



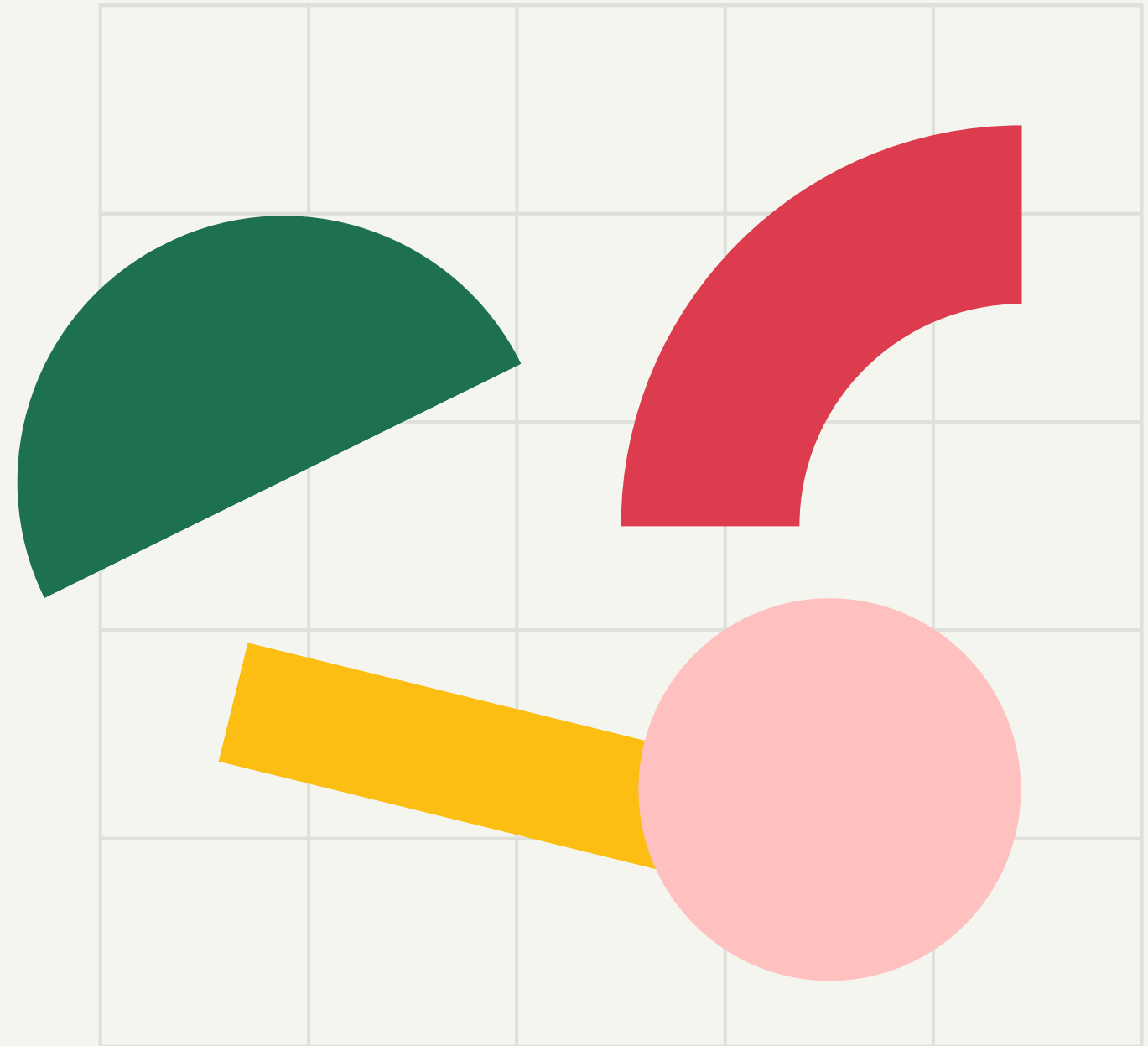
# Initiation MySQL

LASNAÏ SARA

M o n c e f      B e n h a d j

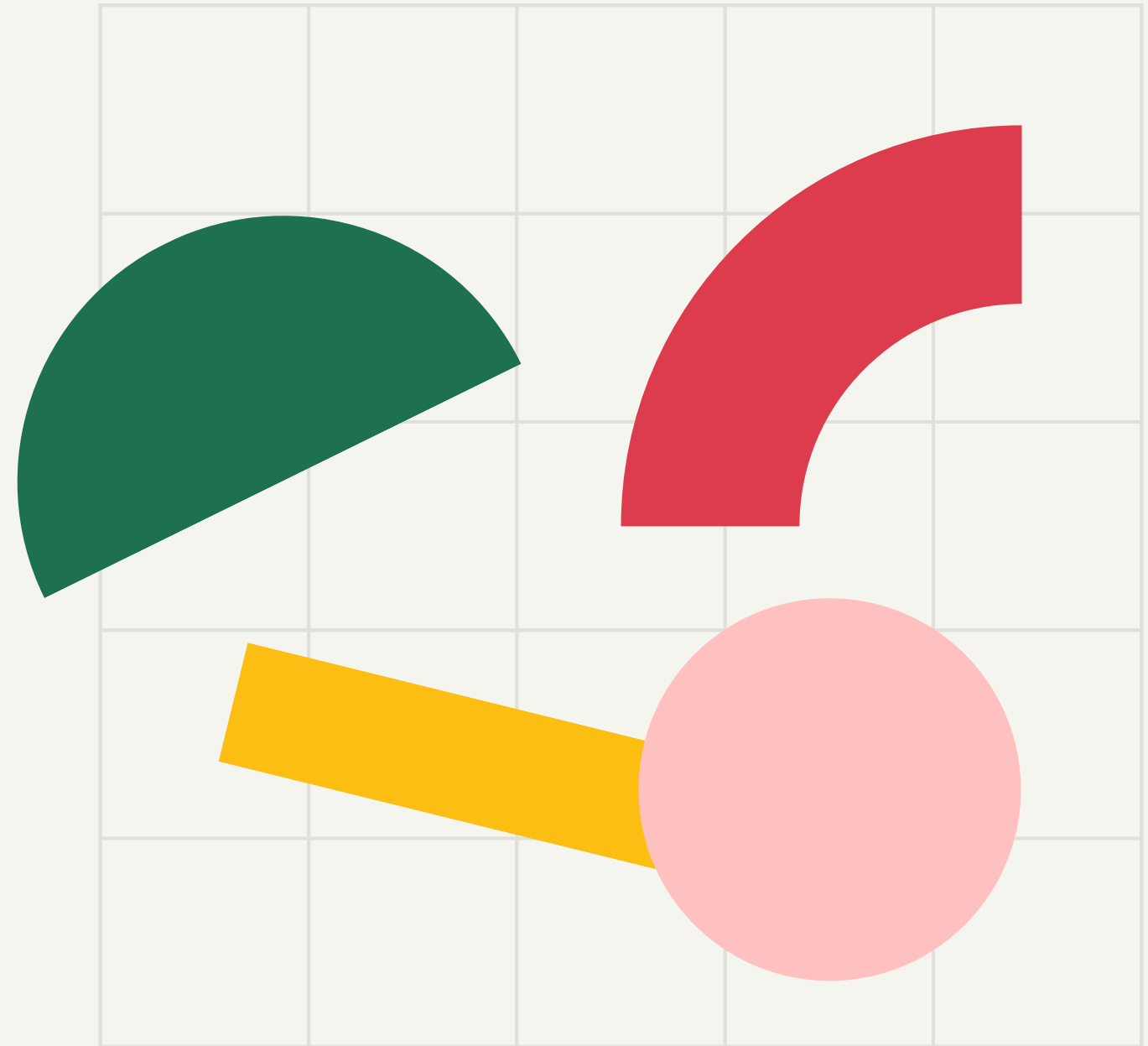
# Présentation:

- Nom
- Prénom
- Expérience en informatique (Expl : formation, langages maîtrisés..)
- Qu'est ce que une Base de données ?



