le type des colonnes de la nouvelle table sera déduit des colonnes sélectionnées. Par contre, la plupart des caractéristiques des colonnes sélectionnées seront perdues :

* les index ;
* les clés (primaires ou étrangères) ;
* l'auto-incrémentation.

Les valeurs par défaut et les contraintes NOT NULL  seront par contre conservées.

**Exemple :** suppression puis recréation d'une table temporaire *Animal\_copy* contenant tous les rats de la table *Animal*.

DROP TABLE Animal\_copy;

CREATE TEMPORARY TABLE Animal\_copy

SELECT \*

FROM Animal

WHERE espece\_id = 5;

DESCRIBE Animal;

DESCRIBE Animal\_copy;

##### Forcer le type des colonnes

On peut laisser MySQL déduire le type des colonnes du SELECT, mais il est également possible de préciser le type que l'on désire en faisant attention à la compatibilité entre les types que l'on précise et les colonnes sélectionnées.  
On peut également préciser les index désirés, les clés et l'éventuelle colonne à auto-incrémenter.

La syntaxe est alors similaire à un CREATE [TEMPORARY] TABLE  classique, si ce n'est que l'on rajoute une requête de sélection à la suite.

**Exemple :** recréation d'Animal\_copy, en modifiant quelques types et attributs.

DROP TABLE Animal\_copy;

CREATE TEMPORARY TABLE Animal\_copy (

id *INT* UNSIGNED PRIMARY KEY AUTO\_INCREMENT,

sexe *CHAR*(1),

date\_naissance DATETIME,

nom *VARCHAR*(100),

commentaires *TEXT*,

espece\_id *INT* NOT NULL,

race\_id *INT*,

mere\_id *INT*,

pere\_id *INT*,

disponible BOOLEAN DEFAULT TRUE,

INDEX (nom(10))

) ENGINE=InnoDB

SELECT \*

FROM Animal

WHERE espece\_id = 5;

DESCRIBE Animal\_copy;

Les colonnes id, espece\_id, race\_id, mere\_id et pere\_id ne sont plus des SMALLINT, mais des INT, et seul id est UNSIGNED. Par contre, on a bien un index sur nom, et une clé primaire auto-incrémentée avec id. On n'a pas précisé de contrainte NOT NULL  sur date\_naissance, donc il n'y en a pas, bien que cette contrainte existe dans la table d'origine.

##### Nom des colonnes

Les noms des colonnes de la table créée correspondront aux noms des colonnes dans la requête SELECT. Il est bien sûr possible d'utiliser les alias dans celle-ci pour renommer une colonne.  
Cela implique que, lorsque vous forcez le type des colonnes comme on vient de le voir, il est impératif que vos noms de colonnes correspondent à ceux de la requête SELECT, car l'insertion des données dans leurs colonnes respectives se fera sur la base du nom, et pas sur la base de la position.

**Exemple :** on recrée Animal\_copy, mais les noms des colonnes sont dans le désordre, et certains ne correspondent pas à la requête SELECT.

DROP TABLE Animal\_copy;

CREATE TEMPORARY TABLE Animal\_copy (

id *INT* UNSIGNED PRIMARY KEY AUTO\_INCREMENT,

nom *VARCHAR*(100), -- Ordre différent de la requête SELECT

sexe *CHAR*(1),

espece\_id *INT* NOT NULL, -- Ordre différent de la requête SELECT

date\_naissance DATETIME,

commentaires *TEXT*,

race\_id *INT*,

maman\_id *INT*, -- Nom de colonne différent de la requête SELECT

papa\_id *INT*, -- Nom de colonne différent de la requête SELECT

disponible BOOLEAN DEFAULT TRUE,

INDEX (nom(10))

) ENGINE=InnoDB

SELECT id, sexe, date\_naissance, nom, commentaires, espece\_id, race\_id, mere\_id, pere\_id, disponible

FROM Animal

WHERE espece\_id = 5;

DESCRIBE Animal\_copy;

Les colonnes *mere\_id* et *pere\_id* de la requête SELECT  ne correspondaient à aucune colonne définie par la commande CREATE TABLE. Ces colonnes ont donc été créées dans la nouvelle table, mais leur type a été déduit à partir du SELECT. Quant aux colonnes *maman\_id* et *papa\_id*, elles ont bien été créées bien qu'elles ne correspondent à rien dans la sélection. Elles ne contiendront simplement aucune donnée.

Voyons maintenant le contenu d'*Animal\_copy*.

SELECT maman\_id, papa\_id, id, sexe, nom, espece\_id, mere\_id, pere\_id

FROM Animal\_copy;

Les données sont dans les bonnes colonnes, et les colonnes *maman\_id* et *papa\_id* sont vides, contrairement à *mere\_id* et *pere\_id*.

Une conséquence intéressante de ce comportement est qu'il est tout à fait possible de préciser le type et les attributs pour une partie des colonnes uniquement.

**Exemple :** Recréation d'*Animal\_copy*. Cette fois, on ne précise le type et les attributs que de la colonne *id* et on ajoute un index sur *nom*.

DROP TABLE Animal\_copy;

CREATE TEMPORARY TABLE Animal\_copy (

id *INT* UNSIGNED PRIMARY KEY AUTO\_INCREMENT,

INDEX (nom(10))

) ENGINE=InnoDB

SELECT \*

FROM Animal

WHERE espece\_id = 5;

DESCRIBE Animal\_copy;

##### Réunir les données de plusieurs tables

Puisque cette méthode de création de tables se base sur une requête de sélection, cela signifie que l'on peut très bien créer une table sur la base d'une partie des colonnes d'une table existante.  
Cela signifie également que l'on peut créer une table à partir de données provenant de plusieurs tables différentes.  
Bien entendu, une table ne pouvant avoir deux colonnes de même nom, il ne faut pas oublier d'utiliser les alias pour renommer certaines colonnes en cas de noms dupliqués.

**Exemple 1 :** création de Client\_mini, qui correspond à une partie de la table Client.

CREATE TABLE Client\_mini

SELECT nom, prenom, date\_naissance

FROM Client;

**Exemple 2 :** création de Race\_espece, à partir des tables Espece et Race.

CREATE TABLE Race\_espece

SELECT Race.id, Race.nom, Espece.nom\_courant AS espece, Espece.id AS espece\_id

FROM Race

INNER JOIN Espece ON Espece.id = Race.espece\_id;

### Utilité des tables temporaires

Les tables temporaires n'existant que le temps d'une session, leur usage est limité à des situations précises.

#### Gain de performance

Si, **dans une même session**, vous devez effectuer des calculs et/ou des requêtes plusieurs fois sur le même set de données, il peut être intéressant de stocker ce set de données dans une table temporaire, pour travailler sur cette table.  
En effet, une fois vos données de travail isolées dans une table temporaire, les requêtes vous servant à sélectionner les données qui vous intéressent seront simplifiées, donc plus rapides.

Typiquement, imaginons que l'on désire faire une série de statistiques sur les adoptions de chiens.

* Quelles races sont les plus demandées ?
* Y a-t-il une période de l'année pendant laquelle les gens adoptent beaucoup ?
* Nos clients ayant adopté un chien sont-ils souvent dans la même région ?
* Les chiens adoptés sont-ils plutôt des mâles ou des femelles ?
* …

On va avoir besoin de données provenant des tables Animal, Race, Client et Adoption, en faisant au minimum une jointure à chaque requête pour n'avoir que les données liées aux chiens.  
Il peut dès lors être intéressant de stocker toutes les données dont on aura besoin dans une table temporaire.

CREATE TEMPORARY TABLE TMP\_Adoption\_chien

SELECT Animal.id AS animal\_id, Animal.nom AS animal\_nom, Animal.date\_naissance AS animal\_naissance, Animal.sexe AS animal\_sexe, Animal.commentaires AS animal\_commentaires,

Race.id AS race\_id, Race.nom AS race\_nom,

Client.id AS client\_id, Client.nom AS client\_nom, Client.prenom AS client\_prenom, Client.adresse AS client\_adresse,

Client.code\_postal AS client\_code\_postal, Client.ville AS client\_ville, Client.pays AS client\_pays, Client.date\_naissance AS client\_naissance,

Adoption.date\_reservation AS adoption\_reservation, Adoption.date\_adoption AS adoption\_adoption, Adoption.prix

FROM Animal

LEFT JOIN Race ON Animal.race\_id = Race.id

INNER JOIN Adoption ON Animal.id = Adoption.animal\_id

INNER JOIN Client ON Client.id = Adoption.client\_id

WHERE Animal.espece\_id = 1;

Toutes les requêtes pour obtenir les statistiques et informations demandées pourront donc se faire directement sur TMP\_Adoption\_chien, ce qui sera plus rapide que de refaire chaque fois une requête avec jointure(s).  
Il peut bien sûr être intéressant d'ajouter des index sur les colonnes qui serviront souvent dans les requêtes.

#### Tests

Il arrive que l'on veuille tester **l'effet des instructions avant de les appliquer sur les données**.  
Pour cela, on peut par exemple utiliser les transactions et annuler ou valider les requêtes selon que l'effet produit correspond à ce qui était attendu ou non.  
Une autre solution est d'utiliser les tables temporaires. On crée des **tables temporaires contenant les données sur lesquelles on veut faire des tests**, on effectue les tests sur celles-ci, et une fois que l'on est satisfait du résultat, on applique les mêmes requêtes aux vraies tables. Cette solution permet de ne pas risquer de valider inopinément des requêtes de test. Sans oublier qu'il n'est pas toujours possible d'utiliser les transactions : si les tables sont de type MyISAM, les transactions ne sont pas disponibles. De même, les transactions sont inutilisables si les tests comprennent des requêtes provoquant une validation implicite des transactions.

**Petite astuce :** pour tester une série d'instructions à partir d'un fichier (en utilisant la commande SOURCE), il est souvent plus simple de créer des tables temporaires **ayant le même nom** que les tables normales, qui seront ainsi masquées. Il ne sera alors pas nécessaire de modifier le nom des tables dans les requêtes à exécuter par le fichier selon que l'on est en phase de test ou non.

#### Sets de résultats et procédures stockées

Lorsque l'on crée une procédure stockée, on peut utiliser les paramètres OUT  et INOUT  pour récupérer les résultats de la procédure. Mais ces paramètres ne peuvent contenir qu'un seul élément.

Que faire si l'on veut récupérer plusieurs lignes de résultats à partir d'une procédure stockée ?

On peut avoir besoin de récupérer le résultat d'une requête SELECT, ou une liste de valeurs calculées dans la procédure. Ou encore, on peut vouloir passer un set de résultats d'une procédure stockée à une autre.

Une solution est d'utiliser une table temporaire pour stocker les résultats que l'on veut utiliser en dehors de la procédure.

**Exemple :** création par une procédure d'une table temporaire stockant les adoptions qui ne sont pas en ordre de paiement.

DELIMITER |

CREATE PROCEDURE table\_adoption\_non\_payee()

BEGIN

DROP TEMPORARY TABLE IF EXISTS Adoption\_non\_payee;

CREATE TEMPORARY TABLE Adoption\_non\_payee

SELECT Client.id AS client\_id, Client.nom AS client\_nom, Client.prenom AS client\_prenom, Client.email AS client\_email,

Animal.nom AS animal\_nom, Espece.nom\_courant AS espece, Race.nom AS race,

Adoption.date\_reservation, Adoption.date\_adoption, Adoption.prix

FROM Adoption

INNER JOIN Client ON Client.id = Adoption.client\_id

INNER JOIN Animal ON Animal.id = Adoption.animal\_id

INNER JOIN Espece ON Espece.id = Animal.espece\_id

LEFT JOIN Race ON Race.id = Animal.race\_id

WHERE Adoption.paye = FALSE;

END |

DELIMITER ;

CALL table\_adoption\_non\_payee();

SELECT client\_id, client\_nom, client\_prenom, animal\_nom, prix

FROM Adoption\_non\_payee;

## Vues matérialisées

Pour le dernier chapitre de cette partie, nous allons parler des vues matérialisées. Comme leur nom l'indique, les vues matérialisées sont **des vues dont les données sont matérialisées**, c'est-à-dire stockées.

Les vues matérialisées sont des objets assez utiles, permettant un **gain de performance** relativement important lorsqu'ils sont bien utilisés. Malheureusement, MySQL n'implémente pas les vues matérialisées en tant que telles.  
Cependant, en utilisant les moyens du bord (c'est-à-dire les objets et concepts disponibles avec MySQL), il est tout à fait possible de construire des vues matérialisées soi-même.

Voyons donc quel est le principe de ces vues matérialisées, ce que l'on peut gagner à les utiliser, et comment les construire et les mettre à jour.

### Principe

#### Vues - rappels et performance

Une vue, c'est tout simplement **une requête SELECT  à laquelle on donne un nom**. Lorsque l'on sélectionne des données à partir d'une vue, on exécute en réalité la requête SQL de la vue. Par conséquent, les vues ne permettent pas de gagner en performance.

En fait, dans certains cas, les requêtes sur des vues peuvent même être moins rapides que si l'on fait directement la requête sur la ou les tables.

Les vues peuvent en effet utiliser deux algorithmes différents :

* MERGE  : les clauses de la requête sur la vue (WHERE, ORDER BY…) sont fusionnées à la requête définissant la vue ;
* TEMPTABLE  : une table temporaire est créée avec les résultats de la requête définissant la vue, et la requête de sélection sur la vue est ensuite exécutée sur cette table temporaire.

Avec l'algorithme MERGE, tout se passe comme si l'on exécutait la requête directement sur les tables contenant les données. On perd un petit peu de temps à fusionner les clauses des deux requêtes, mais c'est négligeable.

Par contre, avec TEMPTABLE, non seulement on exécute deux requêtes (une pour créer la vue temporaire, l'autre sur cette dernière), mais en plus, **la table temporaire ne possède aucun index**, contrairement aux tables normales. Une recherche dans la table temporaire peut donc prendre plus de temps que la même recherche sur une table normale, pour peu que cette dernière possède des index.

Enfin, n'oublions pas que MySQL nécessite l'utilisation de l'algorithme TEMPTABLE  pour les vues contenant au moins un des éléments suivants :

* DISTINCT  ;
* LIMIT  ;
* une fonction d'agrégation (SUM(), COUNT(), MAX(), etc.) ;
* GROUP BY  ;
* HAVING  ;
* UNION  ;
* une sous-requête dans la clause SELECT.

#### Vues matérialisées

Il fallait donc quelque chose qui combine les avantages des vues et les avantages des tables normales : c'est ainsi que le concept des vues matérialisées est né !

Une vue matérialisée est **un objet qui permet de stocker le résultat d'une requête SELECT**. Là où une vue se contente de stocker la requête, la vue matérialisée va stocker directement les résultats (elle va donc les matérialiser), plutôt que la requête. Lorsque l'on fait une requête sur une vue matérialisée, on va donc chercher directement des données dans celle-ci, sans passer par les tables d'origine et/ou une table temporaire intermédiaire.

##### "Vraies" vues matérialisées

Certains SGBD (Oracle, par exemple) permettent de créer directement des vues matérialisées, avec des outils et des options dédiés. Il suffit alors de créer les vues matérialisées que l'on désire, de les paramétrer correctement, et tout se fait automatiquement.

Malheureusement, MySQL ne propose pas encore ce type d'objet. Il faut donc se débrouiller avec les outils disponibles.

##### "Fausses" vues matérialisées pour MySQL

On veut donc une structure qui permet de stocker des données, provenant d'une requête SELECT.

Qu'a-t-on comme structure qui permet de stocker les données ?

La réponse semble évidente : **une table !**

Pour créer une vue matérialisée avec MySQL, on utilise donc tout simplement une table dans laquelle on stocke les résultats d'une requête SELECT.

