

# Bienvenue !

Pierre-Julien VILLOUD



- [pjvilloud@protonmail.com](mailto:pjvilloud@protonmail.com)
- <https://github.com/pjvilloud>
- <https://linkedin.com/pjvilloud>
- <http://pjvilloud.github.io>

# Introduction

⚠ MySQL est un système de gestion de bases de données relationnelles. Il reste le deuxième SGBDR le plus utilisé au monde (derrière Oracle et devant SQL Server) même s'il est en perte de vitesse.



ℹ MySQL est un logiciel libre et open source, mais distribué sous double license (GPL et propriétaire).

# Historique

Créé par Michael Widenius en 1995

**Version 4 en 2004**

(sous-requêtes et *prepared statements*)

**Version 5 en 2005**

( curseurs, procédures stockées, *triggers*, vues)

Racheté par *Sun Microsystems* en 2008

Oracle rachète Sun en 2010

**Version 5.7 en 2015**

(dernière mise à jour le 17/07/2017)

# Avantages de MySQL

Communauté

Open Source...

Orienté Web, Cloud, Big Data

Solidité et fiabilité

Supports OS

# Inconvénients de MySQL

Open Source ?

Fonctionnalités

De grands noms quittent MySQL

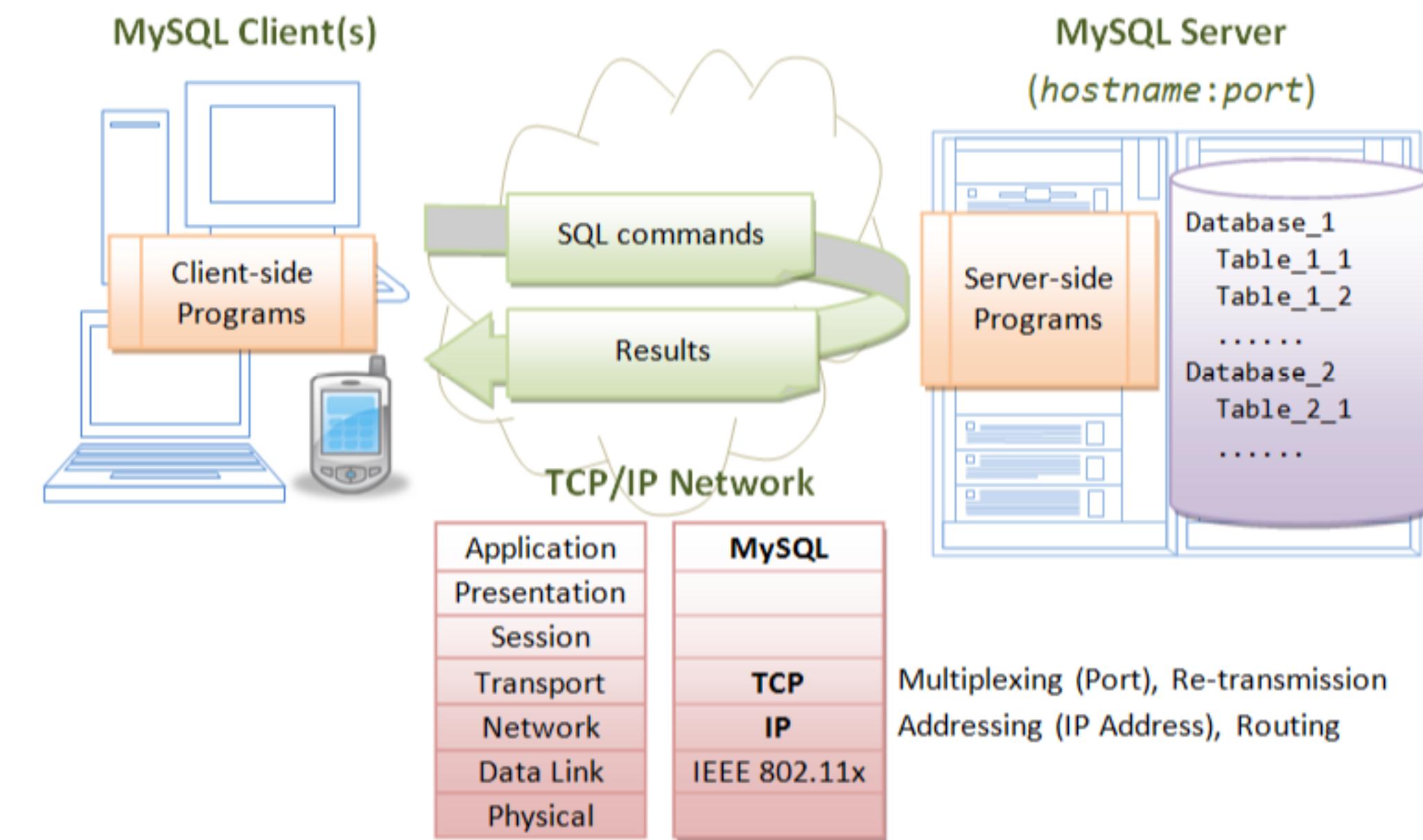
Syntaxe

Avenir incertain



# Principe du *client - serveur*

MySQL, comme beaucoup de SGBDR, repose sur le principe du *client - serveur*. La base de données à proprement parler est installée sur un **serveur**. Afin de communiquer ou d'interagir avec ce serveur, il est nécessaire de passer par un logiciel **client**.



Bien que ce soit souvent le cas dans les applications en entreprise, il n'est pas nécessaire que le client soit installé sur une machine différente de celle du serveur. Il peut bien évidemment avoir plusieurs clients qui accèdent à un même serveur.

# Vocabulaire

 Un peu de vocabulaire est nécessaire lorsqu'on parle de SGBDR.

Base de données

Schéma

Table

Colonne

Ligne

Requêtes

Contraintes

# Installation

MySQL est disponible pour de nombreux systèmes d'exploitation. Nous verrons l'installation sous Windows et Linux.

## Windows

Pour installer le serveur, [l'installateur MySQL](#) est recommandé.

Installer le client MySQL Workbench en téléchargeant l'installateur [ici](#).

## Linux (console)

Exécuter la commande suivante dans une console de votre VM Linux

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

✓ Ça y'est un serveur MySQL et un client en ligne de commande sont installés sur votre VM Linux, et un client graphique est installé sur votre VM Windows !

# Actions sur le serveur

❶ Une fois MySQL installé, il est possible d'interagir avec le serveur avec le script `/etc/init.d/mysql` (sous Linux) et via un service (sous Windows).

```
/etc/init.d/mysql start|stop|restart|reload|force-reload|status  
■ + R 'services.msc'
```

✍ Le serveur peut être démarré, arrêté, redémarré... On peut également consulter son statut. Il est également possible d'utiliser la commande `mysqladmin`.