# But du cours :

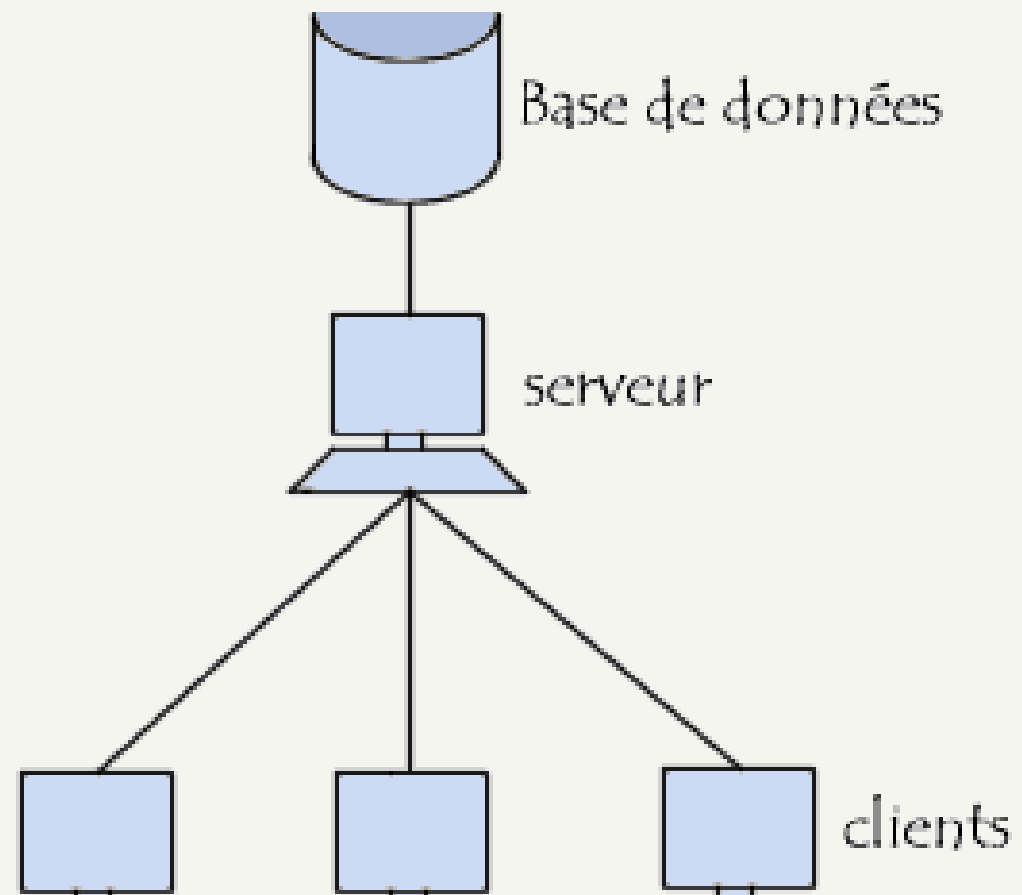
- Comprendre le fonctionnement et savoir utiliser les données
- manipuler le langage SQL.
- Savoir créer et configurer une base de données *MySQL*
- Savoir sauvegarder et restaurer les données



# I - INTRODUCTION

# Base de données

**Une base de données est un ensemble d'informations qui est organisé de manière à être facilement accessible, géré et mis à jour. Elle est utilisée par les organisations comme méthode de stockage, de gestion et de récupération de l'informations**



# C'est quoi un SGBD

**Un système de gestion de base de données (SGBD) est un logiciel système permettant aux utilisateurs et programmeurs de créer et de gérer des bases de données.**



# Présentation de SQL

**SQL signifie "Structured Query Language" c'est-à-dire "Langage d'interrogation structuré". En fait SQL est un langage complet de gestion de bases de données relationnelles. Il a été conçu par IBM dans les années 70.**

**-Il est devenu le langage standard des systèmes de gestion de bases de données (SGBD) relationnelles (SGBDR). C'est à la fois :**

- **un langage d'interrogation de la base**
- **un langage de manipulation des données**
- **un langage de définition des données**
- **un langage de contrôle de l'accès aux données**



# Normes SQL

SQL a été normalisé dès 1986 mais les premières normes, trop incomplètes, ont été ignorées par les éditeurs de SGBD.

## La norme SQL2

(appelée aussi SQL92) date de 1992. Le niveau est à peu près respecté par tous les SGBD relationnels qui dominent actuellement le marché.

SQL-2 définit trois niveaux :

- Full SQL (ensemble de la norme)
- Intermediate SQL
- Entry Level

## SQL3 (appelée aussi SQL99)

**est la nouvelle norme SQL. Malgré ces normes, il existe des différences non négligeables entre les syntaxes et fonctionnalités des différents SGBD. En conséquence, écrire du code portable n'est pas toujours simple dans le domaine des bases de données.**





# Identificateurs

**SQL utilise des identificateurs pour désigner les objets qu'il manipule : utilisateurs, tables, colonnes, index, fonctions, etc...**

**Un identificateur est un mot formé de caractères, commençant obligatoirement par une lettre de l'alphabet.**

**Les caractères suivants peuvent être une lettre, un chiffre, ou l'un des symboles # \$ et \_.**

**SQL ne fait pas la différence entre les lettres minuscules et majuscules. (Mais MySQL sous linux oui!!! Alors que sous windows non.)**

**Les voyelles accentuées ne sont pas acceptées. (Même si MySQL les acceptent, évitez absolument!)**



**Un identificateur ne doit pas figurer dans la liste des mot clés réservés. Voici quelques mots clés que l'on risque d'utiliser comme identificateurs : ASSERT, ASSIGN, AUDIT, COMMENT, DATE, DECIMAL, DEFINITION, FILE, FORMAT, INDEX, LIST, MODE, OPTION, PARTITION, PRIVILEGES, PUBLIC, REF, REFERENCES, SELECT, SEQUENCE, SESSION, SET, TABLE, TYPE, NAME...**

**Avec MySQL les mots réservés sont employés comme nom d'un identificateur (table, champ, base de données, etc...) sont "échappés" avec l'apostrophe inversé**

**Exemple :**

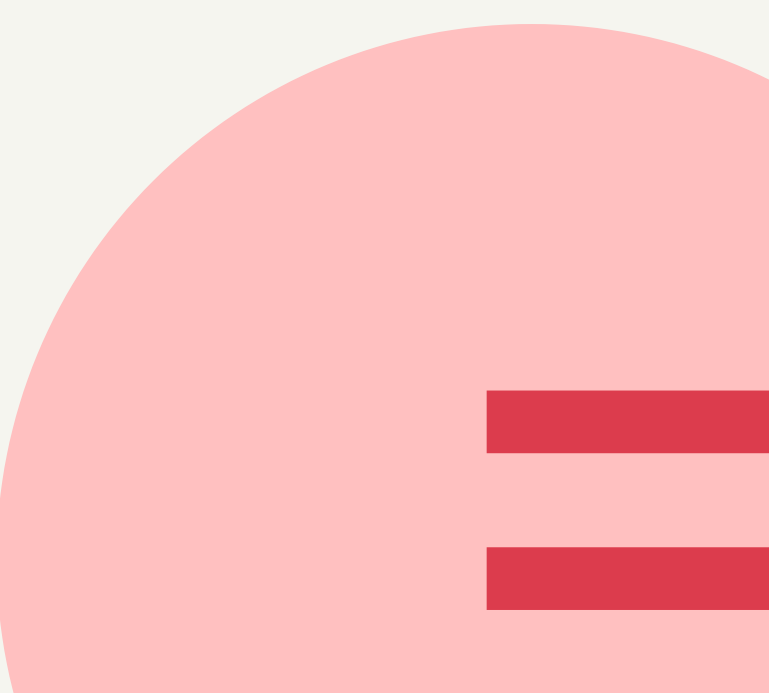

```
CREATE TABLE `user` (  
  id SERIAL,  
  name VARCHAR(255)  
);
```