**Exemple :** on veut matérialiser V\_Revenus\_annee\_espece.

CREATE TABLE VM\_Revenus\_annee\_espece

ENGINE = InnoDB

SELECT YEAR(date\_reservation) AS annee, Espece.id AS espece\_id, SUM(Adoption.prix) AS somme, COUNT(Adoption.animal\_id) AS nb

FROM Adoption

INNER JOIN Animal ON Animal.id = Adoption.animal\_id

INNER JOIN Espece ON Animal.espece\_id = Espece.id

GROUP BY annee, Espece.id;

Par facilité, on a utilisé ici la commande CREATE TABLE ... SELECT. Mais il est tout à fait possible de créer la table avec la commande habituelle et de la remplir par la suite, avec un INSERT INTO... SELECT, par exemple. Il s'agit d'une table tout à fait normale. Ce n'est que l'usage auquel elle est destinée qui en fait une vue matérialisée.

On a maintenant matérialisé la vue V\_Revenus\_annee\_espece, avec VM\_Revenus\_annee\_espece.

Et c'est tout ?

Non, bien sûr. C'est bien beau d'avoir créé une table, mais il va falloir aussi tenir cette vue matérialisée à jour.

### Mise à jour des vues matérialisées

On l'a dit, il ne suffit pas de créer une vue matérialisée à partir des données, il faut aussi la tenir à jour lorsque les données des tables d'origine changent.

Deux possibilités :

* une mise à jour sur demande ;
* une mise à jour automatique chaque fois qu'un changement est fait.

#### Mise à jour sur demande

Si l'on veut pouvoir mettre la vue matérialisée à jour ponctuellement, on peut utiliser une procédure stockée. Le plus simple sera de supprimer les données de la vue matérialisée, puis d'insérer les données à jour grâce à la même requête de sélection ayant servi à créer/initialiser la vue matérialisée.

**Exemple :** procédure stockée permettant de mettre à jour la vue VM\_Revenus\_annee\_especes.

DELIMITER |

CREATE PROCEDURE maj\_vm\_revenus()

BEGIN

TRUNCATE VM\_Revenus\_annee\_espece;

INSERT INTO VM\_Revenus\_annee\_espece

SELECT YEAR(date\_reservation) AS annee, Espece.id AS espece\_id, SUM(Adoption.prix) AS somme, COUNT(Adoption.animal\_id) AS nb

FROM Adoption

INNER JOIN Animal ON Animal.id = Adoption.animal\_id

INNER JOIN Espece ON Animal.espece\_id = Espece.id

GROUP BY annee, Espece.id;

END |

DELIMITER ;

La commande TRUNCATE nom\_table;  a le même effet que DELETE FROM nom\_table;  (sans clause WHERE) ; elle supprime toutes les lignes de la table. Cependant, TRUNCATE  est un peu différent de DELETE FROM.

* TRUNCATE  ne supprime pas les lignes une à une : TRUNCATE  supprime la table, puis la recrée (sans les données).
* Par conséquent, TRUNCATE  ne traite pas les clés étrangères : on ne peut pas faire un TRUNCATE  sur une table dont une colonne est référencée par une clé étrangère, sauf si celle-ci est dans la même table. En outre, les options ON DELETE  ne sont pas traitées lorsque des données sont supprimées avec cette commande.
* TRUNCATE  valide implicitement les transactions, et ne peut pas être annulé par un rollback.
* Pour toutes ces raisons, TRUNCATE  est beaucoup plus rapide que DELETE FROM  (en particulier si le nombre de lignes à supprimer est important).

Dans le cas de VM\_Revenus\_annee\_espece, on peut sans problème utiliser TRUNCATE.

Pour mettre à jour la vue matérialisée, il suffit donc d'exécuter la procédure stockée.

CALL maj\_vm\_revenus();

Ce type de mise à jour est généralement utilisé pour des données qu'il n'est pas impératif d'avoir à jour en permanence. La plupart du temps, la procédure est appelée par un script qui tourne périodiquement (une fois par jour ou une fois par semaine, par exemple).

#### Mise à jour automatique

Si, à l'inverse, on veut que les données soient toujours à jour par rapport aux derniers changements de la base de données, on utilisera plutôt les triggers pour mettre à jour la vue matérialisée.

Dans le cas de VM\_Revenus\_annee\_espece, la vue matérialisée doit être mise à jour en cas de modification de la table Adoption.

* Une insertion provoquera la mise à jour de la ligne correspondant à l'année et à l'espèce de l'adoption insérée (majoration du total des prix et du nombre d'adoptions), ou insérera une nouvelle ligne si elle n'existe pas encore.
* Une suppression provoquera la mise à jour de la ligne correspondant à l'année et à l'espèce de l'adoption supprimée, ou la suppression de celle-ci s'il s'agissait de la seule adoption correspondante.
* Une modification sera un mix de la suppression et de l'insertion.

En ce qui concerne la colonne espece\_id de la vue matérialisée, il vaut mieux lui ajouter une clé étrangère, avec l'option ON DELETE CASCADE. En principe, Espece.id ne devrait jamais être modifiée, mais en mettant cette clé étrangère, on s'assure que la correspondance entre les tables existera toujours.

On va également ajouter une clé primaire, (annee, espece\_id), afin de simplifier nos triggers. Cela permettra d'utiliser la commande INSERT INTO ... ON DUPLICATE KEY UPDATE.

ALTER TABLE VM\_Revenus\_annee\_espece

ADD CONSTRAINT fk\_vm\_revenu\_espece\_id FOREIGN KEY (espece\_id) REFERENCES Espece (id) ON DELETE CASCADE,

ADD PRIMARY KEY (annee, espece\_id);

Et voici donc les triggers nécessaires à la mise à jour de la vue matérialisée :

Ces trois triggers existaient déjà et permettaient de mettre à jour la colonne Animal.disponible. Il faut donc d'abord les supprimer, puis les recréer en ajoutant la mise à jour de VM\_Revenus\_annee\_espece.

DELIMITER |

DROP TRIGGER after\_insert\_adoption |

CREATE TRIGGER after\_insert\_adoption AFTER INSERT

ON Adoption FOR EACH ROW

BEGIN

UPDATE Animal

*SET* disponible = FALSE

WHERE id = NEW.animal\_id;

INSERT INTO VM\_Revenus\_annee\_espece (espece\_id, annee, somme, nb)

SELECT espece\_id, YEAR(NEW.date\_reservation), NEW.prix, 1

FROM Animal

WHERE id = NEW.animal\_id

ON DUPLICATE KEY UPDATE somme = somme + NEW.prix, nb = nb + 1;

END |

DROP TRIGGER after\_update\_adoption |

CREATE TRIGGER after\_update\_adoption AFTER UPDATE

ON Adoption FOR EACH ROW

BEGIN

IF OLD.animal\_id <> NEW.animal\_id THEN

UPDATE Animal

*SET* disponible = TRUE

WHERE id = OLD.animal\_id;

UPDATE Animal

*SET* disponible = FALSE

WHERE id = NEW.animal\_id;

END IF;

INSERT INTO VM\_Revenus\_annee\_espece (espece\_id, annee, somme, nb)

SELECT espece\_id, YEAR(NEW.date\_reservation), NEW.prix, 1

FROM Animal

WHERE id = NEW.animal\_id

ON DUPLICATE KEY UPDATE somme = somme + NEW.prix, nb = nb + 1;

UPDATE VM\_Revenus\_annee\_espece

*SET* somme = somme - OLD.prix, nb = nb - 1

WHERE annee = YEAR(OLD.date\_reservation)

AND espece\_id = (SELECT espece\_id FROM Animal WHERE id = OLD.animal\_id);

DELETE FROM VM\_Revenus\_annee\_espece

WHERE nb = 0;

END |

DROP TRIGGER after\_delete\_adoption |

CREATE TRIGGER after\_delete\_adoption AFTER DELETE

ON Adoption FOR EACH ROW

BEGIN

UPDATE Animal

*SET* disponible = TRUE

WHERE id = OLD.animal\_id;

UPDATE VM\_Revenus\_annee\_espece

*SET* somme = somme - OLD.prix, nb = nb - 1

WHERE annee = YEAR(OLD.date\_reservation)

AND espece\_id = (SELECT espece\_id FROM Animal WHERE id = OLD.animal\_id);

DELETE FROM VM\_Revenus\_annee\_espece

WHERE nb = 0;

END |

DELIMITER ;

### Gain de performance

#### Tables vs vue vs vue matérialisée

On va tester les performances des vues et des vues matérialisées en créant trois procédures stockées différentes répondant à la même question : quelle est l'année ayant rapporté le plus en termes d'adoption de chats ?  
Les trois procédures utiliseront des objets différents :

* l'une fera la requête directement sur les tables ;
* l'autre fera la requête sur la vue ;
* la dernière utilisera la vue matérialisée.

Nos tables contenant très peu de données, la requête sera répétée un millier de fois afin que les temps d'exécution soient utilisables.

##### Empêcher MySQL d'utiliser le cache

Lorsque l'on exécute la même requête plusieurs fois de suite, MySQL ne l'exécute en réalité qu'une seule fois, et stocke le résultat en cache. Toutes les fois suivantes, MySQL ressort simplement ce qu'il a en cache, ce qui ne prend quasiment rien comme temps.  
Pour faire des tests de performance, cela pose évidemment problème, puisque cela fausse le temps d'exécution. Heureusement, il existe une clause spéciale qui permet d'empêcher MySQL de stocker le résultat de la requête en cache : SQL\_NO\_CACHE.

#### Les trois procédures

##### Sur les tables

On utilise SELECT... INTO  afin de stocker les résultats dans des variables, plutôt que de les afficher mille fois.

DELIMITER |

CREATE PROCEDURE test\_perf\_table()

BEGIN

DECLARE v\_max *INT* DEFAULT 1000;

DECLARE v\_i *INT* DEFAULT 0;

DECLARE v\_nb *INT*;

DECLARE v\_somme *DECIMAL*(15,2);

DECLARE v\_annee *CHAR*(4);

boucle: LOOP

IF v\_i = v\_max THEN LEAVE boucle; END IF;

-- Condition d'arrêt de la boucle

SELECT SQL\_NO\_CACHE YEAR(date\_reservation) AS annee,

SUM(Adoption.prix) AS somme,

COUNT(Adoption.animal\_id) AS nb

INTO v\_annee, v\_somme, v\_nb

FROM Adoption

INNER JOIN Animal ON Animal.id = Adoption.animal\_id

INNER JOIN Espece ON Animal.espece\_id = Espece.id

WHERE Espece.id = 2

GROUP BY annee

ORDER BY somme DESC

LIMIT 1;

*SET* v\_i = v\_i + 1;

END LOOP;

END |

DELIMITER ;

##### Sur la vue

DELIMITER |

CREATE PROCEDURE test\_perf\_vue()

BEGIN

DECLARE v\_max *INT* DEFAULT 1000;

DECLARE v\_i *INT* DEFAULT 0;

DECLARE v\_nb *INT*;

DECLARE v\_somme *DECIMAL*(15,2);

DECLARE v\_annee *CHAR*(4);

boucle: LOOP

IF v\_i = v\_max THEN LEAVE boucle; END IF;

SELECT SQL\_NO\_CACHE annee, somme, nb

INTO v\_annee, v\_somme, v\_nb

FROM V\_Revenus\_annee\_espece

WHERE espece\_id = 2

ORDER BY somme DESC

LIMIT 1;

*SET* v\_i = v\_i + 1;

END LOOP;

END |

DELIMITER ;

##### Sur la vue matérialisée

DELIMITER |

CREATE PROCEDURE test\_perf\_vm()

BEGIN

DECLARE v\_max *INT* DEFAULT 1000;

DECLARE v\_i *INT* DEFAULT 0;

DECLARE v\_nb *INT*;

DECLARE v\_somme *DECIMAL*(15,2);

DECLARE v\_annee *CHAR*(4);

boucle: LOOP

IF v\_i = v\_max THEN LEAVE boucle; END IF;

SELECT SQL\_NO\_CACHE annee, somme, nb

INTO v\_annee, v\_somme, v\_nb

FROM VM\_Revenus\_annee\_espece

WHERE espece\_id = 2

ORDER BY somme DESC

LIMIT 1;

*SET* v\_i = v\_i + 1;

END LOOP;

END |

DELIMITER ;

#### Le test

Il n'y a plus maintenant qu'à exécuter les trois procédures pour voir leur temps d'exécution.

CALL test\_perf\_table();

CALL test\_perf\_vue();

CALL test\_perf\_vm();

mysql> CALL test\_perf\_table();

Query OK, 1 row affected (0.27 sec)

mysql> CALL test\_perf\_vue();

Query OK, 1 row affected (0.29 sec)

mysql> CALL test\_perf\_vm();

Query OK, 1 row affected (0.06 sec)

Le temps d'exécution semble équivalent pour la requête faite sur les tables directement, et sur la vue. Par contre, on a un gros gain de performance avec la requête sur la vue matérialisée.

Voici les moyennes et écarts types des temps d'exécution des trois procédures, pour 20 exécutions :

|  | **Sur tables** | **Sur vue** | **Sur vue matérialisée** |
| --- | --- | --- | --- |
| **Moyenne** | 0,266 s | 0,2915 s | 0,058 |
| **Écart type** | 0,00503 | 0,00366 | 0,00410 |

La requête sur la vue est donc bien légèrement plus lente que la requête faite directement sur les tables. Mais le résultat le plus intéressant est bien celui obtenu avec la vue matérialisée : 5 fois plus rapide que la requête sur la vue !

Par ailleurs, on peut maintenant ajouter des clés et des index à VM\_Revenus\_annee\_especes pour accélérer encore les requêtes. Ici, le nombre réduit de lignes fait que l'effet ne sera pas perceptible, mais sur des tables plus fournies, cela vaut la peine !

ALTER TABLE VM\_Revenus\_annee\_espece ADD INDEX (somme);

#### Conclusion

Nos tables contiennent très peu de données par rapport à la réalité. Les applications moyennes classiques exploitent des bases de données dont les tables peuvent compter des centaines de milliers, voire des millions de lignes. Le gain de performance permis par les vues matérialisées, et les index, sera encore plus visible sur de telles bases.

Cependant, **toutes les vues ne sont pas intéressantes à matérialiser**. De manière générale, une vue contenant des fonctions d'agrégation, des regroupements, et qui donc devra utiliser l'algorithme TEMPTABLE, est une bonne candidate.

Mais il faut également prendre en compte le coût de la mise à jour de la vue matérialisée.  
S'il s'agit d'une vue qui doit être constamment à jour, et qui utiliserait donc des triggers, il ne sera sans doute pas intéressant de la matérialiser si les données sur lesquelles elle s'appuie sont souvent modifiées, car **le gain de performance pour les requêtes de sélection sur la vue ne compenserait pas le coût des mises à jour de celle-ci**.

Par ailleurs, **certaines vues ne nécessitent pas qu'on leur consacre une table entière**. Ainsi, la vue V\_Nombre\_espece peut être matérialisée simplement en ajoutant une colonne à la table Espece. Cette colonne peut alors être tenue à jour avec des triggers sur Animal (comme on le fait avec la colonne Animal.disponible).

Comme souvent, il faut donc bien choisir au cas par cas, peser le pour et le contre et réfléchir à toutes les implications quant au choix des vues à matérialiser.

## Gestion des utilisateurs

À plusieurs reprises dans ce cours, j'ai mentionné la possibilité de créer des utilisateurs et de leur permettre de faire certaines actions. Le moment est venu de découvrir comment faire.  
Au sommaire :

* création, modification et suppression d'utilisateurs ;
* explication des privilèges et options des utilisateurs ;
* attribution et révocation des privilèges des utilisateurs.