# Les premiers réflexes

Une fois MySQL installé, effectuer les actions suivantes permet d'assurer la sécurité et la simplicité d'utilisation :

## 1 : Changer le mot de passe root

```
sudo mysqladmin -u root -h localhost password nouveau_mdp
```

## 2 : Interdire l'accès distant pour root

```
UPDATE mysql.user SET Host = 'localhost' WHERE user='root'; FLUSH PRIVILEGES;
```

## 3 : Supprimer le compte anonymous

```
DELETE FROM mysql.user WHERE User=''; FLUSH PRIVILEGES;
```

## 4 : Supprimer les éventuelles bases de test créées

```
DROP DATABASE test; DELETE FROM db WHERE db='test' OR db='test\_%';
```

# Développement côté client

💡 Par l'intermédiaire d'un client (graphique ou en ligne de commande), il va être possible d'effectuer les actions suivantes :

Manipuler des bases de données
Manipuler des tables
Manipuler des données
Exécuter des requêtes SQL issues d'un fichier
Manipuler des index
Gérer des contraintes
Gérer des index
Effectuer des requêtes complexes
Manipuler des vues

# Le client mysql

❶ Le client en ligne de commande `mysql` permet d'effectuer toutes les opérations possibles sur un serveur MySQL.

#Connexion au serveur MySQL local avec l'utilisateur courant et sans mot de passe

```
mysql
```

#Idem mais avec l'utilisateur 'toto' et sans mot de passe

```
mysql -u toto
```

#Idem mais avec le mot de passe 'pass'

```
mysql -u toto -p pass
```

#Idem mais demande le mot de passe

```
mysql -u toto -p
```

#Idem mais se connecte à la base 'base'

```
mysql -u toto -p -D base
```

#Idem mais connexion à un serveur distant dont l'adresse est 'hote'

```
mysql -u toto -p -D base -h hote
```

#Idem mais on spécifie le port (par défaut 3306)

```
mysql -u toto -p -D base -h hote -P 3307
```

❷ Pour toutes les informations sur la commande `mysql`, tapez dans une console `man mysql` ou `mysql --help` pour accéder à la documentation.

# Créer une base de données

**i** Pour créer une **base de données** ou un **schéma** (équivalent en MySQL), on utilise la commande suivante :

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name [create_specification] ...
```

create\_specification:

[DEFAULT] CHARACTER SET [-] char\_set\_name

[DEFAULT] COLLATE [=] collation\_name

La **collation** représente un ensemble de règle pour comparer les caractères entre eux. Lorsqu'il n'est pas spécifié, c'est la collation par défaut de l'encodage qui est utilisé.

**IF NOT EXISTS** permet de ne pas créer la base si elle existe déjà. Sans cette commande, la tentative de création d'une base existante retourne une erreur.

Le nom qui sera donné à la base de données

Cette commande permet de définir l'**encodage** utilisé pour stocker les données de type texte. Lorsqu'il n'est pas spécifié, c'est l'encodage par défaut qui est utilisé.

--crée la base de données 'mabase'

```
CREATE DATABASE IF NOT EXISTS mabase;
```

--Crée la base 'mabase' avec l'encodage 'utf8' et la collation utf8\_general\_ci

```
CREATE DATABASE mabase CHARACTER SET utf8 COLLATE utf8_general_ci;
```

--Pour voir l'encodage et la collation par défaut du serveur

```
SELECT @@character_set_server, @@collation_server;
```

**i** On recommandera l'utilisation de l'encodage **utf8** afin de gérer correctement les caractères accentués.

# Modifier une base de données

**i** Pour modifier une **base de données**, on utilise la commande suivante :

```
ALTER {DATABASE | SCHEMA} [db_name]
      alter_specification ...
ALTER {DATABASE | SCHEMA} db_name

alter_specification:
  [DEFAULT] CHARACTER SET [=] charset_name
  | [DEFAULT] COLLATE [=] collation_name

--Modifie la base 'mabase' pour utiliser l'encodage utf-8
--et la collation utf8_general_ci
ALTER DATABASE mabase CHARACTER SET = 'utf8' COLLATE = utf8_general_ci;
```

**i** On recommandera l'utilisation de l'encodage **utf8** afin de gérer correctement les caractères accentués.

# Supprimer une base de données

 Pour supprimer une **base de données**, on utilise la commande suivante :