## II - Les tables



**Les relations (d'un schéma relationnel) sont stockées sous forme de tables composées de lignes et de colonnes.**

**Il est d'usage (mais non obligatoire évidemment) de mettre les noms de table au singulier : plutôt EMPLOYE que EMPLOYES pour une table d'employés.**



EMPLOYEE
ID : NUMBER
F_NAME : VARCHAR
L_NAME : VARCHAR
SALARY : NUMBER
MANAGE_ID : NUMBER
ADDRESS_ID : NUMBER

# Colonnes

**Les données contenues dans une colonne doivent être toutes d'un même type de données.**

**Ce type est indiqué au moment de la création de la table qui contient la colonne.**

**Chaque colonne est repérée par un identificateur unique à l'intérieur de chaque table.**

**Deux colonnes de deux tables différentes peuvent porter le même nom.**



**Il est ainsi fréquent de donner le même nom à deux colonnes de deux tables différentes lorsqu'elles correspondent à une clé étrangère à la clé primaire référencée.**

**Une colonne peut porter le même nom que sa table.**

**Le nom complet d'une colonne est en fait celui de sa table, suivi d'un point et du nom de la colonne. Par exemple, la colonne EMPLOYEE.ID**

**Le nom de la table peut être omis quand il n'y a pas d'ambiguïté sur la table à laquelle elle appartient, ce qui est généralement le cas.**

# 3 - Les types de données

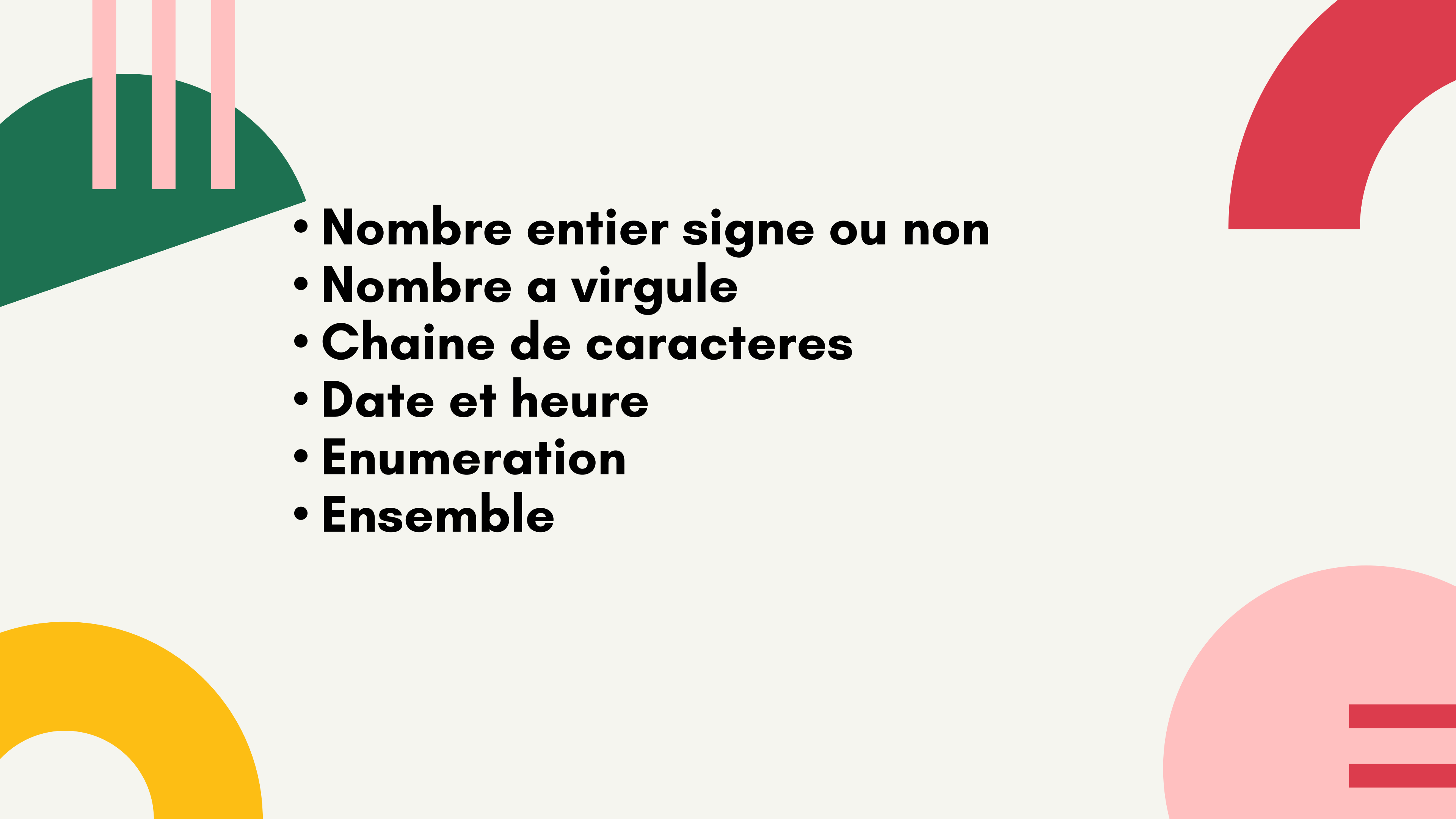


**Nous avons vu dans l'introduction qu'une base de données contenait des tables qui, elles-mêmes sont organisées en colonnes, dans lesquelles sont stockées des données. En SQL (et dans la plupart des langages informatiques), les données sont séparées en plusieurs types (par exemple : texte, nombre entier, 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.**



- 
- **Nombre entier signe ou non**
  - **Nombre a virgule**
  - **Chaine de caracteres**
  - **Date et heure**
  - **Enumeration**
  - **Ensemble**





## Nombres entiers

Les types de données qui acceptent des nombres entiers comme valeur 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

Pour les Nombres entiers non signé c'est à dire supérieur à zéro : **Unsigned**

on peut limiter la taille d'affichage avec l'attribut **ZEROFILL**. Il est possible de préciser le nombre de chiffres minimum à l'affichage d'une colonne de type INT. Il suffit alors de préciser ce nombre entre parenthèses : **INT(x)**.




exemple : **INT(4) ZEROFILL** ( si on a 45 stocké on aura 0045 dans l'affichage )



## FLOAT, DOUBLE et REAL

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

Quant à REAL et DOUBLE, ils ne supportent pas de paramètres. DOUBLE est normalement plus précis que REAL (stockage dans 8 octets contre stockage dans 4 octets), mais ce n'est pas le cas avec MySQL qui utilise 8 octets dans les deux cas.



Je vous conseille donc d'utiliser **DOUBLE** pour éviter les surprises en cas de changement de SGBDR.



## CHAR et VARCHAR

# Chaînes de type texte



**Pour stocker un texte relativement court (moins de 255 octets), 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 octets, en remplissant si nécessaire le texte avec des espaces vides pour le compléter, tandis qu'un VARCHAR(x) stockera jusqu'à x octets (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.**





## **TEXT / MEDIUMTEXT / LONGTEXT**

**Une colonne TEXT avec une longueur maximum de 65,535 ( $2^{16} - 1$ ) caractères (65k).**

**MEDIUMTEXT est une colonne TEXT avec une longueur de 16MB ( $2^{24} - 1$ ) caractères.  
(coût 3 bytes)**

**LONGTEXT est une colonne TEXT avec une longueur de 4GB ( $2^{32} - 1$ ) caractères. (coût 4 bytes)**

## **BLOB / MEDIUMBLOB / LONGBLOB**

**Un BLOB est un objet binaire de taille variable.**



# type ENUM

**Une colonne de type ENUM est une colonne pour laquelle on définit un certain nombre de valeurs autorisées, de type "chaîne de caractère".**

**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 ça 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).**



## **type set**



**SET est fort semblable à ENUM. Une colonne SET est en effet une colonne qui permet de stocker une chaîne de caractères dont les valeurs possibles sont prédéfinies par l'utilisateur.**

**La différence avec ENUM, c'est qu'on peut stocker dans la colonne entre 0 et x valeur(s), x étant le nombre de valeurs autorisées.**

**Donc, si l'on définit une colonne de type SET de la manière suivante :**  
**espece SET('chat', 'chien', 'tortue')**

**On pourra stocker dans cette colonne :**

- " (chaîne vide) ;**
  - 'chat' ;**
  - 'chat,tortue' ;**
  - 'chat,chien,tortue' ;**
  - 'chien,tortue' ;**
- 
- 

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

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

1 categorie ENUM("Soupes", "Viandes", "Tarte", "Dessert")

2 categorie SET("Soupes", "Viandes", "Tarte", "Dessert")


Tout se passe plutôt bien tant que vos éléments appartiennent aux catégories que vous avez définies au départ. Et puis tout à coup, vous vous retrouvez avec un élément qui ne correspond à aucune de vos catégories, mais qui devrait plutôt se trouver dans la catégorie "Entrées". Avec SET ou ENUM, il vous faut modifier la colonne categorie pour ajouter "Entrées" aux valeurs possibles.