### État actuel de la base de données

**Note :** les tables de test, les vues de test et les procédures stockées ne sont pas reprises.

*SET* NAMES utf8;

DROP VIEW IF EXISTS V\_Animal\_details;

DROP VIEW IF EXISTS V\_Animal\_espece;

DROP VIEW IF EXISTS V\_Animal\_stagiaire;

DROP VIEW IF EXISTS V\_Chien;

DROP VIEW IF EXISTS V\_Chien\_race;

DROP VIEW IF EXISTS V\_Client;

DROP VIEW IF EXISTS V\_Espece\_dollars;

DROP VIEW IF EXISTS V\_Nombre\_espece;

DROP VIEW IF EXISTS V\_Revenus\_annee\_espece;

DROP TABLE IF EXISTS Erreur;

DROP TABLE IF EXISTS Animal\_histo;

DROP TABLE IF EXISTS VM\_Revenus\_annee\_espece;

DROP TABLE IF EXISTS Adoption;

DROP TABLE IF EXISTS Animal;

DROP TABLE IF EXISTS Race;

DROP TABLE IF EXISTS Espece;

DROP TABLE IF EXISTS Client;

CREATE TABLE Client (

id smallint(5) unsigned NOT NULL AUTO\_INCREMENT,

nom *varchar*(100) NOT NULL,

prenom *varchar*(60) NOT NULL,

adresse *varchar*(200) DEFAULT NULL,

code\_postal *varchar*(6) DEFAULT NULL,

ville *varchar*(60) DEFAULT NULL,

pays *varchar*(60) DEFAULT NULL,

email varbinary(100) DEFAULT NULL,

date\_naissance *date* DEFAULT NULL,

PRIMARY KEY (id),

UNIQUE KEY ind\_uni\_email (email)

) ENGINE=InnoDB AUTO\_INCREMENT=17;

LOCK TABLES Client WRITE;

INSERT INTO Client VALUES (1,'Dupont','Jean','Rue du Centre, 5','45810','Houtsiplou','France','jean.dupont@email.com',NULL),(2,'Boudur','Marie','Place de la Gare, 2','35840','Troudumonde','France','marie.boudur@email.com',NULL),(3,'Trachon','Fleur','Rue haute, 54b','3250','Belville','Belgique','fleurtrachon@email.com',NULL),

(4,'Van Piperseel','Julien',NULL,NULL,NULL,NULL,'jeanvp@email.com',NULL),

(5,'Nouvel','Johan',NULL,NULL,NULL,'Suisse','johanetpirlouit@email.com',NULL),(6,'Germain','Frank',NULL,NULL,NULL,NULL,'francoisgermain@email.com',NULL),

(7,'Antoine','Maximilien','Rue Moineau, 123','4580','Trocoul','Belgique','max.antoine@email.com',NULL),(8,'Di Paolo','Hector',NULL,NULL,NULL,'Suisse','hectordipao@email.com',NULL),

(9,'Corduro','Anaelle',NULL,NULL,NULL,NULL,'ana.corduro@email.com',NULL),

(10,'Faluche','Eline','Avenue circulaire, 7','45870','Garduche','France','elinefaluche@email.com',NULL),(11,'Penni','Carine','Boulevard Haussman, 85','1514','Plasse','Suisse','cpenni@email.com',NULL),(12,'Broussaille','Virginie','Rue du Fleuve, 18','45810','Houtsiplou','France','vibrousaille@email.com',NULL),

(13,'Durant','Hannah','Rue des Pendus, 66','1514','Plasse','Suisse','hhdurant@email.com',NULL),

(14,'Delfour','Elodie','Rue de Flore, 1','3250','Belville','Belgique','e.delfour@email.com',NULL),(15,'Kestau','Joel',NULL,NULL,NULL,NULL,'joel.kestau@email.com',NULL);

UNLOCK TABLES;

CREATE TABLE Espece (

id smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

nom\_courant *varchar*(40) NOT NULL,

nom\_latin *varchar*(40) NOT NULL,

description *text*,

prix *decimal*(7,2) unsigned DEFAULT NULL,

PRIMARY KEY (id),

UNIQUE KEY nom\_latin (nom\_latin)

) ENGINE=InnoDB AUTO\_INCREMENT=8;

LOCK TABLES Espece WRITE;

INSERT INTO Espece VALUES (1,'Chien','Canis canis','Bestiole à quatre pattes qui aime les caresses et tire souvent la langue',200.00),(2,'Chat','Felis silvestris','Bestiole à quatre pattes qui saute très haut et grimpe aux arbres',150.00),

(3,'Tortue d''Hermann','Testudo hermanni','Bestiole avec une carapace très dure',140.00),

(4,'Perroquet amazone','Alipiopsitta xanthops','Joli oiseau parleur vert et jaune',700.00),(5,'Rat brun','Rattus norvegicus','Petite bestiole avec de longues moustaches et une longue queue sans poils',10.00),(6,'Perruche terrestre','Pezoporus wallicus','Joli oiseau aux plumes majoritairement vert brillant',20.00);

UNLOCK TABLES;

CREATE TABLE Race (

id smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

nom *varchar*(40) NOT NULL,

espece\_id smallint(6) unsigned NOT NULL,

description *text*,

prix *decimal*(7,2) unsigned DEFAULT NULL,

date\_insertion datetime DEFAULT NULL,

utilisateur\_insertion *varchar*(20) DEFAULT NULL,

date\_modification datetime DEFAULT NULL,

utilisateur\_modification *varchar*(20) DEFAULT NULL,

PRIMARY KEY (id)

) ENGINE=InnoDB AUTO\_INCREMENT=11;

LOCK TABLES Race WRITE;

INSERT INTO Race VALUES (1,'Berger allemand',1,'Chien sportif et élégant au pelage dense, noir-marron-fauve, noir ou gris.',485.00,'2012-05-21 00:53:36','Test','2012-05-21 00:53:36','Test'),(2,'Berger blanc suisse',1,'Petit chien au corps compact, avec des pattes courtes mais bien proportionnées et au pelage tricolore ou bicolore.',935.00,'2012-05-21 00:53:36','Test','2012-05-21 00:53:36','Test'),

(3,'Singapura',2,'Chat de petite taille aux grands yeux en amandes.',985.00,'2012-05-21 00:53:36','Test','2012-05-21 00:53:36','Test'),

(4,'Bleu russe',2,'Chat aux yeux verts et à la robe épaisse et argentée.',835.00,'2012-05-21 00:53:36','Test','2012-05-21 00:53:36','Test'),(5,'Maine Coon',2,'Chat de grande taille, à poils mi-longs.',735.00,'2012-05-21 00:53:36','Test','2012-05-27 18:10:46','student@localhost'),(7,'Sphynx',2,'Chat sans poils.',1235.00,'2012-05-21 00:53:36','Test','2012-05-21 00:53:36','Test'),

(8,'Nebelung',2,'Chat bleu russe, mais avec des poils longs...',985.00,'2012-05-21 00:53:36','Test','2012-05-21 00:53:36','Test'),

(9,'Rottweiller',1,'Chien d''apparence solide, bien musclé, à la robe noire avec des taches feu bien délimitées.',630.00,'2012-05-21 00:53:36','Test','2012-05-22 00:54:13','student@localhost'),(10,'Yorkshire terrier',1,'Chien de petite taille au pelage long et soyeux de couleur bleu et feu.',700.00,'2012-05-22 00:58:25','student@localhost','2012-05-22 00:58:25','student@localhost');

UNLOCK TABLES;

CREATE TABLE Animal (

id smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

sexe *char*(1) DEFAULT NULL,

date\_naissance datetime NOT NULL,

nom *varchar*(30) DEFAULT NULL,

commentaires *text*,

espece\_id smallint(6) unsigned NOT NULL,

race\_id smallint(6) unsigned DEFAULT NULL,

mere\_id smallint(6) unsigned DEFAULT NULL,

pere\_id smallint(6) unsigned DEFAULT NULL,

disponible tinyint(1) DEFAULT '1',

PRIMARY KEY (id),

UNIQUE KEY ind\_uni\_nom\_espece\_id (nom,espece\_id)

) ENGINE=InnoDB AUTO\_INCREMENT=81;

LOCK TABLES Animal WRITE;

INSERT INTO Animal VALUES (1,'M','2010-04-05 13:43:00','Rox','Mordille beaucoup',1,1,18,22,1),(2,NULL,'2010-03-24 02:23:00','Roucky',NULL,2,NULL,40,30,1),(3,'F','2010-09-13 15:02:00','Schtroumpfette',NULL,2,4,41,31,0),

(4,'F','2009-08-03 05:12:00',NULL,'Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),(5,NULL,'2010-10-03 16:44:00','Choupi','Né sans oreille gauche',2,NULL,NULL,NULL,0),(6,'F','2009-06-13 08:17:00','Bobosse','Carapace bizarre',3,NULL,NULL,NULL,1),

(7,'F','2008-12-06 05:18:00','Caroline',NULL,1,2,NULL,NULL,1),(8,'M','2008-09-11 15:38:00','Bagherra',NULL,2,5,NULL,NULL,0),(9,NULL,'2010-08-23 05:18:00',NULL,'Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),

(10,'M','2010-07-21 15:41:00','Bobo','Petit pour son âge',1,NULL,7,21,1),(11,'F','2008-02-20 15:45:00','Canaille',NULL,1,NULL,NULL,NULL,1),(12,'F','2009-05-26 08:54:00','Cali',NULL,1,2,NULL,NULL,1),

(13,'F','2007-04-24 12:54:00','Rouquine',NULL,1,1,NULL,NULL,1),(14,'F','2009-05-26 08:56:00','Fila',NULL,1,2,NULL,NULL,1),(15,'F','2008-02-20 15:47:00','Anya',NULL,1,NULL,NULL,NULL,0),

(16,'F','2009-05-26 08:50:00','Louya',NULL,1,NULL,NULL,NULL,0),(17,'F','2008-03-10 13:45:00','Welva',NULL,1,NULL,NULL,NULL,1),(18,'F','2007-04-24 12:59:00','Zira',NULL,1,1,NULL,NULL,0),

(19,'F','2009-05-26 09:02:00','Java',NULL,2,4,NULL,NULL,1),(20,NULL,'2007-04-24 12:45:00','Balou',NULL,1,1,NULL,NULL,1),(21,'F','2008-03-10 13:43:00','Pataude','Rhume chronique',1,NULL,NULL,NULL,0),

(22,'M','2007-04-24 12:42:00','Bouli',NULL,1,1,NULL,NULL,1),(24,'M','2007-04-12 05:23:00','Cartouche',NULL,1,NULL,NULL,NULL,1),(25,'M','2006-05-14 15:50:00','Zambo',NULL,1,1,NULL,NULL,1),

(26,'M','2006-05-14 15:48:00','Samba',NULL,1,1,NULL,NULL,0),(27,'M','2008-03-10 13:40:00','Moka',NULL,1,NULL,NULL,NULL,0),(28,'M','2006-05-14 15:40:00','Pilou',NULL,1,1,NULL,NULL,1),

(29,'M','2009-05-14 06:30:00','Fiero',NULL,2,3,NULL,NULL,1),(30,'M','2007-03-12 12:05:00','Zonko',NULL,2,5,NULL,NULL,0),(31,'M','2008-02-20 15:45:00','Filou',NULL,2,4,NULL,NULL,1),

(32,'M','2009-07-26 11:52:00','Spoutnik',NULL,3,NULL,52,NULL,0),(33,'M','2006-05-19 16:17:00','Caribou',NULL,2,4,NULL,NULL,1),(34,'M','2008-04-20 03:22:00','Capou',NULL,2,5,NULL,NULL,1),

(35,'M','2006-05-19 16:56:00','Raccou','Pas de queue depuis la naissance',2,4,NULL,NULL,1),(36,'M','2009-05-14 06:42:00','Boucan',NULL,2,3,NULL,NULL,1),(37,'F','2006-05-19 16:06:00','Callune',NULL,2,8,NULL,NULL,1),

(38,'F','2009-05-14 06:45:00','Boule',NULL,2,3,NULL,NULL,0),(39,'F','2008-04-20 03:26:00','Zara',NULL,2,5,NULL,NULL,0),(40,'F','2007-03-12 12:00:00','Milla',NULL,2,5,NULL,NULL,0),

(41,'F','2006-05-19 15:59:00','Feta',NULL,2,4,NULL,NULL,0),(42,'F','2008-04-20 03:20:00','Bilba','Sourde de l''oreille droite à 80%',2,5,NULL,NULL,0),(43,'F','2007-03-12 11:54:00','Cracotte',NULL,2,5,NULL,NULL,1),

(44,'F','2006-05-19 16:16:00','Cawette',NULL,2,8,NULL,NULL,1),(45,'F','2007-04-01 18:17:00','Nikki','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),(46,'F','2009-03-24 08:23:00','Tortilla','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),

(48,'F','2006-03-15 14:56:00','Lulla','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),(49,'F','2008-03-15 12:02:00','Dana','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,0),(50,'F','2009-05-25 19:57:00','Cheli','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),

(51,'F','2007-04-01 03:54:00','Chicaca','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),(52,'F','2006-03-15 14:26:00','Redbul','Insomniaque',3,NULL,NULL,NULL,1),

(54,'M','2008-03-16 08:20:00','Bubulle','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),

(55,'M','2008-03-15 18:45:00','Relou','Surpoids',3,NULL,NULL,NULL,0),(56,'M','2009-05-25 18:54:00','Bulbizard','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1),

(57,'M','2007-03-04 19:36:00','Safran','Coco veut un gâteau !',4,NULL,NULL,NULL,0),

(58,'M','2008-02-20 02:50:00','Gingko','Coco veut un gâteau !',4,NULL,NULL,NULL,0),(59,'M','2009-03-26 08:28:00','Bavard','Coco veut un gâteau !',4,NULL,NULL,NULL,0),

(60,'F','2009-03-26 07:55:00','Parlotte','Coco veut un gâteau !',4,NULL,NULL,NULL,1),(61,'M','2010-11-09 00:00:00','Yoda',NULL,2,5,NULL,NULL,1),(62,'M','2010-11-05 00:00:00','Pipo',NULL,1,9,NULL,NULL,0),

(69,'F','2012-02-13 15:45:00','Baba',NULL,5,NULL,NULL,NULL,0),(70,'M','2012-02-13 15:48:00','Bibo','Agressif',5,NULL,72,73,1),(72,'F','2008-02-01 02:25:00','Momy',NULL,5,NULL,NULL,NULL,1),

(73,'M','2007-03-11 12:45:00','Popi',NULL,5,NULL,NULL,NULL,1),(75,'F','2007-03-12 22:03:00','Mimi',NULL,5,NULL,NULL,NULL,0),(76,'M','2012-03-12 00:00:00','Rocco',NULL,1,9,NULL,NULL,1),

(77,'F','2011-09-21 15:14:00','Bambi',NULL,2,NULL,NULL,NULL,1),(78,'M','2012-02-28 03:05:00','Pumba','Prématuré, à surveiller',1,9,NULL,NULL,1),(79,'M','2011-05-24 23:51:00','Lion',NULL,2,5,NULL,NULL,1);

UNLOCK TABLES;

CREATE TABLE Adoption (

client\_id smallint(5) unsigned NOT NULL,

animal\_id smallint(5) unsigned NOT NULL,

date\_reservation *date* NOT NULL,

date\_adoption *date* DEFAULT NULL,

prix *decimal*(7,2) unsigned NOT NULL,

paye tinyint(1) NOT NULL DEFAULT '0',

PRIMARY KEY (client\_id,animal\_id),

UNIQUE KEY ind\_uni\_animal\_id (animal\_id)

) ENGINE=InnoDB;

LOCK TABLES Adoption WRITE;

INSERT INTO Adoption VALUES (1,8,'2012-05-21',NULL,735.00,1),(1,39,'2008-08-17','2008-08-17',735.00,1),(1,40,'2008-08-17','2008-08-17',735.00,1),