```
DROP {DATABASE | SCHEMA} [IF EXISTS] db_name
```

IF EXISTS permet de ne supprimer une table que si cette dernière existe. Si cette commande n'est pas utilisée, une tentative de suppression d'une table non existante remontera une erreur.

--Supprime la base *donnees*  
`DROP DATABASE mabase;`

 Évidemment, supprimer une base de données supprime toutes les données contenu dans cette dernière...

# Les types de données

 MySQL supporte la plupart des **types de données** de SQL.

Types numériques

Types date et heure

Types chaîne

Autres types

 MySQL propose également ces **propres types**.

# Types de données entier

**i** MySQL supporte tous les **types de données** entier standards de SQL ainsi que d'autres spécifiques à MySQL.

Type	Occupation en mémoire	Intervalle (Signed)	Intervalle (Unsigned)
BIT(M)	1 bit à 8 octets	Ce type est utilisé pour stocker des valeurs binaires, avec la notation <code>b'value'</code> (ex : <code>b'111'</code> = 7)	
TINYINT(M)	1 octet	-128 à 127	0 à 255
BOOL ou BOOLEAN	1 octet	Les booléens sont représentés comme des <code>TINYINT(1)</code> . La valeur 0 équivaut à <code>false</code> , toute autre valeur à <code>true</code> . A noter que les valeurs <code>TRUE</code> et <code>FALSE</code> sont respectivement des alias de 1 et 0.	
SMALLINT(M)	2 octets	-37 768 à 32 767	0 à 65 535
MEDIUMINT(M)	3 octets	-8 388 608 à 8 388 607	0 à 16 777 215
INT(M) ou INTEGER(M)	4 octets	-2 147 489 648 à 2 147 483 647	0 à 4 294 967 295
BIGINT(M)	8 octets	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807	0 à 18 446 744 073 709 551 615

**i** `M` représente la largeur maximum d'affichage pour les types numériques. C'est un paramètre facultatif et sa valeur maximale est 255. Ce paramètre ne contraint pas l'intervalle de valeurs qui peut être stocké dans chaque type.

# Types de données à virgule fixe

**i** Les **types de données** à virgule fixe permettent de stocker des valeurs décimales exactes, comme des données monétaires par exemple (prix, salaire...).

Type	Occupation en mémoire	Exemple
<code>DECIMAL(P,E)</code> ou <code>NUMERIC(P,E)</code>	X octets pour la partie décimale, Y octets pour la partie entière	<code>DECIMAL(5,2)</code> permet de représenter 5 chiffres dont 2 après la virgule, soit l'intervalle -999.99 à 999.99

**Nombre de chiffres Nombre d'octets**

0	0
1-2	1
3-4	2
5-6	3
7-9	4

**i** `P` représente la précision, c'est-à-dire le nombre de chiffres significatifs à stocker, tandis que `E` représente l'échelle, c'est-à-dire le nombre de chiffres après la virgule. A noter que `P` a pour valeur par défaut 10 (maximum 65) et `E` 0.

# Types de données à virgule flottante

❶ Les types de données à virgule flottante (standards ou spécifiques) permettent de stocker des valeurs décimales approchées.

Type	Occupation en mémoire	Remarques
FLOAT(P)	4 octets	P est facultatif mais s'il est spécifié et qu'il est compris entre 0 et 23, ce sera bien un type FLOAT qui sera utilisé, si c'est entre 24 et 53, c'est un type DOUBLE qui sera en fait utilisé.
DOUBLE	8 octets	
FLOAT(P, E) ou REAL(P, E) ou DOUBLE PRECISION(P, E)	4 ou 8 octets	FLOAT(5, 2) permet de représenter 5 chiffres dont 2 après la virgule, soit l'intervalle -999.99 à 999.99

❶ Ces types sont à utiliser avec précaution car des opérations d'arrondi sont effectuées par MySQL. Pour des valeurs fixes, préférer le type DECIMAL

# Précisions sur les types numériques

❶ **UNSIGNED** peut être spécifié après le type (entier ou flottant) pour préciser que les valeurs seront positives ou nulles. L'intervalle de valeurs est décalé à partir de 0 (sauf pour les flottants où cela n'a pas d'effet). Ce paramètre ne fait pas partie du standard SQL.

❶ **ZEROFILL** peut être spécifié après le type (entier uniquement) lorsqu'on spécifie la largeur d'affichage. Ainsi un champ `SMALLINT(3) ZEROFILL`, 5 sera affiché `005` mais 1234 pourra bien être stocké dans ce champ et sera affiché `1234`. A noter qu'un champ **ZEROFILL** est automatiquement **UNSIGNED**.

❶ **AUTO\_INCREMENT** peut être spécifié après le type (entier ou flottant). Cela permet de gérer automatiquement des séquences.

❶ MySQL gère le dépassement d'intervalle de deux manières selon le mode du serveur. Lorsque le mode *strict* est activé, le serveur remonte une erreur pour tout dépassement. Lorsqu'il ne l'est pas, un avertissement est affiché au client et la valeur est tronquée ou affectée à la borne la plus proche. `TINYINT : 256 => 127 TINYINT UNSIGNED : 256 => 255`

# Types date

**i** Pour représenter des dates en MySQL, on utilise les types (standards ou spécifiques) suivants :

Type	Format	Remarques
DATE	'YYYY-MM-DD'	Utilisé pour les dates sans nécessité de stocker l'heure. L'intervalle des valeurs possibles est '1000-01-01' à '9999-12-31'
DATETIME	'YYYY-MM-DD HH:MM:SS[.fraction]'	Utilisé pour les dates et heure. L'intervalle des valeurs possibles est '1000-01-01 00:00:00' à '9999-12-31 23:59:59'.
TIMESTAMP	'YYYY-MM-DD HH:MM:SS[.fraction]'	Type correspondant au Timestamp Unix. L'intervalle des valeurs possibles est 1970-01-01 00:00:01 à '2038-01-19 03:14:07'
TIME	HH:MM:SS[.fraction]'	TIME peut être utilisé pour représenter une heure dans la journée, mais aussi pour représenter une durée. L'intervalle des valeurs possibles est '-838:59:59.000000' à '838:59:59.000000'.
YEAR	YYYY	L'intervalle des valeurs possibles est entre 1901 et 2155.

**i** Il est possible de préciser l'heure à la micro-seconde près. On utilise pour cela **fraction**. Ex : '2017-12-31 23:59:59:999999'

# Les types chaînes

**i** Pour stocker des **chaînes de caractère**, MySQL propose plusieurs types (**standards ou spécifiques**).

Type	Occupation mémoire	Remarques
CHAR(X)	X octets	X correspond à la taille maximum de la chaîne de caractère et doit être compris entre 0 et 255. Une chaîne de caractère dont la taille est inférieure à X est automatiquement complétée avec des espaces à droite.
VARCHAR(X)	(Longueur de la chaîne + 1) octet(s)	Contrairement à CHAR, les espaces en début et en fin sont conservés.
TEXT	(Longueur de la chaîne + (1~4)) octet(s)	TEXT 65 535 caractères TINYTEXT 255 caractères MEDIUMTEXT 16 777 215 caractères LONGTEXT 4 294 967 295 caractères

Occupation mémoire en fonction du type

Valeur	CHAR(4)	Mémoire	VARCHAR(4)	Mémoire
''	''	4 octets	''	1 octet
'ab'	'ab'	4 octets	'ab'	3 octets
'abcd'	'abcd'	4 octets	'abcd'	5 octets
'abcdef'	'abcd'	4 octets	'abcd'	5 octets

**i** On représente une chaîne de caractère entre simple quotes en MySQL : Ex : 'Hello' .

# Les types binaires

MySQL permet également de stocker des chaînes de caractères contenant des valeurs binaires avec les types ([standards](#) ou [spécifiques](#)) suivants :

Type	Équivalent
BINARY(X)	CHAR
VARBINARY(X)	VARCHAR
BLOB	TEXT
TINYBLOB	TINYTEXT
MEDIUMBLOB	MEDIUMTEXT
LONGBLOB	LONGTEXT

Ce genre de types est utile pour stocker des fichiers non textuels (images, zip, pdf...).

# Les types à valeurs restreintes

❶ Lorsque l'on veut stocker des valeurs ayant un ensemble de valeurs restreintes, on peut utiliser les types suivants :

Type	Exemple	Remarques	Stockage des valeurs														
ENUM	<code>ENUM( 'XS', 'S', 'M', 'L', 'XL' )</code>	Les valeurs sont stockées sous forme d'index entier et l'ordre de tri correspond à l'ordre dans lequel les valeurs sont déclarées. Un ENUM peut avoir jusqu'à 65 535 valeurs différentes. On peut l'utiliser soit par sa valeur 'M' soit par son index 3 .	<table><thead><tr><th>Valeur</th><th>Index</th></tr></thead><tbody><tr><td>NULL</td><td>NULL</td></tr><tr><td>''</td><td>0</td></tr><tr><td>'XS'</td><td>1</td></tr><tr><td>'S'</td><td>2</td></tr><tr><td>'M'</td><td>3</td></tr><tr><td>...</td><td></td></tr></tbody></table>	Valeur	Index	NULL	NULL	''	0	'XS'	1	'S'	2	'M'	3	...	
Valeur	Index																
NULL	NULL																
''	0																
'XS'	1																
'S'	2																
'M'	3																
...																	
SET	<code>SET('actu', 'politique', 'economie')</code>	Une valeur de type SET peut contenir 0 ou plusieurs valeurs parmi les valeurs spécifiées dans le type. Ces valeurs sont séparées par des , .															

❶ Il est fortement recommandé de ne pas utiliser de chiffres pour les valeurs d'un ENUM afin d'éviter de les confondre avec les index. On ne peut pas utiliser de valeurs contenant des , dans un SET.  
Il ne faut pas abuser des types ENUM et SET ...

# Les autres types

 MySQL propose également d'autres types [spécifiques](#) pour représenter des informations particulières.

Données géographiques

JSON

 Voir la [documentation](#) pour les détails concernant ces types.

Ça va ?



# Création de tables

Voici la syntaxe MySQL pour créer une table.

**CREATE TABLE [IF NOT EXISTS] tbl\_name (create\_definition,...)**

create\_definition:  
col\_name column\_definition

**NOT NULL** permet d'obliger de renseigner une valeur lors de l'insertion de données. **DEFAULT** permet de définir une valeur par défaut.

column\_definition:  
data\_type [**NOT NULL** | **NULL**] [**DEFAULT** default\_value  
[**AUTO\_INCREMENT**] [**UNIQUE** [**KEY**] | [**PRIMARY**] **KEY**]  
data\_type: INT | FLOAT | ...

Il est possible de définir des contraintes sur les colonnes directement lors de la définition de ces dernières.

- **AUTO\_INCREMENT** permet d'incrémenter automatiquement la valeur d'une colonne
- **UNIQUE** permet d'assurer l'unicité des valeurs d'une colonne
- **PRIMARY KEY** permet de définir la colonne en tant que clé primaire

--Exemple

```
CREATE TABLE IF NOT EXISTS vehicule (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    marque VARCHAR(50) NOT NULL,  
    modele VARCHAR(50),  
    immatriculation VARCHAR(30) NOT NULL UNIQUE,  
    date_mise_circulation DATE NOT NULL  
) ;
```

vehicule
- id : INT pk - marque : VARCHAR(50) - modele : VARCHAR(50) - immatriculation : VARCHAR(30) - date_mise_circulation : DATE

Pour voir la liste des tables d'une base de données, on peut exécuter la requête suivante **SHOW TABLES;**

# Modification de tables

Voici la syntaxe MySQL pour modifier une table déjà créée, afin d'y ajouter, de modifier ou de supprimer une colonne, un index ou une contrainte.