Or, une des règles de base à respecter lorsque l'on conçoit une base de données, est que la structure de la base (donc les tables, les colonnes) ne doit pas changer lorsque l'on ajoute des données. Par conséquent, tout ce qui est susceptible de changer doit être une donnée, et non faire partie de la structure de la base.



**Pour éviter ENUM – Vous pouvez faire de la colonne categorie une simple colonne VARCHAR(100). Le désavantage est que vous ne pouvez pas limiter les valeurs entrées dans cette colonne.**

**Vous pouvez aussi ajouter une table Categorie qui reprendra toutes les catégories possibles. Dans la table des éléments, il suffira alors de stocker une référence vers la catégorie de l'élément. Pour éviter SET La solution consiste en la création de deux tables : une table Categorie, qui reprend les catégories possibles, et une table qui lie les éléments aux catégories auxquels ils appartiennent.**







## Date et Heures

- **DATE** pour la date type **YYYY-MM-DD**.
- **TIME** pour le temps avec la forme **HH:MM:SS.ssssss**
- **DATETIME** pour avoir les deux **YYYY-MM-DD HH:MM:SS**.



## Valeur NULL

**Une colonne qui n'est pas renseignée, et donc vide, est dite contenir la valeur NULL. Cette valeur n'est pas zéro, c'est une absence de valeur.**

**Toute expression dont au moins un des termes a la valeur NULL donne comme résultat la valeur NULL.**

**Remarque importante : les valeurs NULL ne sont pas incluses dans les indexes (d'autant plus intéressant dans le cas des indexes uniques).**



## **SQL IS NULL / IS NOT NULL**

**Dans le langage SQL, l'opérateur IS permet de filtrer les résultats qui contiennent la valeur NULL. Cet opérateur est indispensable car la valeur NULL est une valeur inconnue et ne peut par conséquent pas être filtrée par les opérateurs de comparaison (cf. égal, inférieur, supérieur ou différent).**

**Syntaxe Pour filtrer les résultats où les champs d'une colonne sont à NULL il convient d'utiliser la syntaxe suivante:**

```
SELECT *  
FROM `table`  
WHERE nom_colonne IS NULL
```

# 4 - Les Fonctions / Expressions



**+, -, \*, /, %**

**% aussi dit modulo retourne le reste d'une division. Pratique pour tester pai/impaire ou pour connaitre la ligne dans un tableau paginé.**

**SELECT 8 % 3;**

+	-----	+
	<b>8 % 3</b>	
+	-----	+
	<b>2</b>	
+	-----	+



**<=>**

**NULL-safe égal. Cet opérateur effectue une comparaison d'égalité comme l'opérateur =, mais renvoie 1 plutôt que NULL si les deux opérandes sont NULL, et 0 plutôt que NULL si un opérande est NULL.**

**L'opérateur <=> est équivalent à l'opérateur SQL standard IS NOT DISTINCT FROM.**





## BETWEEN()

L'opérateur BETWEEN est utilisé dans une requête SQL pour sélectionner un intervalle de données dans une requête utilisant WHERE. L'intervalle peut être constitué de chaînes de caractères, de nombres ou de dates. L'exemple le plus concret consiste par exemple à récupérer uniquement les enregistrements entre 2 dates définies.

```
SELECT *  
FROM `user`  
WHERE created_at BETWEEN '2021-01-01' AND '2021-01-31';
```

-- equivalent

```
SELECT *  
FROM `user`  
WHERE ('2021-01-01' <= created_at AND created_at <= '2021-01-31')
```





# LIKE()

L'opérateur LIKE est utilisé dans la clause WHERE des requêtes SQL. Ce mot-clé permet d'effectuer une recherche sur un modèle particulier. Il est par exemple possible de rechercher les enregistrements dont la valeur d'une colonne commence par telle ou telle [...]

```
SELECT *  
FROM table  
WHERE colonne LIKE modele
```

- Si l'on souhaite obtenir uniquement les clients des villes qui commencent par un "N", il est possible d'utiliser la requête suivante:

```
SELECT *  
FROM client  
WHERE ville LIKE 'N%'
```





- **LIKE '%a'** : le caractère **"%"** est un caractère joker qui remplace tous les autres caractères. Ainsi, ce modèle permet de rechercher toutes les chaînes de caractère qui se terminent par un **"a"**.
- **LIKE 'a%'** : ce modèle permet de rechercher toutes les lignes de **"colonne"** qui commencent par un **"a"**.
- **LIKE '%a%'** : ce modèle est utilisé pour rechercher tous les enregistrements qui utilisent le caractère **"a"**.
- **LIKE 'pa%on'** : ce modèle permet de rechercher les chaînes qui commencent par **"pa"** et qui se terminent par **"on"**, comme **"pantalon"** ou **"pardon"**.
- **LIKE 'a\_c'** : le caractère **"\_"** (underscore) peut être remplacé par n'importe quel caractère, mais un et un seul caractère uniquement (alors que le symbole pourcentage **"%"** peut être remplacé par un nombre indéterminé de caractères. Ainsi, ce modèle permet de retourner les lignes **"aac"**, **"abc"** ou même **"azc"**.



IN

L' INopérateur vous permet de déterminer si une valeur correspond à une valeur dans une liste de valeurs. Voici la syntaxe de l' INopérateur :

**value IN (value1, value2, value3,...)**

L' INopérateur renvoie 1 (vrai) si valueest égal à n'importe quelle valeur de la liste ( value1, value2, value3,...). Sinon, il renvoie 0.

**SELECT 4 IN (1,2,3);**



### D'opérateurs logiques

- AND
- OR
- NOT

### De comparateurs de chaîne :

- IN
- BETWEEN
- LIKE

### D'opérateurs arithmétiques :

- +
- -
- \*
- /
- %
- &
- |
- ^
- ~

### Et de comparateurs arithmétiques :

- =
- <=
- >
- <
- <=
- >=
- <
- >
- <=

# 5 - La création



**Nous allons donc créer notre base de données, que nous appellerons elevage.  
droit dessus.**

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

**CREATE DATABASE nom\_base;**

**CREATE [OR REPLACE] {DATABASE | SCHEMA} [IF NOT EXISTS] db\_name  
[create\_specification] ...**

**create\_specification:**

**[DEFAULT] CHARACTER SET [=] charset\_name  
| [DEFAULT] COLLATE [=] collation\_name  
| COMMENT [=] 'comment'**





```
CREATE OR REPLACE DATABASE ma_premiere_db  
CHARACTER SET = 'utf8mb4'  
COLLATE = 'utf8mb4_general_ci';
```

```
-- équivalent à  
DROP DATABASE IF EXISTS ma_premiere_db;  
CREATE DATABASE ma_premiere_db;
```

## La commande CREATE TABLE

permet de créer une table en SQL. Un tableau est une entité qui est contenu dans une base de données pour stocker des données ordonnées dans des colonnes. La création d'une table sert à définir les colonnes et le type de données qui seront contenus dans chacun des colonne (entier, chaîne de caractères, date, valeur binaire ...).