(2,3,'2011-03-12','2011-03-12',835.00,1),(2,18,'2008-06-04','2008-06-04',485.00,1),(3,27,'2009-11-17','2009-11-17',200.00,1),

(4,26,'2007-02-21','2007-02-21',485.00,1),(4,41,'2007-02-21','2007-02-21',835.00,1),(5,21,'2009-03-08','2009-03-08',200.00,1),

(6,16,'2010-01-27','2010-01-27',200.00,1),(7,5,'2011-04-05','2011-04-05',150.00,1),(8,42,'2008-08-16','2008-08-16',735.00,1),

(9,38,'2007-02-11','2007-02-11',985.00,1),(9,55,'2011-02-13','2011-02-13',140.00,1),(9,59,'2012-05-22',NULL,700.00,0),

(10,49,'2010-08-17','2010-08-17',140.00,0),(11,32,'2008-08-17','2010-03-09',140.00,1),(11,62,'2011-03-01','2011-03-01',630.00,0),

(12,15,'2012-05-22',NULL,200.00,1),(12,69,'2007-09-20','2007-09-20',10.00,1),(12,75,'2012-05-21',NULL,10.00,0),

(13,57,'2012-01-10','2012-01-10',700.00,1),(14,58,'2012-02-25','2012-02-25',700.00,1),(15,30,'2008-08-17','2008-08-17',735.00,1);

UNLOCK TABLES;

CREATE TABLE VM\_Revenus\_annee\_espece (

annee *int*(4) NOT NULL DEFAULT '0',

espece\_id smallint(6) unsigned NOT NULL DEFAULT '0',

somme *decimal*(29,2) DEFAULT NULL,

nb *bigint*(21) NOT NULL DEFAULT '0',

PRIMARY KEY (annee,espece\_id),

KEY somme (somme)

) ENGINE=InnoDB;

LOCK TABLES VM\_Revenus\_annee\_espece WRITE;

INSERT INTO VM\_Revenus\_annee\_espece VALUES (2007,1,485.00,1),(2007,2,1820.00,2),(2007,5,10.00,1),(2008,1,485.00,1),(2008,2,2940.00,4),

(2008,3,140.00,1),(2009,1,400.00,2),(2010,1,200.00,1),(2010,3,140.00,1),(2011,1,630.00,1),

(2011,2,985.00,2),(2011,3,140.00,1),(2012,1,200.00,1),(2012,2,735.00,1),(2012,4,2100.00,3),(2012,5,10.00,1);

UNLOCK TABLES;

CREATE TABLE Animal\_histo (

id smallint(6) unsigned NOT NULL,

sexe *char*(1) DEFAULT NULL,

date\_naissance datetime NOT NULL,

nom *varchar*(30) DEFAULT NULL,

commentaires *text*,

espece\_id smallint(6) unsigned NOT NULL,

race\_id smallint(6) unsigned DEFAULT NULL,

mere\_id smallint(6) unsigned DEFAULT NULL,

pere\_id smallint(6) unsigned DEFAULT NULL,

disponible tinyint(1) DEFAULT '1',

date\_histo datetime NOT NULL,

utilisateur\_histo *varchar*(20) NOT NULL,

evenement\_histo *char*(6) NOT NULL,

PRIMARY KEY (id,date\_histo)

) ENGINE=InnoDB;

LOCK TABLES Animal\_histo WRITE;

INSERT INTO Animal\_histo VALUES (10,'M','2010-07-21 15:41:00','Bobo',NULL,1,NULL,7,21,1,'2012-05-22 01:00:34','student@localhost','UPDATE'),(19,'F','2009-05-26 09:02:00','Java',NULL,1,2,NULL,NULL,1,'2012-05-27 18:14:29','student@localhost','UPDATE'),(21,'F','2008-03-10 13:43:00','Pataude',NULL,1,NULL,NULL,NULL,0,'2012-05-27 18:10:40','student@localhost','UPDATE'),

(47,'F','2009-03-26 01:24:00','Scroupy','Bestiole avec une carapace très dure',3,NULL,NULL,NULL,1,'2012-05-22 01:00:34','student@localhost','DELETE'),(77,'F','0000-00-00 00:00:00','Toxi',NULL,1,NULL,NULL,NULL,1,'2012-05-27 18:12:38','student@localhost','DELETE');

UNLOCK TABLES;

CREATE TABLE Erreur (

id tinyint(3) unsigned NOT NULL AUTO\_INCREMENT,

erreur *varchar*(255) DEFAULT NULL,

PRIMARY KEY (id),

UNIQUE KEY erreur (erreur)

) ENGINE=InnoDB AUTO\_INCREMENT=8;

LOCK TABLES Erreur WRITE;

INSERT INTO Erreur VALUES (5,'Erreur : date\_adoption doit être >= à date\_reservation.'),(3,'Erreur : paye doit valoir TRUE (1) ou FALSE (0).'),(1,'Erreur : sexe doit valoir \"M\", \"F\" ou NULL.');

UNLOCK TABLES;

ALTER TABLE Race ADD CONSTRAINT fk\_race\_espece\_id FOREIGN KEY (espece\_id) REFERENCES Espece (id);

ALTER TABLE Animal ADD CONSTRAINT fk\_espece\_id FOREIGN KEY (espece\_id) REFERENCES Espece (id);

ALTER TABLE Animal ADD CONSTRAINT fk\_mere\_id FOREIGN KEY (mere\_id) REFERENCES Animal (id);

ALTER TABLE Animal ADD CONSTRAINT fk\_pere\_id FOREIGN KEY (pere\_id) REFERENCES Animal (id);

ALTER TABLE Animal ADD CONSTRAINT fk\_race\_id FOREIGN KEY (race\_id) REFERENCES Race (id);

ALTER TABLE Adoption ADD CONSTRAINT fk\_adoption\_animal\_id FOREIGN KEY (animal\_id) REFERENCES Animal (id);

ALTER TABLE Adoption ADD CONSTRAINT fk\_client\_id FOREIGN KEY (client\_id) REFERENCES Client (id);

ALTER TABLE VM\_Revenus\_annee\_espece ADD CONSTRAINT fk\_vm\_revenu\_espece\_id FOREIGN KEY (espece\_id) REFERENCES Espece (id) ON DELETE CASCADE;

DELIMITER |

CREATE TRIGGER before\_insert\_adoption BEFORE INSERT

ON Adoption FOR EACH ROW

BEGIN

IF NEW.paye != TRUE

AND NEW.paye != FALSE

THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : paye doit valoir TRUE (1) ou FALSE (0).');

ELSEIF NEW.date\_adoption < NEW.date\_reservation THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : date\_adoption doit être >= à date\_reservation.');

END IF;

END |

CREATE TRIGGER after\_insert\_adoption AFTER INSERT

ON Adoption FOR EACH ROW

BEGIN

UPDATE Animal

*SET* disponible = FALSE

WHERE id = NEW.animal\_id;

INSERT INTO VM\_Revenus\_annee\_espece (espece\_id, annee, somme, nb)

SELECT espece\_id, YEAR(NEW.date\_reservation), NEW.prix, 1

FROM Animal

WHERE id = NEW.animal\_id

ON DUPLICATE KEY UPDATE somme = somme + NEW.prix, nb = nb + 1;

END |

CREATE TRIGGER before\_update\_adoption BEFORE UPDATE

ON Adoption FOR EACH ROW

BEGIN

IF NEW.paye != TRUE

AND NEW.paye != FALSE

THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : paye doit valoir TRUE (1) ou FALSE (0).');

ELSEIF NEW.date\_adoption < NEW.date\_reservation THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : date\_adoption doit être >= à date\_reservation.');

END IF;

END |

CREATE TRIGGER after\_update\_adoption AFTER UPDATE

ON Adoption FOR EACH ROW

BEGIN

IF OLD.animal\_id <> NEW.animal\_id THEN

UPDATE Animal

*SET* disponible = TRUE

WHERE id = OLD.animal\_id;

UPDATE Animal

*SET* disponible = FALSE

WHERE id = NEW.animal\_id;

END IF;

INSERT INTO VM\_Revenus\_annee\_espece (espece\_id, annee, somme, nb)

SELECT espece\_id, YEAR(NEW.date\_reservation), NEW.prix, 1

FROM Animal

WHERE id = NEW.animal\_id

ON DUPLICATE KEY UPDATE somme = somme + NEW.prix, nb = nb + 1;

UPDATE VM\_Revenus\_annee\_espece

*SET* somme = somme - OLD.prix, nb = nb - 1

WHERE annee = YEAR(OLD.date\_reservation)

AND espece\_id = (SELECT espece\_id FROM Animal WHERE id = OLD.animal\_id);

DELETE FROM VM\_Revenus\_annee\_espece

WHERE nb = 0;

END |

CREATE TRIGGER after\_delete\_adoption AFTER DELETE

ON Adoption FOR EACH ROW

BEGIN

UPDATE Animal

*SET* disponible = TRUE

WHERE id = OLD.animal\_id;

UPDATE VM\_Revenus\_annee\_espece

*SET* somme = somme - OLD.prix, nb = nb - 1

WHERE annee = YEAR(OLD.date\_reservation)

AND espece\_id = (SELECT espece\_id FROM Animal WHERE id = OLD.animal\_id);

DELETE FROM VM\_Revenus\_annee\_espece

WHERE nb = 0;

END |

CREATE TRIGGER before\_insert\_animal BEFORE INSERT

ON Animal FOR EACH ROW

BEGIN

IF NEW.sexe IS NOT NULL

AND NEW.sexe != 'M'

AND NEW.sexe != 'F'

AND NEW.sexe != 'A'

THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : sexe doit valoir "M", "F", "A" ou NULL.');

END IF;

END |

CREATE TRIGGER before\_update\_animal BEFORE UPDATE

ON Animal FOR EACH ROW

BEGIN

IF NEW.sexe IS NOT NULL

AND NEW.sexe != 'M'

AND NEW.sexe != 'F'

AND NEW.sexe != 'A'

THEN

*SET* NEW.sexe = NULL;

END IF;

END |

CREATE TRIGGER after\_update\_animal AFTER UPDATE

ON Animal FOR EACH ROW

BEGIN

INSERT INTO Animal\_histo (

id, sexe, date\_naissance, nom, commentaires, espece\_id, race\_id,

mere\_id, pere\_id, disponible, date\_histo, utilisateur\_histo, evenement\_histo)

VALUES (

OLD.id, OLD.sexe, OLD.date\_naissance, OLD.nom, OLD.commentaires, OLD.espece\_id, OLD.race\_id,

OLD.mere\_id, OLD.pere\_id, OLD.disponible, NOW(), CURRENT\_USER(), 'UPDATE');

END |

CREATE TRIGGER after\_delete\_animal AFTER DELETE

ON Animal FOR EACH ROW

BEGIN

INSERT INTO Animal\_histo (

id, sexe, date\_naissance, nom, commentaires, espece\_id, race\_id,

mere\_id, pere\_id, disponible, date\_histo, utilisateur\_histo, evenement\_histo)

VALUES (

OLD.id, OLD.sexe, OLD.date\_naissance, OLD.nom, OLD.commentaires, OLD.espece\_id, OLD.race\_id,

OLD.mere\_id, OLD.pere\_id, OLD.disponible, NOW(), CURRENT\_USER(), 'DELETE');

END |

CREATE TRIGGER before\_delete\_espece BEFORE DELETE

ON Espece FOR EACH ROW

BEGIN

DELETE FROM Race

WHERE espece\_id = OLD.id;

END |

CREATE TRIGGER before\_insert\_race BEFORE INSERT

ON Race FOR EACH ROW

BEGIN

*SET* NEW.date\_insertion = NOW();

*SET* NEW.utilisateur\_insertion = CURRENT\_USER();

*SET* NEW.date\_modification = NOW();

*SET* NEW.utilisateur\_modification = CURRENT\_USER();

END |

CREATE TRIGGER before\_update\_race BEFORE UPDATE

ON Race FOR EACH ROW

BEGIN

*SET* NEW.date\_modification = NOW();

*SET* NEW.utilisateur\_modification = CURRENT\_USER();

END |

CREATE TRIGGER before\_delete\_race BEFORE DELETE

ON Race FOR EACH ROW

BEGIN

UPDATE Animal

*SET* race\_id = NULL

WHERE race\_id = OLD.id;

END |

DELIMITER ;

CREATE VIEW V\_Animal\_details AS

select Animal.id,Animal.sexe,Animal.date\_naissance,Animal.nom,Animal.commentaires,Animal.espece\_id,Animal.race\_id,

Animal.mere\_id,Animal.pere\_id,Animal.disponible,Espece.nom\_courant AS espece\_nom,Race.nom AS race\_nom

from Animal

join Espece on Animal.espece\_id = Espece.id

left join Race on Animal.race\_id = Race.id;

CREATE VIEW V\_Animal\_espece AS

select Animal.id,Animal.sexe,Animal.date\_naissance,Animal.nom,Animal.commentaires,Animal.espece\_id,Animal.race\_id,

Animal.mere\_id,Animal.pere\_id,Animal.disponible,Espece.nom\_courant AS espece\_nom,Espece.nom\_latin AS espece\_nom\_latin

from Animal

join Espece on Espece.id = Animal.espece\_id;

CREATE VIEW V\_Animal\_stagiaire AS

select Animal.id,Animal.nom,Animal.sexe,Animal.date\_naissance,Animal.espece\_id,Animal.race\_id,Animal.mere\_id,Animal.pere\_id,Animal.disponible

from Animal

where Animal.espece\_id = 2

WITH CASCADED CHECK OPTION;

CREATE VIEW V\_Chien AS

select Animal.id,Animal.sexe,Animal.date\_naissance,Animal.nom,Animal.commentaires,Animal.espece\_id,Animal.race\_id,Animal.mere\_id,Animal.pere\_id,Animal.disponible

from Animal

where Animal.espece\_id = 1;

CREATE VIEW V\_Chien\_race AS

select V\_Chien.id,V\_Chien.sexe,V\_Chien.date\_naissance,V\_Chien.nom,V\_Chien.commentaires,V\_Chien.espece\_id,V\_Chien.race\_id,V\_Chien.mere\_id,V\_Chien.pere\_id,V\_Chien.disponible

from V\_Chien

where V\_Chien.race\_id is not null

WITH CASCADED CHECK OPTION;

CREATE VIEW V\_Client AS

select Client.id,Client.nom,Client.prenom,Client.adresse,Client.code\_postal,Client.ville,Client.pays,Client.email

from Client;

CREATE VIEW V\_Espece\_dollars AS

select Espece.id,Espece.nom\_courant,Espece.nom\_latin,Espece.description,round((Espece.prix \* 1.30813),2) AS prix\_dollars

from Espece;

CREATE VIEW V\_Nombre\_espece AS

select Espece.id,count(Animal.id) AS nb

from Espece

left join Animal on Animal.espece\_id = Espece.id

group by Espece.id;

CREATE VIEW V\_Revenus\_annee\_espece AS

select year(Adoption.date\_reservation) AS annee,Espece.id AS espece\_id,sum(Adoption.prix) AS somme,count(Adoption.animal\_id) AS nb

from Adoption

join Animal on Animal.id = Adoption.animal\_id

join Espece on Animal.espece\_id = Espece.id

group by year(Adoption.date\_reservation),Espece.id;

### Introduction

Pendant ce cours, nous avons créé une base de données : elevage. Vous avez peut-être créé également d'autres bases de données pour vos tests et projets personnels.