```
ALTER TABLE tbl_name [alter_specification [, alter_specification] ...]  
alter_specification:  
| ADD [COLUMN] (col_name column_definition [FIRST | AFTER col_name])  
| ADD {INDEX|KEY} [index_name] [idx_type] (idx_col,...) [idx_opt] ...  
| ADD [CONSTRAINT] [symbol] PRIMARY KEY [idx_type] (idx_col,...) [idx_opt] ...  
| ADD [CONSTRAINT] [symbol] UNIQUE {INDEX|KEY} [idx_name] [idx_type] (idx_col,...) [idx_opt] .  
| ADD [CONSTRAINT] [symbol] FOREIGN KEY [idx_name] (idx_col,...) reference_definition  
  
| CHANGE [COLUMN] old_col_name new_col_name column_definition  
  
| DROP [COLUMN] col_name  
| DROP {INDEX|KEY} index_name  
| DROP PRIMARY KEY  
| DROP FOREIGN KEY fk_symbol
```

Permet d'ajouter une colonne, un index, une clé primaire ou une clé étrangère

Permet de redéfinir une colonne afin de changer son nom, son type ou des contraintes portant sur cette colonne.

Permet de supprimer une colonne, un index, une clé primaire ou une clé étrangère.

--Exemple

```
ALTER TABLE vehicule  
DROP COLUMN modele,  
CHANGE COLUMN date_mise_circulation date_circulation DATE NOT NULL,  
ADD CONSTRAINT UNIQUE (immatriculation);
```

La redéfinition d'une colonne peut altérer tout ou partie des données.

# Suppression de tables (et son contenu)

Pour supprimer une table d'une base de données, on utilise la syntaxe suivante :

```
DROP TABLE [IF EXISTS]
tbl_name [, tbl_name] ...
```

--Exemple

```
DROP TABLE IF EXISTS vehicule;
```

Supprimer une table supprime évidemment également son contenu...

# Vider une table

**i** Pour supprimer les données d'une table tout en conservant sa structure, on utilise la syntaxe suivante :

**TRUNCATE [TABLE] tbl\_name**

--Exemple

**TRUNCATE TABLE vehicule;**

**i** **TRUNCATE** diffère d'un **DELETE** car la table est supprimée puis recréée, ce qui est plus rapide. Cependant, un **TRUNCATE** est impossible lorsque la table possède des clés étrangères. A noter que les séquences portant sur la table sont réinitialisées

# Insertion de données dans une table

❶ Pour insérer une ou plusieurs lignes dans une table, on utilise la syntaxe suivante :

```
INSERT [INTO] tbl_name [(col_name [, col_name] ...)]  
{VALUES | VALUE} (value_list) [, (value_list)] ...
```

value:

```
{expr | DEFAULT}
```

value\_list:

```
value [, value] ...
```

--Exemple

```
INSERT INTO vehicule (marque, modele, immatriculation, date_mise_circulation) VALUES  
( 'Peugeot', '208', 'AA-123-BB', '2017-10-02'),  
( 'Renault', 'Clio', 'DD-456-CC', '2017-10-01');
```

❶ Les valeurs non spécifiées sont affectées avec la valeur par défaut définie lors de la création de la table, ou `null`. Toutes les colonnes déclarées `NOT NULL` doivent être définies.

# Lecture des données des tables

**i** Pour récupérer des lignes d'une ou plusieurs tables, on utilise la syntaxe suivante :

**SELECT [DISTINCT]**

select\_expr [, select\_expr ...]

[**FROM** table\_references]

Sélection des tables sur lesquelles on va travailler

[**WHERE** where\_condition]

[**GROUP BY** col\_name [**ASC** | **DESC**], ... ]

Aggrégier les données (somme, moyenne, nombre d'éléments...) avec éventuellement une condition

[**HAVING** where\_condition]

[**ORDER BY** col\_name [**ASC** | **DESC**], ... ]

Limiter le nombre de résultats en sortie.

[**LIMIT** row\_count]

Pour ne récupérer que les données qui satisfont une condition

Ordonner les lignes selon une ou plusieurs colonnes dans l'ordre ascendant ou descendant

--Tous les véhicules BMW ordonné par modèle croissant

**SELECT \* FROM** vehicule

**WHERE** marque = 'BMW'

**ORDER BY** modele **ASC**;

--Récupère toutes les marques ayant au moins trois véhicules et affiche le nombre

**SELECT** marque, **count(\*)** **FROM** vehicule

**GROUP BY** marque

**HAVING** count(\*) > 2;

**i** Nous verrons plus tard comment réaliser des requêtes plus complexes avec des jointures et des sous-requêtes.

# Mise à jour de données

💡 Pour changer les valeurs d'une ou plusieurs colonnes des lignes d'une table, on utilise la syntaxe suivante :

**UPDATE** *table\_name*

**SET** *assignment\_list*

On définit ici les affectations à faire sur les lignes sélectionnées.

**WHERE** *where\_condition*

On peut aussi ne mettre à jour que les données qui satisfont une condition

**LIMIT** *row\_count*

Enfin, on peut limiter le nombre d'éléments mis à jour.

**value:**

{*expr* | **DEFAULT**}

**assignment:**

*col\_name* = **value**

**assignment\_list:**

*assignment* [, *assignment*] ...

--Met à jour tous les véhicules de marque 'bmw' pour mettre la marque en majuscule

**UPDATE** *vehicule* **SET** *marque* = '**BMW**' **WHERE** *marque* = '**bmw**';

💡 Évidemment, les mises à jour doivent respecter les contraintes définies sur les colonnes.

# Suppression de données

💡 Pour supprimer tout (on peut alors plutôt utiliser **TRUNCATE**) ou partie des données d'une table, on utilise la syntaxe suivante :

**DELETE FROM** *tbl\_name*

On ne supprime que les X premiers éléments.

[**WHERE** *where\_condition*] [**LIMIT** *row\_count*]

On ne supprime que les éléments qui satisfont une condition.

--*Suppression de tous les véhicules dont la date de mise en circulation est antérieure au 1er janvier 2017.*

**DELETE FROM** *vehicule* **WHERE** *date\_mise\_circulation < '2017-01-01'*;

💡 On ne peut pas supprimer les lignes qui sont référencées par une clé étrangère d'une autre table.

# Exécuter un fichier SQL

**i** Exécuter des requêtes une par une n'est pas adapté pour la création de toute une base de données. Il est possible de mettre plusieurs requêtes dans un fichier et de le faire exécuter séquentiellement par le serveur.