La syntaxe générale pour créer une table est la suivante :

```
CREATE TABLE nom_de_la_table  
(  
    colonne1 type_donnees,  
    colonne2 type_donnees,  
    colonne3 type_donnees,  
    colonne4 type_donnees  
);
```



**Dans cette requête, 4 colonnes ont été définies. Le mot-clé “type\_donnees” sera à remplacer par un mot-clé pour définir le type de données (INT, DATE, TEXT ...). Pour chaque colonne, il est également possible de définir des options telles que (liste non-exhaustive):**

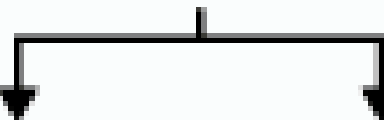
- **NOT NULL** : empêche d’enregistrer une valeur nulle pour une colonne.
  - **DEFAULT** : attribuer une valeur par défaut si aucune données n’est indiquée pour cette colonne lors de l’ajout d’une ligne dans la table.
  - **PRIMARY KEY** : indiquer si cette colonne est considérée comme clé primaire pour un index.
- 
- 



## Contraintes de clé primaire

Une table contient généralement une colonne ou une combinaison de colonnes dont les valeurs identifient de façon unique chaque ligne dans la table. Cette colonne (ou ces colonnes), appelée clé primaire (PK, Primary Key), assure l'intégrité de l'entité de la table. Les contraintes de clé primaire garantissent des données uniques, c'est pourquoi elles sont souvent définies pour une colonne d'identité.

Primary Key

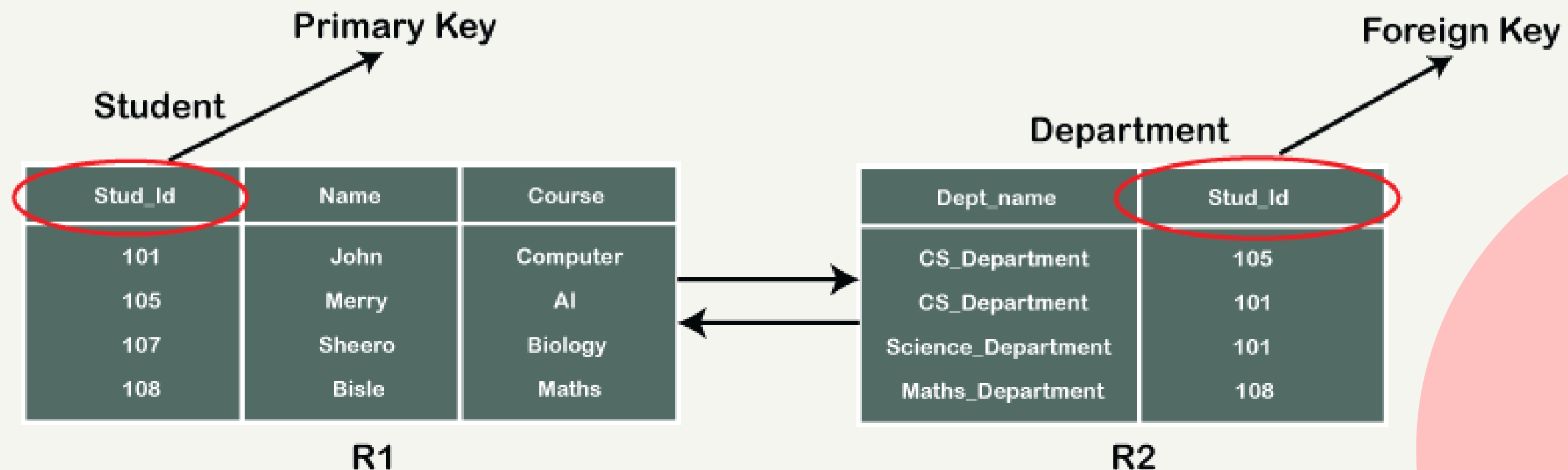


ProductID	VendorID	AverageLeadTime	StandardPrice	LastReceiptCost
1	1	17	47.8700	50.2635
2	104	19	39.9200	41.9160
7	4	17	54.3100	57.0255
609	7	17	25.7700	27.0585
609	100	19	28.1700	29.5785

ProductVendor table

## Foreign Key

On appelle « clé étrangère » une colonne ou une combinaison de colonnes utilisée pour établir et conserver une liaison entre les données de deux tables pour contrôler les données qui peuvent être stockées dans la table de clés étrangères. Dans une référence de clé étrangère, la création d'une liaison entre deux tables s'effectue lors du référencement de la ou des colonnes contenant les valeurs de clé primaire d'une table dans la ou les colonnes de l'autre table. Cette colonne devient alors une clé étrangère dans la seconde table.





## SQL INSERT INTO

**L'insertion de données dans une table s'effectue à l'aide de la commande INSERT INTO.  
Cette commande permet au choix d'inclure une seule ligne à la base existante ou  
plusieurs lignes d'un coup.**



**Insérer une ligne en spécifiant toutes les colonnes**  
**La syntaxe pour remplir une ligne avec cette méthode est la suivante :**

**INSERT INTO table VALUES ('valeur 1', 'valeur 2', ...)**





**Insérer une ligne en spécifiant seulement les colonnes souhaitées**

**Cette deuxième solution est très similaire, excepté qu'il faut indiquer le nom des colonnes avant "VALUES". La syntaxe est la suivante :**

**INSERT INTO table (nom\_colonne\_1, nom\_colonne\_2, ...  
VALUES ('valeur 1', 'valeur 2', ...)**

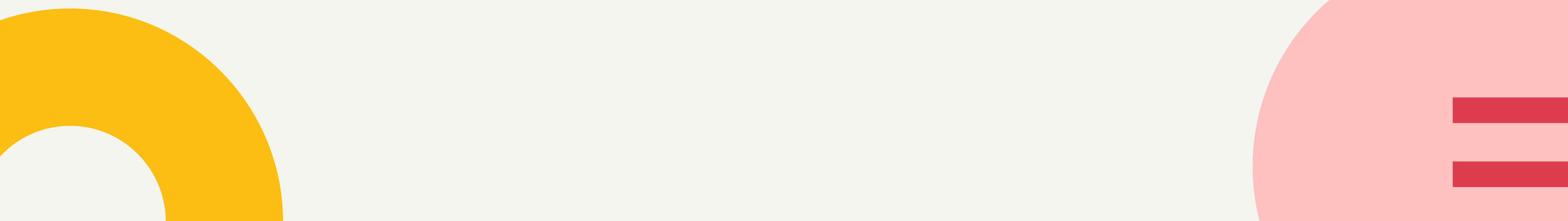




## Insertion de plusieurs lignes à la fois

**Il est possible d'ajouter plusieurs lignes à un tableau avec une seule requête. Pour ce faire, il convient d'utiliser la syntaxe suivante :**

```
INSERT INTO client (prenom, nom, ville, age)  
VALUES  
('Rébecca', 'Armand', 'Saint-Didier-des-Bois', 24),  
('Aimée', 'Hebert', 'Marigny-le-Châtel', 36),  
('Marielle', 'Ribeiro', 'Maillères', 27),  
('Hilaire', 'Savary', 'Conie-Molitard', 58);
```





# **select**

**L'utilisation la plus courante de SQL consiste à lire des données issues de la base de données. Cela s'effectue grâce à la commande SELECT, qui retourne des enregistrements dans un tableau de résultat. Cette commande peut sélectionner une ou plusieurs colonnes d'une table.**

**SELECT nom\_du\_champ FROM nom\_du\_tableau**





**Il est possible de retourner automatiquement toutes les colonnes d'un tableau sans avoir à connaître le nom de toutes les colonnes. Au lieu de lister toutes les colonnes, il faut simplement utiliser le caractère "\*" (étoile). C'est un joker qui permet de sélectionner toutes les colonnes. Il s'utilise de la manière suivante:**

**SELECT \* FROM client**

**Cette requête SQL retourne exactement les mêmes colonnes qu'il y a dans la base de données**







# WHERE



La commande **WHERE** dans une requête SQL permet d'extraire les lignes d'une base de données qui respectent une condition. Cela permet d'obtenir uniquement les informations désirées.