Mais ce ne sont pas les seules bases de données existant sur votre serveur MySQL. Connectez-vous avec l'utilisateur "root" (sinon certaines bases de données vous seront cachées) et exécutez la requête suivante :

SHOW DATABASES;

La base elevage est bien là, en compagnie de quelques autres :

* information\_schema : cette base de données stocke les informations sur toutes les bases de données. Les tables, les colonnes, le type des colonnes, les procédures des bases de données y sont recensés, avec leurs caractéristiques. Nous verrons cette base de données plus en détail dans le prochain chapitre.
* performance\_schema : permet de stocker des informations sur les actions effectuées sur le serveur (temps d'exécution, temps d'attente dus aux verrous, etc.)
* test : il s'agit d'une base de test automatiquement créée. Si vous ne l'avez pas utilisée, elle ne contient rien.
* mysql : contient de nombreuses informations sur le serveur. Entre autres, c'est dans cette base que sont stockés **les utilisateurs et leurs privilèges**.

#### Les utilisateurs et leurs privilèges

##### Privilèges des utilisateurs

Lorsque l'on se connecte à MySQL, on le fait avec un utilisateur. Chaque utilisateur possède une série de privilèges relatifs aux données stockées sur le serveur : le privilège de sélectionner les données, de les modifier, de créer des objets, etc.  
Ces privilèges peuvent exister à plusieurs niveaux : base de données, tables, colonnes, procédures, etc.

Par exemple, tout au début de ce cours, vous avez créé un utilisateur, puis lui avez donné des privilèges avec la commande suivante :

GRANT ALL PRIVILEGES ON elevage.\* TO 'student'@'localhost';

Cette requête a donné tous les droits sur la base de données elevage à l'utilisateur 'student'@'localhost'.

##### Stockage des utilisateurs et privilèges

Toutes ces informations sont stockées dans la base de données mysql.  
Les utilisateurs sont stockés dans la table user, avec leurs privilèges globaux (c'est-à-dire valables au niveau du serveur, sur toutes les bases de données). La base mysql possède par ailleurs quatre tables permettant de stocker les privilèges des utilisateurs à différents niveaux.

* db : privilèges au niveau des bases de données.
* tables\_priv : privilèges au niveau des tables.
* columns\_priv : privilèges au niveau des colonnes.
* proc\_priv : privilèges au niveau des routines (procédures et fonctions stockées).

##### Modifications

Il est tout à fait possible d'ajouter, de modifier et de supprimer des utilisateurs en utilisant des requêtes INSERT, UPDATE  ou DELETE  directement sur la table mysql.user. De même pour leurs privilèges, avec les tables mysql.db, mysql.tables\_priv, mysql.columns\_priv et mysql.procs\_priv.

Cependant, en général, on utilise plutôt des commandes dédiées à cet usage. Ainsi, pas besoin de se soucier de la structure de ces tables, et le risque d'erreur est moins grand. Ce sont ces commandes que nous allons décortiquer dans la suite de ce chapitre.

Tout comme on peut préciser la table à laquelle appartient une colonne en préfixant le nom de la colonne par le nom de la table, nom\_table.nom\_colonne, il est possible de préciser à quelle base de données appartient une table, nom\_bdd.nom\_table, voire de préciser à la fois la table et la base de données dans laquelle se trouve une colonne : nom\_bdd.nom\_table.nom\_colonne.

### Création, modification et suppression des utilisateurs

#### Syntaxe pour la création et la suppression

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

-- Création

CREATE USER 'login'@'hote' [IDENTIFIED BY 'mot\_de\_passe'];

-- Suppression

DROP USER 'login'@'hote';

#### Utilisateur

L'utilisateur est donc défini par deux éléments :

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

##### Login

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

##### Hôte

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

**Exemples**

CREATE USER 'max'@'localhost' IDENTIFIED BY 'maxisthebest';

CREATE USER 'elodie'@'194.28.12.4' IDENTIFIED BY 'ginko1';

CREATE USER 'gabriel'@'arb.brab.net' IDENTIFIED BY 'chinypower';

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

**Exemples**

-- thibault peut se connecter à partir de n'importe quel hôte dont l'adresse IP commence par 194.28.12.

CREATE USER 'thibault'@'194.28.12.%' IDENTIFIED BY 'basketball8';

-- joelle peut se connecter à partir de n'importe quel hôte du domaine brab.net

CREATE USER 'joelle'@'%.brab.net' IDENTIFIED BY 'singingisfun';

-- hannah peut se connecter à partir de n'importe quel hôte

CREATE USER 'hannah'@'%' IDENTIFIED BY 'looking4sun';

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

Les guillemets se placent indépendamment autour du login **et** autour de l'hôte. N'entourez pas tout par des guillemets : CREATE USER 'marie@localhost'  créera un utilisateur dont le login est 'marie@localhost', autorisé à se connecter à partir de n'importe quel hôte.

##### Renommer l'utilisateur

Pour modifier l'identifiant d'un utilisateur (login et/ou hôte), on peut utiliser RENAME USER ancien\_utilisateur TO nouvel\_utilisateur.

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

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

#### Mot de passe

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

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

##### Modifier le mot de passe

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

**Exemple**

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

### Les privilèges - introduction

Lorsque l'on crée un utilisateur avec CREATE USER, celui-ci n'a au départ aucun privilège, aucun droit.  
En SQL, avoir un privilège, c'est avoir l'autorisation d'effectuer une action sur un objet.

Un utilisateur sans aucun privilège ne peut rien faire d'autre que se connecter. Il n'aura pas accès aux données, ne pourra créer aucun objet (base/table/procédure/autre) ni en utiliser.

#### Les différents privilèges

Il existe de nombreux privilèges dont voici une sélection des plus utilisés (à l'exception des privilèges particuliers ALL, USAGE  et GRANT OPTION  que nous verrons plus loin).

##### Privilèges du CRUD

Les privilèges SELECT, INSERT, UPDATE  et DELETE  permettent aux utilisateurs d'utiliser ces mêmes commandes.

##### Privilèges concernant les tables, les vues et les bases de données

| **Privilège** | **Action autorisée** |
| --- | --- |
| CREATE TABLE | Création de tables |
| CREATE TEMPORARY TABLE | Création de tables temporaires |
| CREATE VIEW | Création de vues (il faut également avoir le privilège SELECTsur les colonnes sélectionnées par la vue) |
| ALTER | Modification de tables (avec ALTER TABLE) |
| DROP | Suppression de tables, vues et bases de données |

##### Autres privilèges

| **Privilège** | **Action autorisée** |
| --- | --- |
| CREATE ROUTINE | Création de procédures stockées (et de fonctions stockées - non couvertes dans ce cours) |
| ALTER ROUTINE | Modification et suppression de procédures stockées (et de fonctions stockées) |
| EXECUTE | Exécution de procédures stockées (et de fonctions stockées) |
| INDEX | Création et suppression d'index |
| TRIGGER | Création et suppression de triggers |
| LOCK TABLES | Verrouillage de tables (sur lesquelles on a le privilège SELECT) |
| CREATE USER | Gestion d'utilisateur (commandes CREATE USER, DROP USER, RENAME USER  et SET PASSWORD) |

#### Les différents niveaux d'application des privilèges

Lorsque l'on accorde un privilège à un utilisateur, il faut également préciser à quoi s'applique ce privilège.

| **Niveau** | **Application du privilège** |
| --- | --- |
| \*.\* | Privilège global : s'applique à toutes les bases de données, à tous les objets. Un privilège de ce niveau sera stocké dans la table mysql.user. |
| \* | Si aucune base de données n'a été préalablement sélectionnée (avec USE nom\_bdd), c'est l'équivalent de \*.\* (privilège stocké dans mysql.user). Sinon, le privilège s'appliquera à tous les objets de la base de données que l'on utilise (et sera stocké dans la table mysql.db). |
| nom\_bdd.\* | Privilège de base de données : s'applique à tous les objets de la base nom\_bdd (stocké dans mysql.db). |
| nom\_bdd.nom\_table | Privilège de table (stocké dans mysql.tables\_priv). |
| nom\_table | Privilège de table : s'applique à la table nom\_table de la base de données dans laquelle on se trouve, sélectionnée au préalable avec USE nom\_bdd  (stocké dans mysql.tables\_priv). |
| nom\_bdd.nom\_routine | S'applique à la procédure (ou fonction) stockée nom\_bdd.nom\_routine (privilège stocké dans mysql.procs\_priv). |

Les privilèges peuvent aussi être restreints à certaines colonnes, auquel cas ils seront stockés dans la table mysql.columns\_priv. Nous verrons comment restreindre un privilège à certaines colonnes avec la commande GRANT.

### Ajout et révocation de privilèges

#### Ajout de privilèges

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

##### Syntaxe

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

GRANT privilege [(liste\_colonnes)] [, privilege [(liste\_colonnes)], ...]

ON [type\_objet] niveau\_privilege

TO utilisateur [IDENTIFIED BY mot\_de\_passe];

* privilege  : le privilège à accorder à l'utilisateur (SELECT, CREATE VIEW, EXECUTE…) ;
* (liste\_colonnes)  : facultatif - liste des colonnes auxquelles le privilège s'applique ;
* niveau\_privilege  : niveau auquel le privilège s'applique (\*.\*, nom\_bdd.nom\_table…) ;
* type\_objet  : en cas de noms ambigus, il est possible de préciser à quoi se rapporte le niveau, TABLE  ou PROCEDURE.

On peut accorder plusieurs privilèges en une fois : il suffit de séparer les privilèges par une virgule. Si l'on veut restreindre tout ou partie des privilèges à certaines colonnes, la liste doit en être précisée pour chaque privilège.

**Exemples**

**1.** On crée un utilisateur 'john'@'localhost', en lui donnant les privilèges SELECT, INSERT  et DELETE  sur la table elevage.Animal, et UPDATE  sur les colonnes nom, sexe et commentaires de la table elevage.Animal.

CREATE USER 'john'@'localhost' IDENTIFIED BY 'exemple2012';

GRANT SELECT,

UPDATE (nom, sexe, commentaires),

DELETE,

INSERT

ON elevage.Animal

TO 'john'@'localhost';

**2.** On accorde le privilège SELECT  à l'utilisateur 'john'@'localhost'  sur la table elevage.Espece.

GRANT SELECT

ON TABLE elevage.Espece -- On précise que c'est une table (facultatif)

TO 'john'@'localhost';

**3.** On accorde à 'john'@'localhost'  le privilège de créer et d'exécuter des procédures stockées dans la base de données elevage.

GRANT CREATE ROUTINE, EXECUTE

ON elevage.\*

TO 'john'@'localhost';

#### Révocation de privilèges

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

REVOKE privilege [, privilege, ...]

ON niveau\_privilege

FROM utilisateur;

**Exemple**

REVOKE DELETE

ON elevage.Animal

FROM 'john'@'localhost';

### Privilèges particuliers

#### Les privilèges ALL, USAGE et GRANT OPTION

Pour terminer avec les différents privilèges, nous allons parler de trois privilèges un peu particuliers.

##### ALL

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

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

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

GRANT ALL

ON elevage.Client

TO 'john'@'localhost';

##### USAGE

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

##### GRANT OPTION

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

On peut accorder GRANT OPTION  de deux manières :

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

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

GRANT SELECT, UPDATE, INSERT, DELETE, GRANT OPTION

ON elevage.\*

TO 'joseph'@'localhost' IDENTIFIED BY 'ploc4';

-- OU

GRANT SELECT, UPDATE, INSERT, DELETE

ON elevage.\*

TO 'joseph'@'localhost' IDENTIFIED BY 'ploc4'

WITH GRANT OPTION;

Le privilège ALL  doit s'utiliser tout seul. Il n'est donc pas possible d'accorder GRANT OPTION  et ALL  de la manière suivante : GRANT ALL, GRANT OPTION…  Il est nécessaire dans ce cas d'utiliser WITH GRANT OPTION.

#### Particularité des triggers, vues et procédures stockées

Les triggers, les vues et les procédures stockées (ainsi que les événements et les fonctions stockées, non couvertes par ce cours) ont un système spécial quant à la vérification des privilèges des utilisateurs.  
En effet, ces objets sont créés dans le but d'être exécutés dans le futur, et l'utilisateur créant un tel objet pourrait bien être différent de l'utilisateur se servant de l'objet. Il y a donc deux types d'utilisateurs potentiels pour ces types d'objet : celui ayant défini l'objet, et celui utilisant l'objet. Quels privilèges faut-il vérifier lorsqu'une procédure est exécutée ? Lorsque la requête SELECT  d'une vue est exécutée ?

Par défaut, ce sont les privilèges du définisseur (celui qui a défini l'objet) qui sont vérifiés. Ce qui veut dire qu'un utilisateur pourrait exécuter une procédure agissant sur des tables sur lesquelles il n'a lui-même aucun privilège, l'important étant les privilèges de l'utilisateur ayant défini la procédure. Bien entendu, un utilisateur doit toujours avoir le privilège EXECUTE  pour exécuter une procédure stockée. Les privilèges du définisseur concernent les instructions à l'intérieur de la procédure ou du trigger (ou de la requête SELECT  pour une vue).

**Exemple :** avec l'utilisateur student, on définit une procédure faisant une requête SELECT  sur la table Adoption. On exécute ensuite cette procédure avec l'utilisateur john, qui n'a aucun droit sur la table Adoption.

Utilisateur student :

USE elevage;

DELIMITER |

CREATE PROCEDURE test\_definer()

BEGIN

SELECT \* FROM Adoption;

END |

DELIMITER ;

Utilisateur john :

USE elevage;

SELECT \* FROM Adoption;

CALL test\_definer();

L'utilisateur john n'a aucun droit sur Adoption. La requête SELECT  échoue donc avec le message d'erreur suivant :

ERROR 1142 (42000): SELECT command denied to user 'john'@'localhost' for table 'adoption'

Par contre, john a le droit d'exécuter les procédures de la base de données elevage. Il exécute donc sans problème test\_definer(), qui lui affiche le contenu d'Adoption. Les privilèges vérifiés au niveau des instructions exécutées par la procédure sont en effet ceux de l'utilisateur student ayant défini celle-ci.

##### Préciser et modifier le définisseur

Les commandes de création des vues, triggers et procédures stockées permettent de préciser une clause DEFINER, dans laquelle on précise l'identifiant d'un utilisateur.  
Par défaut, c'est l'utilisateur courant qui est utilisé.  
Cependant, il est possible de donner un autre utilisateur comme définisseur de l'objet, à condition d'avoir le privilège global SUPER. Sans ce privilège, on ne peut donner comme DEFINER  que soi-même, soit avec CURRENT\_USER  ou CURRENT\_USER(), soit avec l'identifiant ('student'@'localhost', par exemple).  
La clause DEFINER  se place après le mot-clé CREATE.

**Exemple :** définition de deux procédures stockées avec l'utilisateur root (le seul ayant le privilège SUPER  sur notre serveur), l'une avec root pour DEFINER  (CURRENT\_USER()), l'autre avec john.

DELIMITER |

CREATE DEFINER = CURRENT\_USER() PROCEDURE test\_definer2()

BEGIN

SELECT \* FROM Race;

END |

CREATE DEFINER = 'john'@'localhost' PROCEDURE test\_definer3()

BEGIN

SELECT \* FROM Race;

END |

DELIMITER ;

Si l'on exécute ces deux procédures avec l'utilisateur john, on obtient deux résultats différents.

CALL test\_definer2();

CALL test\_definer3();

La première procédure s'exécute sans problème. john n'a aucun droit sur la table Race, mais le définisseur de test\_definer2() étant root, ce sont ses privilèges qui comptent. Par contre, la seconde échoue, car le définisseur de test\_definer3() a été initialisé à john.

##### Modification du contexte

Il est possible, pour les vues et les procédures stockées, de changer la manière dont les privilèges sont vérifiés à l'exécution. On peut faire en sorte que ce soient les privilèges de l'utilisateur qui se sert de la vue ou de la procédure ("l'invocateur") qui soient vérifiés, et non plus les privilèges du définisseur.