```
--Fichier create.sql
--Création de la table véhicule
CREATE TABLE vehicule ... ;

--Insertion des valeurs
INSERT INTO vehicule (...) VALUES (...);
INSERT INTO vehicule (...) VALUES (...);
INSERT INTO vehicule (...) VALUES (...);

...
--Depuis un shell
mysql mabase < create.sql

--Depuis le client mysql
source create.sql
\c create.sql
```

**i** Pour l'exécution de gros fichiers SQL, on préférera l'utilitaire `mysqlimport` qui est plus rapide.

# Les contraintes d'intégrité

**i** Il est possible de contraindre certaines colonnes pour assurer leur non nullité, leur unicité, ou pour préciser leur lien avec des colonnes d'autres tables.

## Clé primaire

Une clé primaire permet d'identifier une ligne de manière unique dans une table. Il ne peut y en avoir qu'une par table et à la différence d'une clé unique, une clé primaire ne peut prendre la valeur `NULL`.

## Clé unique

Une contrainte `UNIQUE` permet d'assurer l'unicité d'une colonne au sein d'une table (la valeur `NULL` est autorisée et n'a pas besoin d'être unique). Ainsi, il ne sera pas possible d'insérer deux véhicules avec la même immatriculation. A noter que la création d'une contrainte `UNIQUE` crée automatiquement un index sur

## Clé étrangère

Soit la table `vehicule` avec le champ `proprietaire_id` et la table `proprietaire` avec la clé primaire `id`. On peut définir un lien entre les deux colonnes à l'aide d'une clé étrangère. Lors de l'insertion d'un véhicule, le champ `proprietaire_id` devra référencer un propriétaire existant.

-Ajout dans la table `vehicule` d'une clé primaire sur le champ `id`

`ALTER TABLE vehicule ADD PRIMARY KEY(id);`

-Suppression de la clé primaire

`ALTER TABLE vehicule DROP PRIMARY KEY;`

-Ajout d'une contrainte d'unicité sur l'immatriculation

`ALTER TABLE vehicule ADD CONSTRAINT immat_unique UNIQUE(immatriculation);`

-Suppression de la contrainte

`ALTER TABLE vehicule DROP INDEX immat_unique;`

-Ajout d'une clé étrangère entre `vehicule` et `proprietaire`

`ALTER TABLE vehicule ADD CONSTRAINT fk_vehicule_proprietaire  
FOREIGN KEY (proprietaire_id) REFERENCES proprietaire(id);`

-Suppression de la clé étrangère

`ALTER TABLE vehicule DROP FOREIGN KEY fk_vehicule_proprietaire;`

**i** Une contrainte d'intégrité peut porter sur plusieurs champs. On peut déclarer ces contraintes à la création de la table, ou via un `ALTER TABLE`.

# Les index

❶ Les **index** sont utilisés pour trouver des lignes avec des valeurs spécifiques plus rapidement. Sans index, MySQL doit commencer avec la première ligne et lire la table entièrement. Les recherches portant sur des colonnes indexées sont donc plus rapides. En MySQL, **KEY** et **INDEX** sont équivalents.

```
CREATE [UNIQUE] INDEX index_name  
[index_type]  
ON tbl_name (index_col_name, ...)
```

index\_col\_name:

col\_name [(length)]

Pour les colonnes contenant des chaînes de caractères, il est possible d'indexer uniquement les X premiers caractères en spécifiant la longueur.

--Index sur la marque et le modèle d'un véhicule

```
CREATE INDEX idx_marque ON vehicule(marque, modele);
```

--Suppression de l'index

```
DROP INDEX idx_marque ON vehicule;
```

--Index les deux premiers caractères de l'immatriculation

```
CREATE INDEX idx_immat ON vehicule(immatriculation(2));
```

⚠️ Attention, au plus il y a d'index sur une table, au plus les opérations de création et de mises à jour sur cette dernière prennent du temps (surtout sur les tables comportant beaucoup de lignes). Il s'agit donc de trouver l'équilibre entre insertion et recherche.

# La notion de transaction

Les transactions permettent d'exécuter un ensemble de requête en un seul bloc. Cela permet d'annuler cet ensemble lorsqu'une des requêtes échoue évitant ainsi d'effectuer une opération à moitié.

**SET autocommit=0;**

Instruction pour démarrer une nouvelle transaction. Celle-ci se terminera au premier **COMMIT** ou **ROLLBACK**

Par défaut, chaque transaction exécutée est automatiquement validée (commit). On peut soit désactiver ce comportement (pour la session en cours uniquement) soit démarrer manuellement une transaction.

**START TRANSACTION;**

**COMMIT;**

Permet de valider toutes les requêtes de la transaction en cours.

Permet d'annuler toutes les requêtes de la transaction en cours.

**ROLLBACK;**

--Exemple : Début de transaction

**START TRANSACTION;**  
**INSERT INTO** vehicule ...;  
**INSERT INTO** vehicule ...;

...

--Validation de toutes les requêtes

**COMMIT;**

Il n'est pas possible d'utiliser les transactions pour les opérations **CREATE/DROP DATABASE**, **CREATE/DROP/ALTER/RENAME TABLE**, **CREATE/DROP INDEX** et de manière générale tout ce qui touche à la structure de la base de données.

# Requêtes complexes

 Il est parfois nécessaire de réaliser des requêtes plus complexes :

Jointures

Sous-requêtes

Union

# Jointures

**i** Les jointures permettent d'effectuer des requêtes sur deux tables liées généralement par une clé étrangère.

--Jointure interne

```
SELECT * FROM vehicule v, proprietaire p
INNER JOIN proprietaire p ON v.proprietaire_id = p.id;
```

--Jointure externe à gauche