## Syntaxe

La commande **WHERE** s'utilise en complément à une requête utilisant **SELECT**. La façon la plus simple de l'utiliser est la suivante:

**SELECT nom\_colonnes FROM nom\_table WHERE condition**

Pour obtenir seulement la liste des clients qui habitent à Paris, il faut effectuer la requête suivante:

**SELECT \* FROM client WHERE ville = 'paris'**

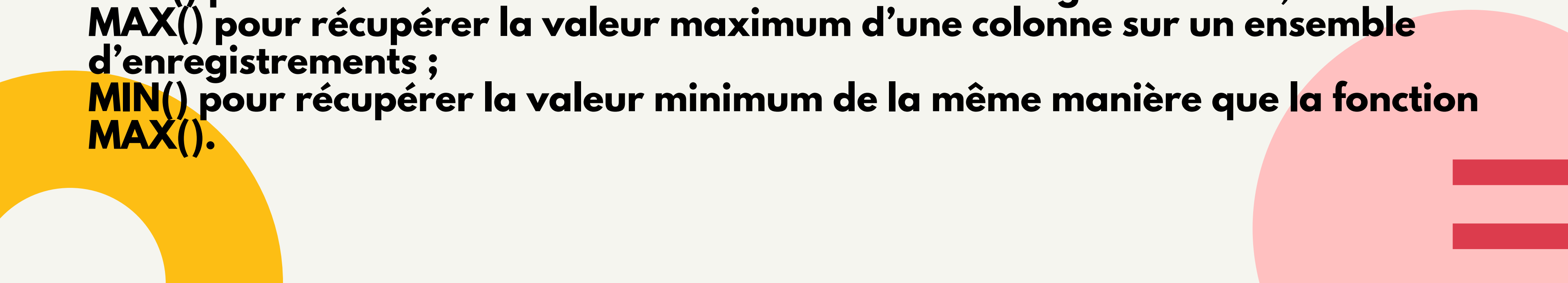


# 6 - Fonctions d'agrégation



**En SQL, les fonctions d'agrégation permettent de réaliser des opérations arithmétiques et statistiques au sein d'une requête. Les principales fonctions sont les suivantes :**

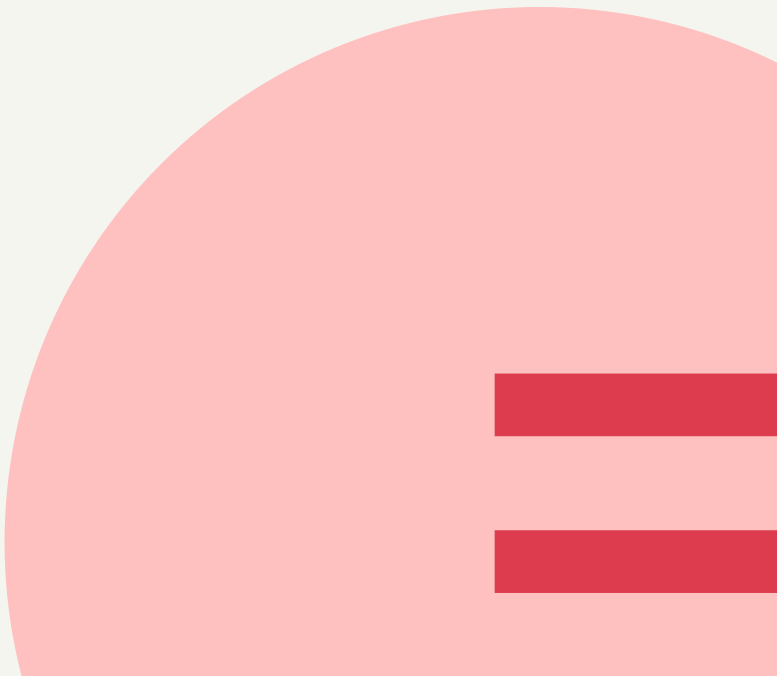

**COUNT()** pour compter le nombre d'enregistrements d'une table ou d'une colonne distincte ;  
**AVG()** pour calculer la moyenne sur un ensemble d'enregistrements ;  
**SUM()** pour calculer la somme sur un ensemble d'enregistrements ;  
**MAX()** pour récupérer la valeur maximum d'une colonne sur un ensemble d'enregistrements ;  
**MIN()** pour récupérer la valeur minimum de la même manière que la fonction **MAX()**.





## COUNT()

**En SQL, la fonction d'agrégation COUNT() permet de compter le nombre d'enregistrements dans une table, ce qui s'avère très pratique dans de nombreux cas. On peut notamment citer le nombre de commandes d'un site e-commerce, le nombre d'articles d'un blog, ou encore le nombre de produits d'un catalogue. Pour compter le nombre d'enregistrements sur une colonne en particulier, il suffit de reprendre la syntaxe précédente. Les enregistrements qui possèdent la valeur NULL ne seront pas comptabilisés**



- **CREATE TABLE student (name CHAR(10), test CHAR(10), score INT);**
- **INSERT INTO student VALUES**  
**('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),**  
**('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),**  
**('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),**  
**('Tatiana', 'SQL', 87), ('Tatiana', 'Tuning', 83);**
- **SELECT COUNT(\*) FROM student;**



Pour compter le nombre total de clients dans la table, il suffit d'utiliser le joker dans la fonction **COUNT()** sur toute la table. La syntaxe est donc la suivante : **COUNT(\*)**

- **SELECT COUNT(\*) FROM clients;**
- **COUNT(DISTINCT) example:**

**SELECT COUNT(DISTINCT (name)) FROM student;**





## **AVG()**

**En SQL, la fonction d'agrégation AVG() permet de calculer une valeur moyenne sur un ensemble d'enregistrements de type numérique et qui ne possèdent pas la valeur NULL. Les cas d'utilisation ne manquent pas. En utilisant la table clients décrite précédemment, on peut envisager de calculer le nombre moyen de commandes par clients, si celui-ci a commandé au moins une fois.**


**SELECT AVG(commandes) FROM clients;**





**La requête sélectionne uniquement les 4 lignes dont la valeur de la colonne commandes n'est pas NULL. Au final, obtient le calcul suivant :  $(3 + 1 + 2 + 2) / 4 = 2$ .**

**Dans la table clients, si le client Paul Bismuth (#4) n'avait pas la valeur NULL pour sa colonne commandes, mais 0, alors le calcul aurait été le suivant :  $(3 + 1 + 2 + 0 + 2) / 5 = 1.6000$**



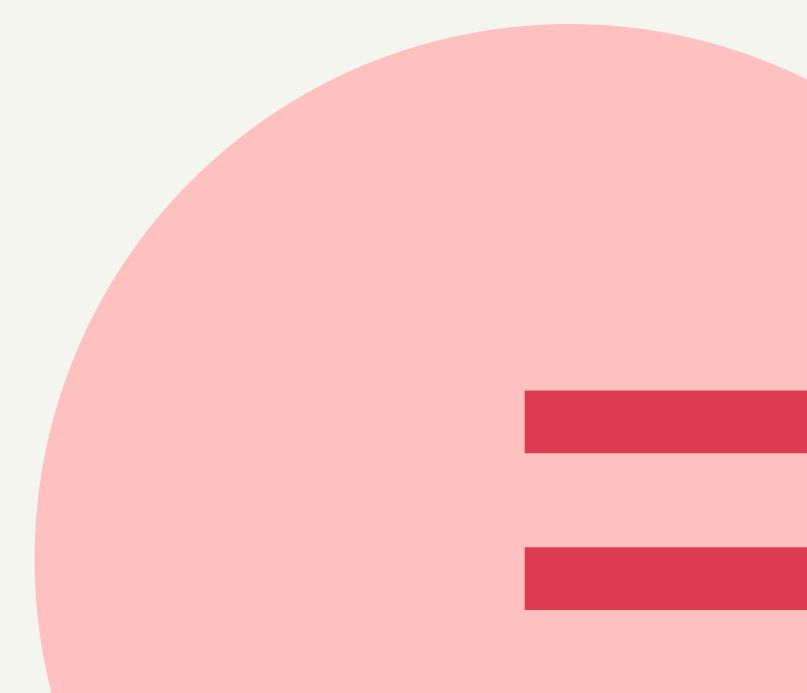





**SUM()**

**En SQL, la fonction d'agrégation SUM() permet de calculer la somme sur un ensemble d'enregistrements de type numérique et qui ne possèdent pas la valeur NULL. En utilisant la table clients décrite précédemment, on peut envisager compter le nombre de commandes dans la table clients.**