Ce n'est pas possible pour les triggers, car ceux-ci ne sont pas exécutés directement par un utilisateur, mais par une action de l'utilisateur (insertion, modification, suppression dans la table sur laquelle le trigger est défini).

Pour changer le contexte de vérification des privilèges d'une vue ou d'une procédure, il faut utiliser la clause SQL SECURITY {DEFINER | INVOKER}.

**Syntaxe de création des vues et des procédures**, clauses DEFINER  et SQL SECURITY  comprises.

-- Vues

CREATE [OR REPLACE]

[ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]

[DEFINER = { utilisateur | CURRENT\_USER }]

[SQL SECURITY { DEFINER | INVOKER }]

VIEW nom\_vue [(liste\_colonnes)]

AS requete\_select

[WITH [CASCADED | LOCAL] CHECK OPTION]

-- Procédures

CREATE

[DEFINER = { utilisateur | CURRENT\_USER }]

PROCEDURE nom\_procedure ([parametres\_procedure])

SQL SECURITY { DEFINER | INVOKER }

corps\_procedure

**Exemple :** création, par l'utilisateur root, de deux vues avec des contextes de vérification des privilèges différents.

CREATE DEFINER = CURRENT\_USER

SQL SECURITY DEFINER

VIEW test\_contexte1

AS SELECT \* FROM Race;

CREATE DEFINER = CURRENT\_USER

SQL SECURITY INVOKER

VIEW test\_contexte2

AS SELECT \* FROM Race;

Toujours avec l'utilisateur root, on autorise john à faire des SELECT  sur ces vues.

GRANT SELECT ON elevage.test\_contexte1 TO 'john'@'localhost';

GRANT SELECT ON elevage.test\_contexte2 TO 'john'@'localhost';

Utilisons maintenant ces vues avec l'utilisateur john, qui n'a toujours aucun droit sur Race.

SELECT \* FROM test\_contexte1;

SELECT \* FROM test\_contexte2;

La première requête affiche bien la table Race. Par contre, la seconde échoue avec l'erreur suivante :

ERROR 1356 (HY000): View 'elevage.test\_contexte2' references invalid table(s) or column(s) or function(s) or definer/invoker of view lack rights to use them

### Options supplémentaires

La commande GRANT  possède encore deux clauses facultatives supplémentaires permettant de limiter les ressources serveur de l'utilisateur, et d'obliger l'utilisateur à se connecter via SSL.

#### Limitation des ressources

On peut limiter trois choses différentes pour un utilisateur :

* le nombre de requêtes par heure (MAX\_QUERIES\_PER\_HOUR) : limitation de toutes les commandes exécutées par l'utilisateur ;
* le nombre de modifications par heure (MAX\_UPDATES\_PER\_HOUR) : limitation des commandes entraînant la modification d'une table ou d'une base de données ;
* le nombre de connexions au serveur par heure (MAX\_CONNECTIONS\_PER\_HOUR).

Pour cela, on utilise la clause WITH MAX\_QUERIES\_PER\_HOUR nb | MAX\_UPDATES\_PER\_HOUR nb | MAX\_CONNECTIONS\_PER\_HOUR nbde la commande GRANT. On peut limiter une des ressources, ou deux, ou les trois en une fois, chacune avec un nombre différent.

**Exemple :** création d'un compte 'aline'@'localhost'  ayant tous les droits sur la base de données elevage, mais avec des ressources limitées.

CREATE USER 'aline'@'localhost' IDENTIFIED BY 'limited';

GRANT ALL ON elevage.\*

TO 'aline'@'localhost'

WITH MAX\_QUERIES\_PER\_HOUR 50

MAX\_CONNECTIONS\_PER\_HOUR 5;

#### Connexion SSL

La clause REQUIRE  de la commande GRANT  permet d'obliger l'utilisateur à se connecter via SSL, c'est-à-dire à l'aide d'une connexion sécurisée. Avec une telle connexion, toutes les données transitant entre le client et le serveur sont chiffrées, et non plus passées en clair.

Nous ne verrons pas les détails de cette clause dans ce cours. Je vous renvoie donc à la [documentation](https://dev.mysql.com/doc/refman/5.5/en/secure-connections.html) si vous êtes intéressé par le sujet.

## Informations sur la base de données et les requêtes

Dans ce chapitre, vous verrez comment aller chercher des informations sur les différents objets de vos bases de données (les tables, les procédures, etc.) de deux manières différentes :

* grâce à la commande SHOW  ;
* en allant chercher ces informations directement dans la base de données information\_schema.

Ensuite, vous découvrirez la commande EXPLAIN, qui donne des indications sur le déroulement des requêtes, ce qui permet d'optimiser celles-ci.

### Commandes de description

Les commandes SHOW  et DESCRIBE  ont été utilisées ponctuellement tout au long du cours pour afficher diverses informations sur les tables et les colonnes. Nous allons ici voir plus en détail comment utiliser ces commandes, et ce qu'elles peuvent nous apprendre.

#### Description d'objets

SHOW objets;

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

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

SHOW TABLES;

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

##### Objets listables avec SHOW

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

| **Commande** | **Description** |
| --- | --- |
| SHOW CHARACTER SET | Montre les sets de caractères (encodages) disponibles. |
| SHOW [FULL] COLUMNS FROM nom\_table [FROM nom\_bdd] | Liste les colonnes de la table précisée, ainsi que diverses informations (type, contraintes…). Il est possible de préciser également le nom de la base de données. En ajoutant le mot-clé FULL, les informations affichées pour chaque colonne sont plus nombreuses. |
| SHOW DATABASES | Montre les bases de données sur lesquelles on possède des privilèges (ou toutes si l'on possède le privilège global SHOW DATABASES). |
| SHOW GRANTS [FOR utilisateur] | Liste les privilèges de l'utilisateur courant, ou de l'utilisateur précisé par la clause FOR  optionnelle. |
| SHOW INDEX FROM nom\_table [FROM nom\_bdd] | Liste les index de la table désignée. Il est possible de préciser également le nom de la base de données. |
| SHOW PRIVILEGES | Liste les privilèges acceptés par le serveur MySQL (dépend de la version de MySQL). |
| SHOW PROCEDURE STATUS | Liste les procédures stockées. |
| SHOW [FULL] TABLES [FROM nom\_bdd] | Liste les tables de la base de données courante, ou de la base de données désignée par la clause FROM. Si FULL  est utilisé, une colonne apparaîtra en plus, précisant s'il s'agit d'une vraie table ou d'une vue. |
| SHOW TRIGGERS [FROM nom\_bdd] | Liste les triggers de la base de données courante, ou de la base de données précisée grâce à la clause FROM. |
| SHOW [GLOBAL | SESSION] VARIABLES | Liste les variables système de MySQL. Si GLOBAL  est précisé, les valeurs des variables seront celles utilisées lors d'une nouvelle connexion au serveur. Si SESSION  est utilisé (ou si l'on ne précise ni GLOBAL  ni SESSION), les valeurs seront celles de la session courante. Plus d'informations sur les variables système seront données dans le prochain chapitre. |
| SHOW WARNINGS | Liste les avertissements générés par la dernière requête effectuée. |

##### Clauses additionnelles

Certaines commandes SHOW objets  acceptent des clauses supplémentaires : LIKE  et WHERE.

* La clause LIKE  permet de restreindre la liste aux objets dont le nom correspond au motif donné.
* WHERE  permet d'ajouter diverses conditions.

**Exemple 1 :** sélection des colonnes d'Adoption dont le nom commence par "date".

SHOW COLUMNS

FROM Adoption

LIKE 'date%';

**Exemple 2 :** sélection des encodages contenant "arab" dans leur description.

SHOW CHARACTER *SET*

WHERE Description LIKE '%arab%';

##### DESCRIBE

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

#### Requête de création d'un objet

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

SHOW CREATE type\_objet nom\_objet;

**Exemple 1 :** requête de création de la table Espece.

SHOW CREATE TABLE Espece \G

Le \G est un délimiteur, comme ;. Il change simplement la manière d'afficher le résultat, qui ne sera plus sous forme de tableau, mais formaté verticalement. Pour les requêtes de description comme SHOW CREATE, qui renvoient peu de lignes (ici : une), mais contenant beaucoup d'informations, c'est beaucoup plus lisible.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Table: Espece

Create Table: CREATE TABLE `Espece` (

`id` smallint(6) unsigned NOT NULL AUTO\_INCREMENT,

`nom\_courant` *varchar*(40) NOT NULL,

`nom\_latin` *varchar*(40) NOT NULL,

`description` *text*,

`prix` *decimal*(7,2) unsigned DEFAULT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY `nom\_latin` (`nom\_latin`)

) ENGINE=InnoDB AUTO\_INCREMENT=8 DEFAULT CHARSET=utf8

**Exemple 2 :** requête de création du trigger before\_insert\_adoption.

SHOW CREATE TRIGGER before\_insert\_adoption \G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Trigger: before\_insert\_adoption

sql\_mode:

SQL Original Statement: CREATE DEFINER=`student`@`localhost` TRIGGER before\_insert\_adoption BEFORE INSERT

ON Adoption FOR EACH ROW

BEGIN

IF NEW.paye != TRUE

AND NEW.paye != FALSE

THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : paye doit valoir TRUE (1) ou FALSE (0).');

ELSEIF NEW.date\_adoption < NEW.date\_reservation THEN

INSERT INTO Erreur (erreur) VALUES ('Erreur : date\_adoption doit être >= à date\_reservation.');

END IF;

END

character\_set\_client: utf8

collation\_connection: utf8\_general\_ci

Database Collation: utf8\_general\_ci

On peut ainsi afficher la syntaxe de création d'une table, d'une base de données, d'une procédure, d'un trigger ou d'une vue.

### La base de données information\_schema

Comme son nom l'indique, la base de données information\_schema contient des informations sur les schémas.  
En MySQL, un schéma est une base de données. Ce sont des synonymes. La base information\_schema contient donc des **informations sur les bases de données**.

Cette définition de "schéma" n'est pas universelle, tant s'en faut. Dans certains SGBD, la notion de schéma est plus proche de celle d'utilisateur que de celle de base de données. Pour Oracle, par exemple, un schéma représente l'ensemble des objets appartenant à un utilisateur.

Voyons ce que l'on trouve comme tables dans cette base de données.

SHOW TABLES FROM information\_schema;

| **Tables\_in\_information\_schema** |
| --- |
| CHARACTER\_SETS |
| COLUMNS |
| COLUMN\_PRIVILEGES |
| REFERENTIAL\_CONSTRAINTS |
| ROUTINES |
| SESSION\_VARIABLES |
| STATISTICS |
| TABLES |
| TABLE\_CONSTRAINTS |
| TABLE\_PRIVILEGES |
| TRIGGERS |
| USER\_PRIVILEGES |
| VIEWS |

Le tableau ci-dessus ne reprend qu'une partie des tables d'information\_schema.

Cette base contient donc des informations sur les tables, les colonnes, les contraintes, les vues, etc., des bases de données stockées sur le serveur MySQL.  
En fait, c'est de cette base de données que sont extraites les informations affichées grâce à la commande SHOW.

Par conséquent, si les informations données par SHOW  ne suffisent pas, il est possible d'interroger directement cette base de données.

Prenons par exemple la table VIEWS de cette base de données. Quelles informations contient-elle ?

SHOW COLUMNS FROM VIEWS FROM information\_schema;

La colonne *TABLE\_NAME* contient le nom de la vue. Interrogeons donc cette table, afin d'avoir des informations sur la vue *V\_Animal\_details*.

USE information\_schema; -- On sélectionne la base de données

SELECT TABLE\_SCHEMA, TABLE\_NAME, VIEW\_DEFINITION, IS\_UPDATABLE, DEFINER, SECURITY\_TYPE

FROM VIEWS

WHERE TABLE\_NAME = 'V\_Animal\_details' \G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TABLE\_SCHEMA: elevage

TABLE\_NAME: V\_Animal\_details

VIEW\_DEFINITION: select `elevage`.`animal`.`id` AS `id`,`elevage`.`animal`.`sexe` AS `sexe`,`elevage`.`animal`.`date\_naissance` AS `date\_naissance`,

`elevage`.`animal`.`nom` AS `nom`,`elevage`.`animal`.`commentaires` AS `commentaires`,

`elevage`.`animal`.`espece\_id` AS `espece\_id`,`elevage`.`animal`.`race\_id` AS `race\_id`,

`elevage`.`animal`.`mere\_id` AS `mere\_id`,`elevage`.`animal`.`pere\_id` AS `pere\_id`,

`elevage`.`animal`.`disponible` AS `disponible`,`elevage`.`espece`.`nom\_courant` AS `espece\_nom`,

`elevage`.`race`.`nom` AS `race\_nom`

from ((`elevage`.`animal`

join `elevage`.`espece` on((`elevage`.`animal`.`espece\_id` = `elevage`.`espece`.`id`)))

left join `elevage`.`race` on((`elevage`.`animal`.`race\_id` = `elevage`.`race`.`id`)))

IS\_UPDATABLE: YES

DEFINER: student@localhost

SECURITY\_TYPE: DEFINER

La définition de la vue s'affiche en réalité sur une seule ligne, et n'est donc pas indentée. J'ai ajouté l'indentation ici pour que ce soit plus clair.

Voyons encore deux exemples d'exploitation des données d'*information\_schema*.

**Exemple 1 :** données sur les contraintes de la table *Animal*.

SELECT CONSTRAINT\_SCHEMA, CONSTRAINT\_NAME, TABLE\_NAME, CONSTRAINT\_TYPE

FROM TABLE\_CONSTRAINTS

WHERE CONSTRAINT\_SCHEMA = 'elevage' AND TABLE\_NAME = 'Animal';

**Exemple 2 :** données sur la procédure maj\_vm\_revenus().

SELECT ROUTINE\_NAME, ROUTINE\_SCHEMA, ROUTINE\_TYPE, ROUTINE\_DEFINITION, DEFINER, SECURITY\_TYPE

FROM ROUTINES

WHERE ROUTINE\_NAME = 'maj\_vm\_revenus' \G

Les routines comprennent les procédures stockées et les fonctions stockées (qui ne sont pas couvertes par ce cours).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ROUTINE\_NAME: maj\_vm\_revenus

ROUTINE\_SCHEMA: elevage

ROUTINE\_TYPE: PROCEDURE

ROUTINE\_DEFINITION: BEGIN

TRUNCATE VM\_Revenus\_annee\_espece;

INSERT INTO VM\_Revenus\_annee\_espece

SELECT YEAR(date\_reservation) AS annee, Espece.id AS espece\_id, SUM(Adoption.prix) AS somme, COUNT(Adoption.animal\_id) AS nb

FROM Adoption

INNER JOIN Animal ON Animal.id = Adoption.animal\_id

INNER JOIN Espece ON Animal.espece\_id = Espece.id

GROUP BY annee, Espece.id;

END

DEFINER: student@localhost

SECURITY\_TYPE: DEFINER

### Déroulement d'une requête

On a vu comment obtenir des informations sur les objets de nos bases de données. Voyons maintenant comment obtenir des **informations sur les requêtes que l'on exécute** sur nos bases de données.

Plus particulièrement, nous allons nous attarder sur la commande EXPLAIN, qui permet de décortiquer l'exécution d'une requête. Grâce à cette commande, il est possible de savoir quelles tables et quels index sont utilisés, et dans quel ordre.