```
SELECT * FROM vehicule v, proprietaire p
LEFT JOIN proprietaire p ON v.proprietaire_id = p.id;
```

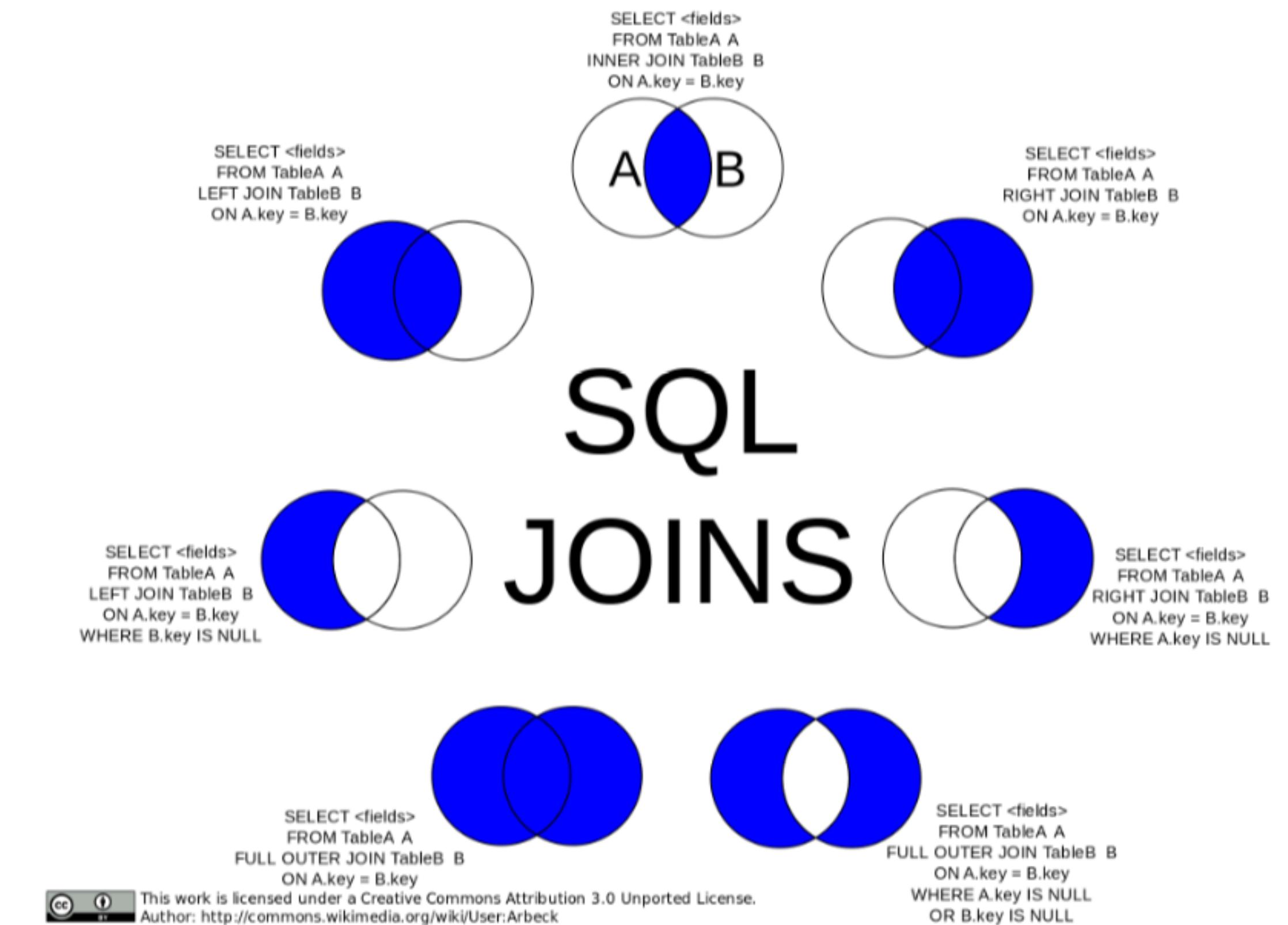
--Jointure externe à droite

```
SELECT * FROM vehicule v, proprietaire p
RIGHT JOIN proprietaire p ON v.proprietaire_id = p.id;
```

--Jointure externe non supportée en MySQL

--Autres syntaxes lorsque les champs des deux tables ont le même nom, ici `prop_id`.

```
SELECT * FROM vehicule INNER JOIN proprietaire USING (prop_id);
SELECT * FROM vehicule NATURAL JOIN proprietaire;
```



**i** Il est possible de faire manuellement une jointure avec une clause `WHERE` mais cela est moins performant et la requête peut être moins lisible lorsqu'il y a d'autres clauses `WHERE`.



This work is licensed under a Creative Commons Attribution 3.0 Unported License.  
Author: http://commons.wikimedia.org/wiki/User:Arbeck

# Sous-requêtes

**i** Il est possible d'imbriquer plusieurs requêtes pour effectuer des opérations poussées d'un seul coup.

--Propriétaire ayant une BMW et une Peugeot

```
SELECT p.* FROM proprietaire p
INNER JOIN vehicule v ON v.proprietaire_id = p.id
WHERE v.marque = 'BMW'
AND EXISTS (
    SELECT p2.* FROM proprietaire p2
    INNER JOIN vehicule v2 ON v2.proprietaire_id = p2.id
    WHERE p2.id = p.id
    AND v2.marque = 'Peugeot'
);
```

--Propriétaire le + jeune vs propriétaire de BMW le + jeune

```
SELECT MAX(prop.date_naissance), MAX(p.dtN)
FROM proprietaire prop,
     (SELECT p2.date_naissance AS dtN FROM proprietaire p2
      INNER JOIN vehicule v2 ON v2.proprietaire_id = p2.id
      AND v2.marque = 'BMW'
     ) AS p;
```

--On supprime les propriétaires de Lada

```
DELETE FROM proprietaire p
WHERE p.id IN (
    SELECT v.proprietaire_id FROM vehicule v
    WHERE v.marque = 'Lada'
);
```

--Les propriétaire de Ferrari sont chanceux...

```
UPDATE test.proprietaire SET commentaires = 'Chanceux !'
WHERE id = (
    SELECT v.proprietaire_id FROM test.vehicule v
    WHERE v.marque = 'Ferrari'
);
```

--Les trentenaires ont droit à une Ferrari !

```
INSERT INTO vehicule (marque, modele, proprietaire_id)
SELECT 'Ferrari', 'F12', id AS proprietaire_id
FROM proprietaire WHERE YEAR(date_naissance) = 1987;
```

**i** Les sous-requêtes ne sont possibles que pour des **SELECT**, **UPDATE**, **DELETE** ou **INSERT**.

# UNION

- ❶ Il est possible de joindre les résultats de deux requêtes en utilisant **UNION** à condition que le nombre de colonnes (ainsi que leur type et l'ordre d'affichage) entre les différents **SELECT** soit identique.

```
--Propriétaires de Citroën et/ou de Peugeot
SELECT p.* FROM test.proprietaire p
INNER JOIN test.vehicule v ON v.proprietaire_id = p.id
WHERE v.marque = 'Citroën'
UNION
SELECT p.* FROM test.proprietaire p
INNER JOIN test.vehicule v ON v.proprietaire_id = p.id
WHERE v.marque = 'Peugeot'
```

- ❷ L'intersection entre deux **SELECT** n'est pas supporté en MySQL. Il faut passer par une clause de type **WHERE EXISTS**.

# Vues

Les vues sont des objets de la base de données permettant de stocker une requête SELECT particulière, ce qui est intéressant pour une requête assez complexe utilisée de nombreuses fois. Une fois créée, la vue se comporte comme une table.

On peut préciser le nom de chaque colonne de la vue lorsque des conflits sont possibles (lors de jointures par exemple). Sinon, on peut également utiliser des alias dans la requête SELECT.

**CREATE [OR REPLACE] VIEW view\_name  
(column\_list)  
AS select\_statement**

Presque n'importe quelle requête SQL peut être utilisée (voir [ici](#) les quelques restrictions).

--Exemple

```
CREATE VIEW proprietaire_vehicule (nom, prenom, marque, modele, immatriculation) AS
SELECT p.nom, p.prenom, v.marque, v.modele, v.immatriculation FROM proprietaire p
INNER JOIN vehicule v ON v.proprietaire_id = p.id;
```