**SELECT SUM(commandes) AS somme FROM clients;**



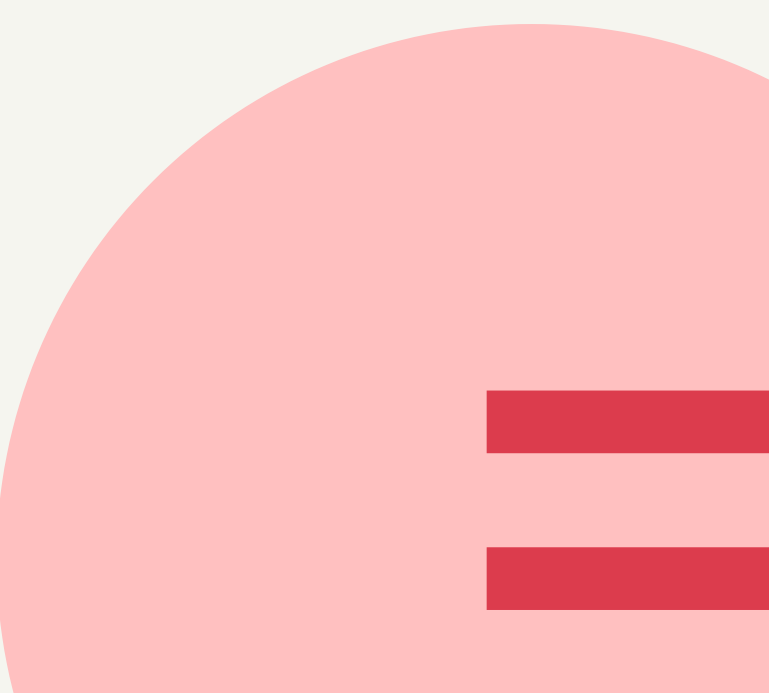



## **MAX() ET MIN()**

**En SQL, les fonctions d'agrégation MAX() et MIN() permettent de sélectionner la valeur respectivement maximale et minimale dans un ensemble d'enregistrements. Les valeurs NULL sont écartées. En utilisant la table clients décrite précédemment, on peut envisager rechercher le nombre de commandes maximal et minimal pour un client.**

**SELECT MAX(commandes) as max\_cmd FROM clients;**

**SELECT MIN(commandes) as min\_cmd FROM clients;**





## **GROUP BY**

**La commande GROUP BY est utilisée en SQL pour grouper plusieurs résultats et utiliser une fonction de totaux sur un groupe de résultat.**

**Sur une table qui contient toutes les ventes d'un magasin, il est par exemple possible de liste regrouper les ventes par clients identiques et d'obtenir le coût total des achats pour chaque client.**



De façon générale, la commande **GROUP BY** s'utilise de la façon suivante

```
SELECT colonne1, fonction(colonne2)  
FROM table  
GROUP BY colonne1
```

Pour obtenir le coût total de chaque client en regroupant les commandes des mêmes clients, il faut utiliser la requête suivante :

```
SELECT client, SUM(tarif)  
FROM achat  
GROUP BY client
```

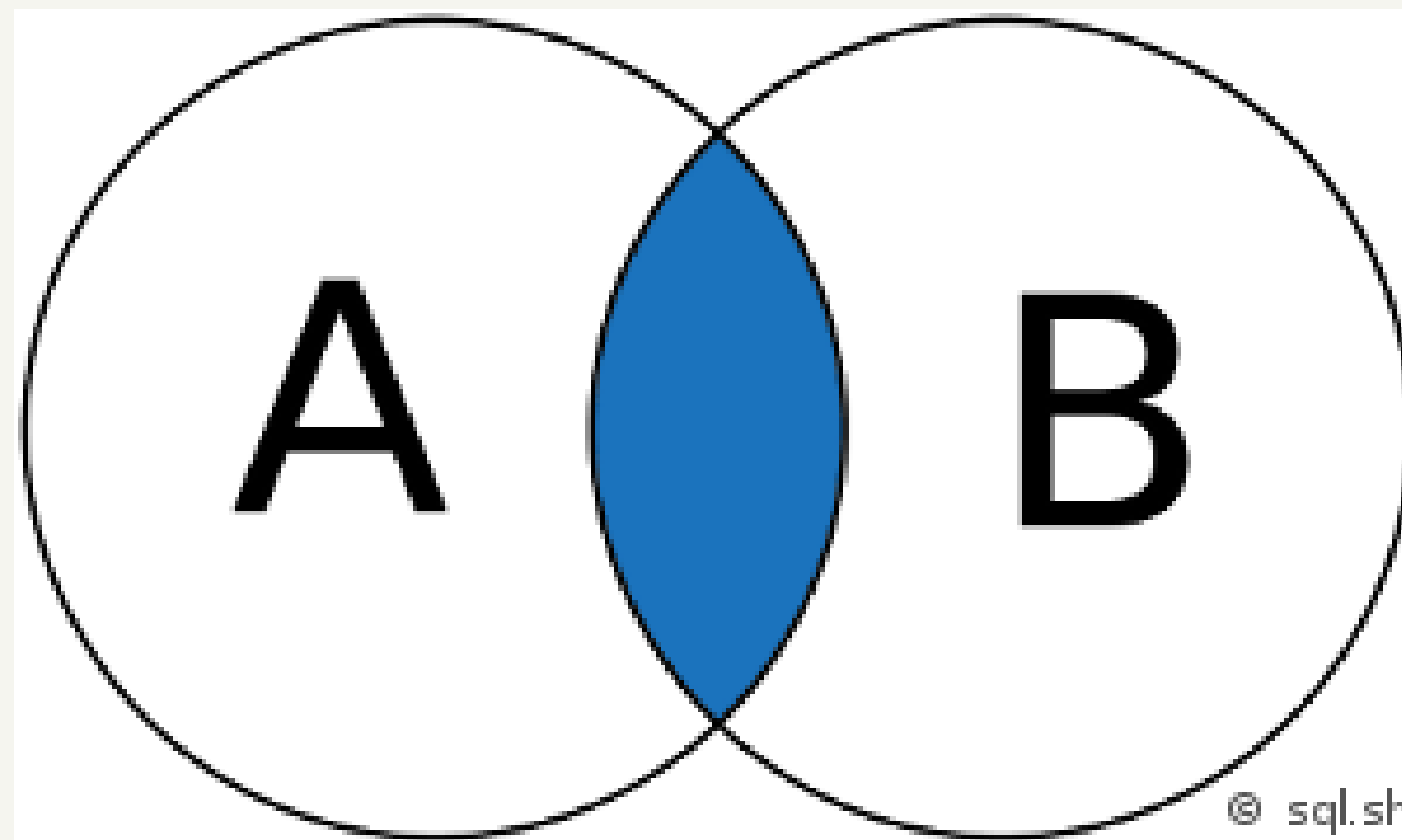
id	client	tarif	date
1	Pierre	102	2012-10-23
2	Simon	47	2012-10-27
3	Marie	18	2012-11-05
4	Marie	20	2012-11-14
5	Pierre	160	2012-12-03



client	SUM(tarif)
Pierre	262
Simon	47
Marie	38

# 6 - Les jointures

**Une jointure permet de récupérer des informations provenant de plusieurs tables. Ces tables doivent être liées par au moins une colonne (un nom, un ID, un numéro de téléphone, ce que vous voulez, mais ces deux tables doivent avoir une colonne en commun). C'est ce qu'on appelle un critère de jointure.**