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

**Exemple**

EXPLAIN SELECT Animal.nom, Espece.nom\_courant AS espece, Race.nom AS race

FROM Animal

INNER JOIN Espece ON Animal.espece\_id = Espece.id

LEFT JOIN Race ON Animal.race\_id = Race.id

WHERE Animal.id = 37;

**Exemple**

EXPLAIN SELECT Animal.nom, Adoption.prix, Adoption.date\_reservation

FROM Animal

INNER JOIN Adoption ON Adoption.animal\_id = Animal.id

WHERE date\_reservation >= '2012-05-01' \G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: Adoption

type: ALL

possible\_keys: ind\_uni\_animal\_id

key: NULL

key\_len: NULL

ref: NULL

rows: 24

Extra: Using where

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: Animal

type: eq\_ref

possible\_keys: PRIMARY

key: PRIMARY

key\_len: 2

ref: elevage7.Adoption.animal\_id

rows: 1

Extra:

Pour la première étape, key, key\_len et ref sont NULL. Aucun index n'est donc utilisé. type vaut  ALL, ce qui est la valeur la moins intéressante. Enfin, rows vaut 24, ce qui est le nombre de lignes dans la table Adoption. Toutes les lignes doivent être parcourues pour trouver les lignes correspondant à la clause WHERE.  
La deuxième étape par contre utilise bien un index (pour faire la jointure avec Animal).

En ajoutant un index sur Adoption.date\_reservation, on peut améliorer les performances de cette requête.

ALTER TABLE Adoption ADD INDEX ind\_date\_reservation (date\_reservation);

La même commande EXPLAIN  donnera désormais le résultat suivant :

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: Adoption

type: range

possible\_keys: ind\_uni\_animal\_id,ind\_date\_reservation

key: ind\_date\_reservation

key\_len: 3

ref: NULL

rows: 4

Extra: Using where

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: Animal

type: eq\_ref

possible\_keys: PRIMARY

key: PRIMARY

key\_len: 2

ref: elevage7.Adoption.animal\_id

rows: 1

Extra:

La première étape utilise l'index, et ne doit donc plus parcourir toutes les lignes, mais seulement une partie (dont le nombre est estimé à 4).

Attention de ne pas tomber dans l'excès en mettant des index partout. En général, on utiliseEXPLAIN  sur des requêtes lourdes, dont on sait qu'elles ralentissent l'application. Il n'est pas nécessaire d'optimiser la moindre petite requête.

##### Comparer le plan d'exécution de plusieurs requêtes

Lorsque l'on fait une condition sur une colonne (dans une clause WHERE  ou pour une condition de jointure), ce n'est pas parce qu'il existe un index sur celle-ci qu'il sera utilisé par la requête. En particulier, si la colonne est utilisée dans une expression, l'index ne sera pas utilisé, car la valeur de l'expression devra être calculée pour chaque ligne.  
Selon la manière d'écrire une condition donc, l'utilisation des index sera possible, ou non. Lorsque l'on hésite entre plusieurs écritures, utiliser EXPLAIN  peut permettre d'utiliser la requête la plus performante.

**Exemple**

EXPLAIN SELECT \*

FROM VM\_Revenus\_annee\_espece

WHERE somme/2 > 1000 \G

EXPLAIN SELECT \*

FROM VM\_Revenus\_annee\_espece

WHERE somme > 1000\*2 \G

Ces deux requêtes SELECT  produisent un résultat équivalent, mais la première empêche l'utilisation de l'index sur somme, contrairement à la deuxième.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: VM\_Revenus\_annee\_espece

type: ALL

possible\_keys: NULL

key: NULL

key\_len: NULL

ref: NULL

rows: 16

Extra: Using where

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: VM\_Revenus\_annee\_espece

type: range

possible\_keys: somme

key: somme

key\_len: 14

ref: NULL

rows: 2

Extra: Using where

## Configuration et options

Dans ce dernier chapitre, vous apprendrez à configurer MySQL :

* en modifiant les variables système avec la commande SET  ;
* en utilisant les options du logiciel client au démarrage d'une session ;
* en utilisant les options du serveur lors de son démarrage ;
* en modifiant le fichier de configuration de MySQL.

Nous commencerons bien sûr par voir ce qu'est exactement une variable système.

Notez que ces variables système sont nombreuses, tout comme les options disponibles au niveau du client et du serveur, ainsi que dans le fichier de configuration.  
Pour ne pas vous donner de longues listes indigestes, je vous renverrai très souvent vers la documentation officielle au cours de ce chapitre, et ne vous donnerai que quelques exemples parmi les plus utilisés.

### Variables système

Le comportement de MySQL est régi par une série de valeurs contenues dans les**variables système**. Elles déterminent par exemple à quelle valeur commencent les colonnes auto-incrémentées (par défaut : 1), de combien ces valeurs s'incrémentent à chaque insertion (par défaut : 1 également), le moteur de stockage par défaut, le temps d'attente acceptable lorsqu'une requête se heurte à un verrou, etc.

Voici quelques variables système. Vous verrez que vous en connaissez déjà quelques-unes.  
Pour une liste complète, reportez-vous à la documentation officielle.

| **Nom** | **Définition** |
| --- | --- |
| autocommit | Définit si le mode autocommit est activé ou non, donc si les requêtes sont automatiquement commitées ou s'il est nécessaire de les commiter pour qu'elles prennent effet. |
| character\_set\_client | Jeu de caractères (encodage) utilisé par le client MySQL. |
| default\_week\_format | Mode par défaut de la fonction WEEK(). |
| foreign\_key\_checks | Définit si les contraintes de clé étrangère doivent être vérifiées. |
| ft\_min\_word\_len | Taille minimale d'un mot pour qu'il soit inclus dans une recherche FULLTEXT. |
| last\_insert\_id | Valeur retournée par la fonction LAST\_INSERT\_ID(). |
| max\_connections | Nombre maximal de connexions autorisées au serveur. |
| storage\_engine | Moteur de stockage par défaut. |
| tx\_isolation | Définit le niveau d'isolation des transactions. |
| unique\_checks | Définit si les contraintes d'unicité doivent être vérifiées. |

Pour connaître les valeurs actuelles des variables système, on peut utiliser la commande suivante :

SHOW VARIABLES;

Si l'on a une idée de la variable que l'on cherche, on peut utiliser la clause LIKE, ou la clause WHERE.

**Exemple 1 :** variables en rapport avec l'auto-incrémentation.

SHOW VARIABLES LIKE '%auto\_increment%';

**Exemple 2 :** affichage de la valeur de *unique\_checks*.

SHOW VARIABLES LIKE 'unique\_checks';

On peut utiliser également une requête SELECT, en faisant précéder le nom de la variable de deux caractères @.

**Exemple**

SELECT @@autocommit;

**Niveau des variables système**

Les variables système existent à deux niveaux différents :

* **Global :** c'est la variable au niveau du serveur MySQL même.
* **Session :** c'est la variable au niveau de la session.

Lorsque l'on démarre le serveur MySQL, les variables système sont donc initialisées à leur valeur par défaut (nous verrons comment modifier ces valeurs plus tard), au niveau global.  
Lorsque l'on ouvre une session MySQL et que l'on se connecte au serveur, les variables système au niveau session sont initialisées à partir des variables système globales. Il est cependant possible de modifier les variables système, au niveau de la session ou directement au niveau global. On peut donc se retrouver avec des variables système différentes pour les deux niveaux.

Il est possible de préciser le niveau que l'on désire afficher avec SHOW VARIABLES, en ajoutant GLOBAL  ou SESSION  (ou LOCAL, qui est synonyme de SESSION).

SHOW GLOBAL VARIABLES;

SHOW SESSION VARIABLES;

SELECT @@GLOBAL.nom\_variable;

SELECT @@SESSION.nom\_variable;

Par défaut, si rien n'est précisé, ce sont les variables de session qui sont affichées.

**Variables système n'existant qu'à un niveau**

Il existe un certain nombre de variables pour lequel le niveau global n'existe pas. Ces variables sont initialisées au début de la connexion, et leur modification ne peut affecter que la session en cours.  
La variable système *autocommit,* par exemple, n'existe que pour la session. De même que *last\_insert\_id*, ce qui est logique : LAST\_INSERT\_ID()  renvoie la valeur de la dernière auto-incrémentation réalisée par la session. Cela n'a donc pas de sens d'avoir cette variable à un niveau global.

De même, certaines variables système n'ont pas de sens au niveau de la session et n'y existent donc pas. C'est le cas par exemple de la variable *max\_connections*, qui détermine le nombre maximum de sessions connectées simultanément au serveur.

Pour ces variables, le comportement de la commande SHOW VARIABLES  diffère du comportement de la commande SELECT @@nom\_variable.

* Si vous ne précisez pas le niveau voulu avec SHOW VARIABLES, vous aurez la valeur de la variable de session si elle existe, sinon, la valeur au niveau global. SELECT  agira de la même manière.
* Si vous précisez un niveau et qu'une des variables n'existe pas à ce niveau, SHOW  donnera sa valeur à l'autre niveau (sans prévenir).
* Par contre, si vous précisez un niveau avec SELECTet que la variable système n'y existe pas, une erreur est déclenchée.

**Exemples :***last\_insert\_id* n'existe qu'au niveau de la session, *max\_connections* n'existe qu'au niveau global.

SHOW VARIABLES LIKE 'last\_insert\_id';

SHOW SESSION VARIABLES LIKE 'last\_insert\_id';

SHOW GLOBAL VARIABLES LIKE 'last\_insert\_id';

## Configuration et options

Dans ce dernier chapitre, vous apprendrez à configurer MySQL :

* en modifiant les variables système avec la commande SET  ;
* en utilisant les options du logiciel client au démarrage d'une session ;
* en utilisant les options du serveur lors de son démarrage ;
* en modifiant le fichier de configuration de MySQL.

Nous commencerons bien sûr par voir ce qu'est exactement une variable système.

Notez que ces variables système sont nombreuses, tout comme les options disponibles au niveau du client et du serveur, ainsi que dans le fichier de configuration.  
Pour ne pas vous donner de longues listes indigestes, je vous renverrai très souvent vers la documentation officielle au cours de ce chapitre, et ne vous donnerai que quelques exemples parmi les plus utilisés.

### Variables système

Le comportement de MySQL est régi par une série de valeurs contenues dans les**variables système**. Elles déterminent par exemple à quelle valeur commencent les colonnes auto-incrémentées (par défaut : 1), de combien ces valeurs s'incrémentent à chaque insertion (par défaut : 1 également), le moteur de stockage par défaut, le temps d'attente acceptable lorsqu'une requête se heurte à un verrou, etc.

Voici quelques variables système. Vous verrez que vous en connaissez déjà quelques-unes.  
Pour une liste complète, reportez-vous à la documentation officielle.

| **Nom** | **Définition** |
| --- | --- |
| autocommit | Définit si le mode autocommit est activé ou non, donc si les requêtes sont automatiquement commitées ou s'il est nécessaire de les commiter pour qu'elles prennent effet. |
| character\_set\_client | Jeu de caractères (encodage) utilisé par le client MySQL. |
| default\_week\_format | Mode par défaut de la fonction WEEK(). |
| foreign\_key\_checks | Définit si les contraintes de clé étrangère doivent être vérifiées. |
| ft\_min\_word\_len | Taille minimale d'un mot pour qu'il soit inclus dans une recherche FULLTEXT. |
| last\_insert\_id | Valeur retournée par la fonction LAST\_INSERT\_ID(). |
| max\_connections | Nombre maximal de connexions autorisées au serveur. |
| storage\_engine | Moteur de stockage par défaut. |
| tx\_isolation | Définit le niveau d'isolation des transactions. |
| unique\_checks | Définit si les contraintes d'unicité doivent être vérifiées. |

Pour connaître les valeurs actuelles des variables système, on peut utiliser la commande suivante :

SHOW VARIABLES;

Si l'on a une idée de la variable que l'on cherche, on peut utiliser la clause LIKE, ou la clause WHERE.

**Exemple 1 :** variables en rapport avec l'auto-incrémentation.

SHOW VARIABLES LIKE '%auto\_increment%';

| **Variable\_name** | **Value** |
| --- | --- |
| auto\_increment\_increment | 1 |
| auto\_increment\_offset | 1 |

**Exemple 2 :** affichage de la valeur de unique\_checks.

SHOW VARIABLES LIKE 'unique\_checks';

| **Variable\_name** | **Value** |
| --- | --- |
| unique\_checks | ON |

On peut utiliser également une requête SELECT, en faisant précéder le nom de la variable de deux caractères @.

**Exemple**

SELECT @@autocommit;

| **@@autocommit** |
| --- |
| 1 |

#### Niveau des variables système

Les variables système existent à deux niveaux différents :

* **Global :** c'est la variable au niveau du serveur MySQL même.
* **Session :** c'est la variable au niveau de la session.

Lorsque l'on démarre le serveur MySQL, les variables système sont donc initialisées à leur valeur par défaut (nous verrons comment modifier ces valeurs plus tard), au niveau global.  
Lorsque l'on ouvre une session MySQL et que l'on se connecte au serveur, les variables système au niveau session sont initialisées à partir des variables système globales. Il est cependant possible de modifier les variables système, au niveau de la session ou directement au niveau global. On peut donc se retrouver avec des variables système différentes pour les deux niveaux.

Il est possible de préciser le niveau que l'on désire afficher avec SHOW VARIABLES, en ajoutant GLOBAL  ou SESSION  (ou LOCAL, qui est synonyme de SESSION).

SHOW GLOBAL VARIABLES;

SHOW SESSION VARIABLES;

SELECT @@GLOBAL.nom\_variable;

SELECT @@SESSION.nom\_variable;

Par défaut, si rien n'est précisé, ce sont les variables de session qui sont affichées.

##### Variables système n'existant qu'à un niveau

Il existe un certain nombre de variables pour lequel le niveau global n'existe pas. Ces variables sont initialisées au début de la connexion, et leur modification ne peut affecter que la session en cours.  
La variable système autocommit, par exemple, n'existe que pour la session. De même que last\_insert\_id, ce qui est logique : LAST\_INSERT\_ID()  renvoie la valeur de la dernière auto-incrémentation réalisée par la session. Cela n'a donc pas de sens d'avoir cette variable à un niveau global.

De même, certaines variables système n'ont pas de sens au niveau de la session et n'y existent donc pas. C'est le cas par exemple de la variable max\_connections, qui détermine le nombre maximum de sessions connectées simultanément au serveur.

Pour ces variables, le comportement de la commande SHOW VARIABLES  diffère du comportement de la commande SELECT @@nom\_variable.

* Si vous ne précisez pas le niveau voulu avec SHOW VARIABLES, vous aurez la valeur de la variable de session si elle existe, sinon, la valeur au niveau global. SELECT  agira de la même manière.
* Si vous précisez un niveau et qu'une des variables n'existe pas à ce niveau, SHOW  donnera sa valeur à l'autre niveau (sans prévenir).
* Par contre, si vous précisez un niveau avec SELECTet que la variable système n'y existe pas, une erreur est déclenchée.

**Exemples :**last\_insert\_id n'existe qu'au niveau de la session, max\_connections n'existe qu'au niveau global.

SHOW VARIABLES LIKE 'last\_insert\_id';

SHOW SESSION VARIABLES LIKE 'last\_insert\_id';

SHOW GLOBAL VARIABLES LIKE 'last\_insert\_id';

Ces trois requêtes donneront exactement le même résultat :

| **Variable\_name** | **Value** |
| --- | --- |
| last\_insert\_id | 0 |

SHOW VARIABLES LIKE 'max\_connections';

SHOW SESSION VARIABLES LIKE 'max\_connections';

SHOW GLOBAL VARIABLES LIKE 'max\_connections';

| **Variable\_name** | **Value** |
| --- | --- |
| max\_connections | 151 |

Avec SELECT, si l'on ne précise pas le niveau, ou si l'on précise le bon niveau, le résultat est le même.