```
SELECT * FROM proprietaire_vehicule WHERE marque = 'BMW';
```

nom	prenom	marque	modele	immatriculation
Girard	Isabelle	BMW	Série 4	BB-456-CC
...	...	...	...	...
...	...	...	...	...

A noter qu'il n'y a pas de gain de performance en termes de temps d'exécution des requêtes par rapport à exécuter directement le SELECT. Il est également possible d'effectuer des INSERT, UPDATE ou DELETE sur certaines vues mais cela est déconseillé.

# Expressions et fonctions

**💡 Il est possible d'utiliser des expressions ou des fonctions pour effectuer des requêtes plus complexes :**

Expression ou fonction	Description	Syntaxe
SUM , AVG	Calcule la somme ou la moyenne de valeurs numériques	<code>SELECT AVG(prix), SUM(prix) FROM vehicule</code>
MIN , MAX	Récupère le minimum ou le maximum d'une ensemble de valeurs	<code>SELECT MIN(date_mise_circulation), MAX(date_mise_circulation) FROM vehicule</code>
CEIL , FLOOR , ROUND	Effectue un arrondit (inférieur, supérieur et normal) sur une valeur numérique	<code>SELECT FLOOR(3.14), CEIL(3.14), ROUND(3.14) --3, 4, 3</code>
CONCAT , UPPER , LOWER , SUBSTRING	Concatène, met en majuscule ou minuscule, extrait une partie d'une chaîne	<code>SELECT CONCAT(UPPER('hello'), LOWER(' WORLD '), SUBSTRING('!ZZ',1,1)) --HELLO world !</code>
LIKE , REGEXP	Teste le contenu d'une chaîne de caractère	<code>SELECT * FROM vehicule WHERE immatriculation LIKE 'AA%' AND modele REGEXP '^Série [0-9]{1}\$'</code>
CURDATE ou CURTIME	Récupère la date ou l'heure courante	<code>SELECT CURDATE(), CURTIME() --2017-10-16, 14:27:48</code>
YEAR , MONTH , DAYOFMONTH	Extrait l'année, le mois ou le jour d'une date	<code>SELECT YEAR(CURDATE()), MONTH(CURDATE()), DAYOFMONTH(CURDATE()) --2017, 10, 16</code>
IF	Teste une condition	<code>SELECT IF(marque = 'Ferrari', 'Riche', 'Moins riche') FROM vehicule</code>
IFNULL , NULLIF	Renvoie une valeur si l'expression est nulle, null si la valeur est égale à une autre	<code>SELECT IFNULL(proprietaire_id, 'Sans propriétaire'), NULLIF(marque, 'Lada') FROM vehicule</code>
CASE	Renvoie une valeur en fonction de cas différents	<code>SELECT CASE marque WHEN 'Ferrari' THEN 'Riche' WHEN 'Lada' THEN 'Lol' ELSE '?' END FROM vehicule</code>

**💡 Il y en a beaucoup d'autres que l'on peut voir en intégralité [ici](#).**

# Serveur

**ⓘ Au lieu d'être simple exécutant de requêtes fournies par le client, on peut stocker des traitements directement au niveau du serveur :**

Procédures stockées

Fonctions

Triggers

Requêtes préparées

**ⓘ Cela s'apparente alors à de la programmation procédurale.**

# Flot d'exécution et variables

**i** Il est donc possible de gérer le flot d'exécution d'un enchaînement de traitements

Element	Description	Syntaxe	Element	Description	Syntaxe
Variables	Une variable est définie par un type et éventuellement une valeur par défaut.	<code>DECLARE monEntier INT DEFAULT 0; SET monEntier = 1;</code>	LOOP et LEAVE et ITERATE	Exécute les traitements tant qu'il n'y a pas une sortie explicite de la boucle avec un <code>LEAVE</code> ou un <code>RETURN</code> par exemple.	<code>etiquette: LOOP IF monEntier = 0 THEN   LEAVE etiquette; ELSE   ITERATE etiquette; END IF; SET x = 1; END LOOP;</code>
CASE	Exécute les traitements en fonction de conditions	<code>CASE monEntier   WHEN 0 THEN ...   WHEN 1 THEN ...   ELSE ... END CASE;</code>			
IF	Exécute les traitements en fonction de conditions	<code>IF monEntier &gt; 0 THEN   ... ELSEIF monEntier &lt; 0 THEN   ... ELSE   ... END IF;</code>	WHILE	Exécute les traitements tant qu'une condition n'est pas atteinte	<code>WHILE x &gt; 0 DO   ... END WHILE;</code>

**i** On utilise ces éléments dans le cadre de fonctions, de procédures stockées ou de triggers par exemple.

# Procédures stockées

❶ Les procédures stockées représentent des ensembles nommés de requêtes SQL qui sont stockées dans la base. Lorsqu'elles sont appelées, les requêtes SQL dont elles sont composées sont exécutées.

```
CREATE PROCEDURE sp_name ([proc_parameter[, ...]]) routine_body
```

```
proc_parameter:
```

```
[ IN | OUT | INOUT ] param_name type
```

```
routine_body:
```

```
Requête(s) SQL valides
```

Changement du délimiteur

--Exemple

```
DELIMITER |
```

Début du corps de la procédure

BEGIN est utilisé pour signaler le début du corps de la procédure.

Fin du corps de la procédure

END suivi du délimiteur précédemment défini est utilisé pour signaler la fin de la requête de création de la procédure.

```
CREATE PROCEDURE supprime_vehicule_sans_immat(OUT nb_immat_null INT)
```

```
BEGIN
```

```
SELECT COUNT(*) INTO nb_immat_null FROM vehicule WHERE immatriculation IS NULL;
```

```
DELETE FROM vehicule WHERE immatriculation IS NULL;
```

```
END|
```

```
DELIMITER ;
```

```
ALL supprime_vehicule_sans_immat(@nb_immat_null);
```

```
SELECT @nb_immat_null;
```

Restauration du délimiteur

On remet le délimiteur à sa valeur par défaut, ;

Nom et paramètres de la procédure

On définit le nom de la procédure, ainsi que ces éventuels paramètres. Ici, un paramètre de sortie de type INT nommé nb.

Corps de la procédure

Le corps de la procédure contient les requêtes SQL à exécuter lors de l'appel de la procédure stockée. Ici, on compte le nombre de véhicules avec une immatriculation non définie (que l'on stocke dans la variable de sortie nb\_immat\_null) et on les supprime.

❷ Pour supprimer une procédure stockée, on utilise `DROP PROCEDURE supprime_vehicule_sans_immat;`

# Fonctions

❶ Une fonction est également un ensemble de requêtes nommé mais avec que des paramètres en entrée et un seul paramètre en sortie. On ne les appelle pas avec `CALL`, mais on les référence directement dans une requête.

```
CREATE FUNCTION sp_name ([func_parameter[,...]]) RETURNS type routine_body
  func_parameter: param_name type
  type:           Type SQL valide
  routine_body:   Requêtes SQL valides
```

--Exemple

```
DELIMITER |
CREATE FUNCTION format_immat(t VARCHAR(50)) RETURNS VARCHAR(10)
BEGIN
  IF t REGEXP '^[a-zA-Z]{2}[0-9]{3}[a-zA-Z]{2}$' THEN
    RETURN UPPER(CONCAT(SUBSTR(t,1,2), '-', SUBSTR(t,3,3), '-', SUBSTR(t,5,2)));
  ELSE
    RETURN 'INCORRECT';
  END IF;
END|
DELIMITER ;
```

## Corps de la fonction

On utilise `RETURN` pour retourner la valeur de sortie de la fonction. Ici si l'immatriculation est au format `aa999aa`, on la retourne formatée : `AA-999-AA`, sinon on renvoie `'INCORRECT'`. Pour toutes les fonctions sur les chaînes de caractère, voir [ici](#).

## Nom et paramètres

On définit le nom de la fonction, les paramètres en entrée facultatifs (nom et type) et le paramètre de sortie obligatoire (nom et type).

```
SELECT format_immat('aa123bb');
--affiche AA-123-BB
```

❶ Pour supprimer une fonction, on utilise la syntaxe `DROP FUNCTION format_immat;`

# Triggers

**i** Les **triggers** (déclencheurs) sont un ensemble de requêtes SQL exécutées avant ou après chaque insertion, suppression ou modification des lignes d'une table.

## Moment de déclenchement

On peut choisir de déclencher le trigger avant (lorsqu'on veut effectuer des vérifications par exemple) ou après (lorsqu'on veut journaliser ce qui vient de se passer par exemple) l'événement.

## Événement de déclenchement

L'événement déclencheur du trigger peut être soit une insertion, soit une modification soit une suppression. Le trigger s'exécute autant de fois que l'événement survient. Si une requête supprime trois lignes, un trigger `DELETE` s'exécutera pour chacune des trois lignes.

```
CREATE TRIGGER trigger_name trigger_time trigger_event ON tbl_name FOR EACH ROW
[trigger_order] trigger_body
```

`trigger_time`: { `BEFORE` | `AFTER` }

`trigger_event`: { `INSERT` | `UPDATE` | `DELETE` }

`trigger_order`: { `FOLLOWS` | `PRECEDES` } other\_trigger\_name

`trigger_body`:

Requête(s) **SQL** valides

Corps du trigger

--Exemple

DELIMITER |

```
CREATE TRIGGER commentaire_insert BEFORE UPDATE ON vehicule FOR EACH ROW
```

BEGIN

SET NEW.commentaire = CONCAT('Immat précédente : ', OLD.immatriculation);

END|

DELIMITER ;

```
INSERT INTO vehicule (...) VALUES ('Peugeot', '208', 'AA-123-BB', '2017-10-02');
```

```
UPDATE vehicule SET immatriculation = 'BB-456-CC' WHERE id = 5;
```

--dans le champ commentaire : Immat précédente : AA-123-BB

## Ordre de déclenchement

Lorsque plusieurs triggers sont définis sur un même événement, on peut définir l'ordre d'exécution des triggers en spécifiant s'il doit s'exécuter avant ou après

**i** Pour supprimer un trigger, on utilise la syntaxe `DROP TRIGGER commentaire_insert;`.

# Utilisation des fonctions, triggers et procédures stockées

**i** Ces techniques sont à utiliser judicieusement et il ne faut pas en abuser. Voici les éléments à considérer avant d'utiliser telle ou telle technique :

Questions ou reflexions	Triggers	Procédures stockées	Fonctions
<i>Il est important que ce code soit portable, testable et versionné</i>			
<i>Ce code effectue des traitements purement fonctionnels</i>			
<i>Ce code doit être évolutif</i>			
<i>Ce code n'est utilisé qu'à un seul endroit</i>			
<i>L'ensemble des clients est maîtrisé</i>			
<i>Le serveur a une capacité de calcul limitée</i>			
<i>Ce code est utilisé par plusieurs clients différents</i>			
<i>Ce code effectue des traitements purement techniques</i>			
<i>Le client a une capacité de calcul limitée</i>			
<i>Vous ne maîtrisez pas tous les clients</i>			

**i** Ces considérations sont subjectives et dépendent de la philosophie adoptée dans le projet et des contraintes techniques.

# Requêtes préparées

**i** Les requêtes préparées sont des requêtes SQL stockées au niveau du serveur pour la session en cours.

**PREPARE** stmt\_name **FROM** preparable\_stmt

Presque n'importe quelle requête peut être utilisée, avec ou sans paramètre (on utilise le ? pour symboliser un paramètre dans la requête).

**EXECUTE** stmt\_name [**USING** @var\_name [, @var\_name] ...]

Lorsqu'on exécute une requête préparée, on peut passer des variables en paramètres de la requête en utilisant **USING**.

**DEALLOCATE PREPARE** stmt\_name

On peut supprimer manuellement une requête préparée avec **DEALLOCATE PREPARE**. Sinon, cela serait fait automatiquement en fin de session.

--Exemple

**PREPARE** vehicule\_par\_marque\_modele **FROM**

**SELECT** \* **FROM** vehicule **WHERE** marque = ? **AND** modele = ?;

**SET** @marque = 'BMW'; **SET** @modele = 'Série 3';

**EXECUTE** vehicule\_par\_marque\_modele **USING** @marque, @modele;

**SET** @modele = 'Série 5';

**EXECUTE** vehicule\_par\_marque\_modele **USING** @marque, @modele;

**DEALLOCATE** vehicule\_par\_marque\_modele;

**i** Les requêtes préparées sont utilisées de manière directe par les développeurs. Cependant, de nombreuses librairies *clients* de MySQL s'appuie sur ce principe lorsqu'on exécute plusieurs fois les mêmes requêtes avec des paramètres différents.

# Importer ou exporter une base

Il est souvent nécessaire d'effectuer des imports ou des exports de base pour effectuer des sauvegardes ou des restaurations de données par exemple. On peut utiliser des clients (MySQL Workbench) pour des besoins ponctuels et sur des petits volumes. Sinon, on peut utiliser `mysqldump` dans une console.

```
mysqldump [options] db_name [tbl_name ...]
mysqldump [options] --all-databases

#Exemples
#Exporter la base db_name dans un fichier SQL
mysqldump db_name > backup-file.sql

#Importer un dump dans une base
mysqlimport -u root -p db_name backup-file.sql
```

Pour la liste complète des options, voir [ici](#).

# Bonus

 Quelques fonctionnalités avancées de MySQL

Clusters

# Curseurs

❶ Les curseurs permettent de parcourir les résultats d'un `SELECT` afin d'effectuer des traitements sur ces derniers.

```
DECLARE cursor_name CURSOR FOR select_statement
OPEN cursor_name
FETCH [[NEXT] FROM] cursor_name INTO var_name [, var_name] ...
LOSE cursor_name
```

Ouverture du curseur obligatoire avant son utilisation.

Fermeture du curseur lorsqu'on ne l'utilise plus

Déclaration du curseur avec définition de la requête `SELECT` dont il sera constitué.

Récupération de la prochaine valeur revoyée par le curseur. On utilise typiquement le `FETCH` au sein d'une boucle.

--Exemple

```
CREATE PROCEDURE ex_vehicules() BEGIN
    DECLARE v_id, v_incr, v_fin INT DEFAULT 0;
    DECLARE c_vehicules CURSOR FOR SELECT id FROM vehicule ORDER BY immatriculation;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_fin = 1;
    OPEN c_vehicules;
    loop_vehicules: LOOP
        SET v_incr = v_incr + 1;
        FETCH FROM c_vehicules INTO v_id;
        IF v_fin = 1 THEN LEAVE loop_vehicules; END IF;
        UPDATE vehicule SET commentaire = CONCAT('Véhicule n°', v_incr) WHERE id = v_id;
    END LOOP;
    CLOSE c_vehicules;
END|
CALL ex_vehicules;
```

Afin de pouvoir sortir de la boucle lorsqu'il n'y a plus d'éléments, on peut utiliser un handler définissant un comportement lorsqu'il n'y a plus de résultats dans le curseur. Ici, on définit la variable `v_fin` à 1.

Le champ commentaire aura pour valeur 'Véhicule n°X' avec X le numéro d'ordre du `SELECT` (immatriculation croissante)

❷ Les curseurs sont relativement peu utilisés directement, la syntaxe étant peu claire et les traitements possibles en MySQL sur les données limités.

# Moteurs de stockage