SELECT @@max\_connections AS max\_connections,

@@last\_insert\_id AS last\_insert\_id;

SELECT @@GLOBAL.max\_connections AS max\_connections,

@@SESSION.last\_insert\_id AS last\_insert\_id;

| **max\_connections** | **last\_insert\_id** |
| --- | --- |
| 151 | 0 |

Par contre, si l'on précise le mauvais niveau, on obtient une erreur.

SELECT @@SESSION.max\_connections;

SELECT @@GLOBAL.last\_insert\_id;

ERROR 1238 (HY000): Variable 'max\_connections' is a GLOBAL variable

ERROR 1238 (HY000): Variable 'last\_insert\_id' is a SESSION variable

### Modification des variables système avec SET

Comme pour les variables utilisateur et les variables locales, il est possible de modifier la valeur des variables système en utilisant la commande SET.

Cependant, toutes ne permettent pas ce genre de modification. La liste des variables système qui l'autorisent, appelées variables système dynamiques, se trouve dans la documentation officielle.

Pour modifier une variable au niveau global, il est nécessaire d'avoir le privilège global SUPER. À moins que vous n'ayez accordé ce privilège à l'un de vos utilisateurs, seul l'utilisateur "root" en est capable.

Deux syntaxes sont possibles avec SET  :

*SET* niveau nom\_variable = valeur;

-- OU

*SET* @@niveau.nom\_variable = valeur;

**Exemples**

*SET* SESSION max\_tmp\_tables = 5; -- Nombre maximal de tables temporaires

*SET* @@GLOBAL.storage\_engine = InnoDB; -- Moteur de stockage par défaut

On peut aussi omettre le niveau, auquel cas c'est la variable système au niveau de la session qui sera modifiée. Si elle n'existe qu'au niveau global, une erreur sera déclenchée.

**Exemples**

*SET* max\_tmp\_tables = 12;

*SET* @@max\_tmp\_tables = 8;

*SET* @@max\_connections = 200;

Les deux premières commandes fonctionnent, mais la troisième échoue avec l'erreur suivante :

ERROR 1229 (HY000): Variable 'max\_connections' is a GLOBAL variable and should be set with SET GLOBAL

#### Effet de la modification selon le niveau

Il est important de se rendre compte de la différence qu'il y a entre modifier la valeur d'une variable système au niveau de la session et au niveau du serveur (niveau global).

Lorsque l'on modifie une variable système globale, la valeur de la même variable système au niveau de la session ne change pas.  
Par conséquent, cela n'affecte pas du tout la session en cours.

Par contre, cela affectera toutes les sessions qui se connecteront au serveur après la modification (jusqu'à arrêt/redémarrage du serveur).

**Exemple :** on a modifié storage\_engine au niveau global, cela n'a pas affecté storage\_engine au niveau session.

SELECT @@GLOBAL.storage\_engine, @@SESSION.storage\_engine;

| **@@GLOBAL.storage\_engine** | **@@SESSION.storage\_engine** |
| --- | --- |
| InnoDB | MyISAM |

Si l'on crée une table dans la session courante, elle utilisera toujours le moteur MyISAM par défaut. Par contre, si l'on ouvre une autre session, les variables système de session étant initialisées à la connexion à partir des variables système du serveur, storage\_engine aura la valeur InnoDB pour le serveur et pour la session. Par conséquent, toute table créée par cette nouvelle session utilisera InnoDB par défaut.

À l'inverse, modifier la valeur d'une variable système au niveau de la session n'affectera que la session. Toutes les sessions futures reprendront la valeur globale à l'initialisation des variables système de la session.

#### Les commandes SET spéciales

Pour certaines variables système, MySQL a créé des commandes SET  spéciales.

**Exemples**

*SET* NAMES encodage;

Cette commande modifie trois variables système au niveau de la session : character\_set\_client, character\_set\_connection et character\_set\_results.

*SET* [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL { REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE }

Cette commande modifie la variable système tx\_isolation au niveau demandé.

### Options au démarrage du client mysql

Lorsque l'on se connecte au serveur avec un client mysql, on peut préciser une série d'options. Vous en connaissez déjà plusieurs : l'hôte, l'utilisateur et le mot de passe sont des options.

Pour rappel, ces options peuvent être données de deux manières : la manière courte, et la longue.

mysql --host=localhost --user=student --password=motdepasse

#OU

mysql -h localhost -u student -pmotdepasse

On peut mélanger les deux notations.

Pour préciser une option, on utilise donc --option[=valeur]. Certaines options permettent un raccourci, et certaines ne nécessitent pas qu'on leur donne une valeur.

En voici quelques-unes.

| **Option** | **Raccourci** | **Explication** |
| --- | --- | --- |
| --default-character-set=encodage | / | Définit l'encodage par défaut. |
| --delimiter=delim | / | Modifie le délimiteur. |
| --no-beep | -b | Le client n'émet plus de son en cas d'erreur. |
| --execute=requetes | -e requetes | Exécute les requêtes données (séparées par ;), puis quitte le client. |
| --init-command=requete | / | Exécute la requête donnée dès le démarrage du client. |
| --safe-updates ou --i-m-a-dummy | -U | N'exécute les requêtes UPDATE  et DELETE  que si les lignes à modifier/supprimer sont spécifiées explicitement grâce à une clause WHERE  sur un index, ou limitées par un LIMIT. Cela empêche l'exécution de commandes comme DELETE FROM nomTable;  qui supprime toutes les données de la table. |
| --skip-column-names | -N | Les noms des colonnes ne sont pas affichés dans les résultats. |
| --vertical | -E | Écrit les résultats de manière verticale (comme lorsque l'on termine une requête par \G). |

Pour la liste complète, comme d'habitude, vous trouverez votre bonheur dans la [documentation officielle](https://dev.mysql.com/doc/refman/5.5/en/mysql-command-options.html).

Certaines options ont un effet sur les variables système, au niveau de la session. Par exemple, --default-character-set modifie trois variables système : character\_set\_client, character\_set\_connection et character\_set\_results. C'est donc l'équivalent de la commande SET NAMES encodage;.  
D'autres, comme --no-beep, n'ont vraiment d'effet que sur le logiciel client.

**Exemple**

mysql -u student -p elevage --skip-column-names

SELECT id, nom, espece\_id, prix

FROM Race;

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | Berger allemand | 1 | 450.00 |
| 2 | Berger blanc suisse | 1 | 900.00 |
| 3 | Singapura | 2 | 950.00 |
| 4 | Bleu russe | 2 | 800.00 |
| 5 | Maine Coon | 2 | 700.00 |
| 7 | Sphynx | 2 | 1200.00 |
| 8 | Nebelung | 2 | 950.00 |
| 9 | Rottweiller | 1 | 600.00 |

### Options au démarrage du serveur mysqld

Vous connaissez maintenant deux techniques permettant de modifier des variables système de session :

* pendant la session avec SET @@SESSION.nomVariable = valeur  ;
* au démarrage de la session avec les options du client mysql.

Pour les variables système globales, il existe l'équivalent :

* vous savez déjà qu'il est possible de faire SET @@GLOBAL.nomVariable = valeur  ;
* nous allons maintenant voir comment préciser des options au démarrage **du serveur**.

Démarrer le serveur ? On n'a jamais démarré le serveur !

Mais si ! Tout au début, lors de l'installation de MySQL, le serveur a été démarré. Selon votre configuration et/ou votre système d'exploitation, il a été démarré soit automatiquement, soit en tapant mysqld\_safe ou mysqld en ligne de commande. Depuis, chaque fois que vous rallumez votre ordinateur, le serveur est démarré automatiquement (c'est en tout cas le comportement par défaut, modifiable à l'installation et/ou par la suite).

Pour stopper le serveur, on utilise l'utilitaire mysqladmin (en ligne de commande, bien sûr) :

mysqladmin -u root -p shutdown

Bien entendu, si vous n'avez pas défini de mot de passe pour l'utilisateur root (mauvaise idée !), enlevez le -p.

Et pour (re)démarrer le serveur :

mysqld

Ou, si vous êtes sous MacOS ou Linux, vous pouvez utiliser mysqld\_safe, qui ajoute quelques options de sécurisation :

mysqld\_safe

Vous connaissez donc maintenant quatre logiciels installés lors de la mise en place de MySQL sur votre ordinateur :

* ***myqsl :*** le logiciel client ;
* ***mysqld :*** le serveur ;
* ***mysqladmin :*** qui permet des opérations d'administration ;
* ***mysqldump :*** qui permet de faire des backups de vos bases de données.

##### Les options du serveur

La syntaxe est la même que lors du démarrage du client : --option[=valeur].

**Quelques exemples :**

| **Option** | **Raccourci** | **Explication** |
| --- | --- | --- |
| --character-set-server=encodage | -C charset\_name | Définit l'encodage par défaut utilisé par le serveur. |
| --default-storage-engine=type | / | Définit le moteur de stockage par défaut. |
| --default-time-zone=timezone | / | Définit le fuseau horaire à utiliser. |
| --init-file=nomFichier | / | Les requêtes définies dans le dossier donné sont exécutées au démarrage du serveur. |
| --language=langue | -L langue | Définit la langue à utiliser pour les messages d'erreur. |
| --transaction-isolation=niveau | / | Définit le niveau d'isolation des transactions. |

La plupart des options du serveur (mais pas toutes) modifient les variables système globales.

**Exemple :** on lance le serveur en définissant l'encodage utilisé par celui-ci.

mysqld -C greek

Il suffit alors de démarrer une session pour tester notre nouvelle configuration.

SHOW GLOBAL VARIABLES LIKE 'character\_set\_server';

| **Variable\_name** | **Value** |
| --- | --- |
| character\_set\_server | greek |

### Fichiers de configuration

Si l'on veut garder la même configuration en permanence malgré les redémarrages de serveur et pour toutes les sessions, il existe une solution plus simple que de démarrer chaque fois le logiciel avec les options désirées : utiliser **les fichiers de configuration**.

Ceux-ci permettent de préciser, pour chacun des logiciels de MySQL (le client mysql, le serveur mysqld, l'utilitaire mysqladmin, etc.), une série d'options qui **seront prises en compte à chaque démarrage** du logiciel.

#### Emplacement du fichier

Lors de leur démarrage, les logiciels MySQL vérifient l'existence de fichiers de configuration à différents endroits.  
Si plusieurs fichiers de configuration sont trouvés, ils sont tous utilisés. Si une option est spécifiée plusieurs fois (par plusieurs fichiers différents), c'est la dernière valeur qui est prise en compte (l'ordre dans lequel les fichiers de configuration sont lus est donné ci-dessous).  
Les emplacements vérifiés sont différents selon que l'on utilise Windows ou Unix.

Il est tout à fait possible que ces fichiers de configuration n'existent pas. Dans ce cas, il suffit de le (ou les) créer avec un simple éditeur de texte.

##### Pour Windows

**Dans l'ordre**, trois emplacements sont utilisés.

| **Emplacement** | **Commentaire** |
| --- | --- |
| WINDIR\my.ini, WINDIR\my.cnf | WINDIR est le dossier de Windows. Généralement, il s'agit du dossier C:\Windows. Pour vérifier, il suffit d'exécuter la commande suivante (dans la ligne de commande Windows) : echo %WINDIR% |
| C:\my.ini ou C:\my.cnf | - |
| INSTALLDIR\my.ini ou INSTALLDIR\my.cnf | INSTALLDIR est le dossier dans lequel MySQL a été installé. |

##### Pour Linux et MacOS

Les emplacements suivants sont parcourus, dans l'ordre.

| **Emplacement** | **Commentaire** |
| --- | --- |
| /etc/my.cnf | - |
| /etc/mysql/my.cnf | - |
| ~/.my.cnf | ~/ est le répertoire de l'utilisateur Unix. Dans un système avec plusieurs utilisateurs, cela permet de définir un fichier de configuration pour chaque utilisateur Unix (le fichier n'étant lu que pour l'utilisateur Unix courant). |

Il existe deux autres emplacements possibles pour les fichiers de configuration sous Unix. Ceux-ci sont les principaux et les plus simples.

##### Fichier de configuration fourni au démarrage

Que ce soit sous Windows ou sous Unix, il est également possible de donner un fichier de configuration dans les options au démarrage du logiciel. Dans ce cas, le fichier peut se trouver n'importe où, il suffit de fournir le chemin complet.

Voici les options permettant cela :

| **Option** | **Commentaire** |
| --- | --- |
| --defaults-extra-file=chemin\_fichier | Le fichier de configuration spécifié est utilisé **en plus** des éventuels autres fichiers de configuration. Ce fichier est utilisé en dernier, sauf sous Unix où le fichier localisé dans le dossier racine de l'utilisateur Unix est toujours utilisé en tout dernier. |
| --defaults-file=chemin\_fichier | Seul ce fichier de configuration est utilisé, les autres sont ignorés. |

**Exemple**

mysql -u student -p --defaults-extra-file=/Users/taguan/Documents/SQLtuto/mysqlConfig.cnf

#### Structure du fichier

Un fichier de configuration MySQL peut contenir trois types de lignes différentes :

* **option ou option=valeur :** définit l'option à utiliser. C'est exactement la même syntaxe que pour les options à préciser lors du démarrage du logiciel, à ceci près que l'on omet les deux tirets -.
* **[logiciel]** ou **[groupe] :** définit le logiciel ou le groupe auquel les options s'appliquent.
* **#commentaire** ou **;commentaire :** ligne de commentaire, elle sera ignorée. Notez qu'il est possible de commencer un commentaire au milieu d'une ligne avec #.

**Exemple**

#début des options pour le serveur mysqld

[mysqld]

character-default-set=utf8 # on modifie l'encodage du serveur

timezone='+01:00' # on ajuste le fuseau horaire du serveur

default-storage-engine=InnoDB # on définit le moteur de stockage par défaut

#début des options pour le client mysql

[mysql]

character-set=utf8 # on modifie l'encodage client

no-beep # le silence est d'or

##### Les options

Toutes les options disponibles en ligne de commande lors du démarrage d'un logiciel sont utilisables dans un fichier de configuration (pour le logiciel correspondant).  
On omet simplement les deux caractères - avant l'option, et les raccourcis ne peuvent pas être utilisés.

Si l'option nécessite que l'on précise une valeur, celle-ci peut, mais ne doit pas, être entourée de guillemets. Si la valeur donnée comprend un #, les guillemets seront nécessaires pour que ce caractère ne soit pas considéré comme le début d'un commentaire.

Par ailleurs, les espaces avant et après l'option sont ignorées, et les espaces autour du signe = sont autorisées (ce n'est pas le cas en ligne de commande).

##### Balise [logiciel] et [groupe]

Les balises [logiciel] et [groupe] permettent de spécifier **à quel(s) logiciel(s) s'appliquent les options suivant cette balise** (jusqu'à la balise [logiciel] ou [groupe] suivante, ou jusqu'à ce que la fin du fichier soit atteinte).

On peut donc soit donner le nom du logiciel concerné ([mysqld], [mysql], [mysqldump], etc.), soit donner le groupe [client].  
Si le groupe [client] est spécifié, **les options suivantes seront prises en compte par tous les logiciels clients**. mysqldump, mysqladmin et mysql sont tous trois des logiciels clients.

Les options données pour ce groupe doivent être valables pour tous les logiciels clients. Si vous essayez de démarrer un logiciel client alors qu'une option non valable pour celui-ci a été donnée dans le fichier de configuration, le logiciel quittera avec une erreur.

Bien entendu, il est possible de spécifier une balise [client] pour les options communes à tous les clients, puis une balise [logiciel] pour les options spécifiques à un logiciel.

**Exemple**

[client]

port=3306 # on précise le port à utiliser pour tous les logiciels clients

[mysqld]

port=3306 # on précise le port aussi pour le serveur

character-default-set=utf8

[mysql]

no-beep

Tous les logiciels clients, ainsi que le serveur utiliseront le port 3306, mais seul le logiciel mysql utilisera l'option no-beep.