**Il existe différents types de jointures en SQL mais tous ne sont pas supportés par le MySQL. Nous allons donc simplement nous concentrer sur les types de jointures suivants dans ce cours :**

- **L'INNER JOIN ;**
  - **Le LEFT (OUTER) JOIN ;**
  - **Le RIGHT (OUTER) JOIN ;**
  - **Le CROSS JOIN ;**
  - **Le SELF JOIN ;**
- 
- 



**INNER JOIN, aussi appelée EQUIJOIN, est un type de jointures très communes pour lier plusieurs tables entre-elles. Cette commande retourne les enregistrements lorsqu'il y a au moins une ligne dans chaque colonne qui correspond à la condition.**







## **Jointures internes (INNER JOIN)**

```
SELECT *  
FROM A  
  INNER JOIN B ON A.key = B.key
```

**Notons que c'est strictement équivalent à :**

```
SELECT *  
FROM A, B  
WHERE A.key = B.key
```





**la commande LEFT JOIN (aussi appelée LEFT OUTER JOIN) est un type de jointure entre 2 tables. Cela permet de lister tous les résultats de la table de gauche (left = gauche) même s'il n'y a pas de correspondance dans la deuxième tables.**




**SELECT \***  
**FROM table1**  
**LEFT OUTER JOIN table2 ON table1.id = table2.fk\_id**







**la commande RIGHT JOIN (ou RIGHT OUTER JOIN) est un type de jointure entre 2 tables qui permet de retourner tous les enregistrements de la table de droite (right = droite) même s'il n'y a pas de correspondance avec la table de gauche. S'il y a un enregistrement de la table de droite qui ne trouve pas de correspondance dans la table de gauche, alors les colonnes de la table de gauche auront NULL pour valeur.**







```
SELECT *  
FROM table1  
LEFT OUTER JOIN table2 ON table1.id = table2.fk_id
```





**la commande CROSS JOIN est un type de jointure sur 2 tables SQL qui permet de retourner le produit cartésien. Autrement dit, cela permet de retourner chaque ligne d'une table avec chaque ligne d'une autre table. Ainsi effectuer le produit cartésien d'une table A qui contient 30 résultats avec une table B de 40 résultats va produire 1200 résultats ( $30 \times 40 = 1200$ ).**





```
SELECT *  
FROM table1  
CROSS JOIN table2
```

**Méthode alternative pour retourner les mêmes résultats :**

```
SELECT *  
FROM table1, table2
```

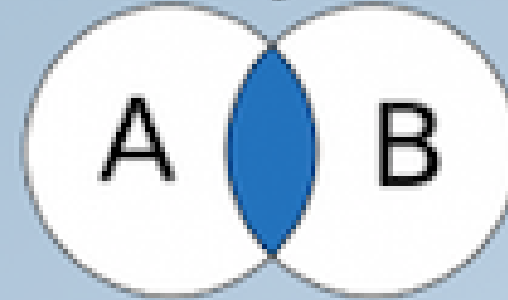




**un SELF JOIN correspond à une jointure d'une table avec elle-même.  
Ce type de requête n'est pas si commun mais très pratique dans le  
cas où une table lie des informations avec des enregistrements de la  
même table.**

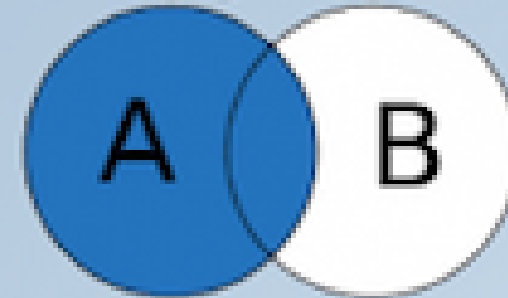


### INNER JOIN



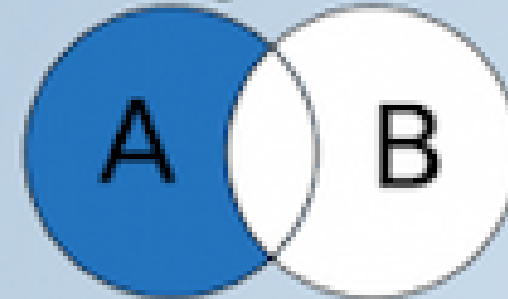
```
SELECT *  
FROM A  
INNER JOIN B ON A.key = B.key
```

### LEFT JOIN



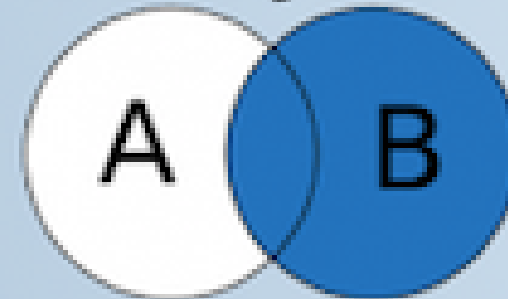
```
SELECT *  
FROM A  
LEFT JOIN B ON A.key = B.key
```

### LEFT JOIN (sans l'intersection de B)



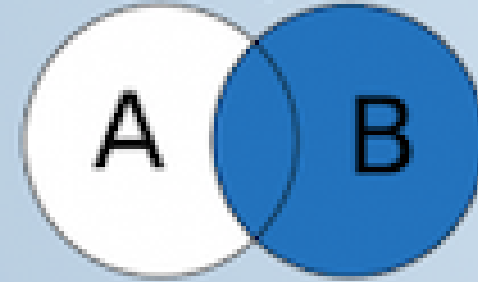
```
SELECT *  
FROM A  
LEFT JOIN B ON A.key = B.key  
WHERE B.key IS NULL
```

### RIGHT JOIN



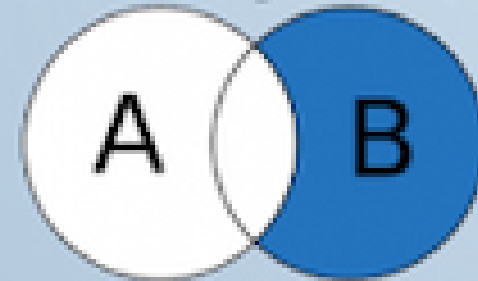
```
SELECT *  
FROM A  
RIGHT JOIN B ON A.key = B.key
```

### RIGHT JOIN



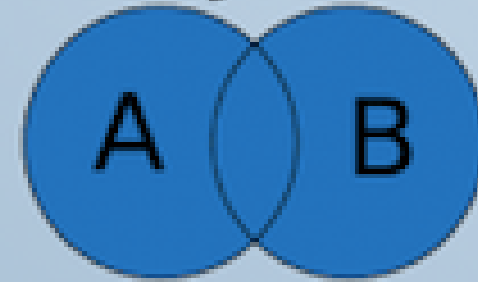
```
SELECT *  
FROM A  
RIGHT JOIN B ON A.key = B.key
```

### RIGHT JOIN (sans l'intersection de A)



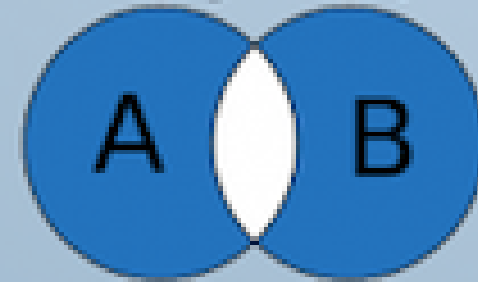
```
SELECT *  
FROM A  
RIGHT JOIN B ON A.key = B.key  
WHERE B.key IS NULL
```

### FULL JOIN



```
SELECT *  
FROM A  
FULL JOIN B ON A.key = B.key
```

### FULL JOIN (sans intersection)



```
SELECT *  
FROM A  
FULL JOIN B ON A.key = B.key  
WHERE A.key IS NULL  
OR B.key IS NULL
```

# 7 - Les manipulations



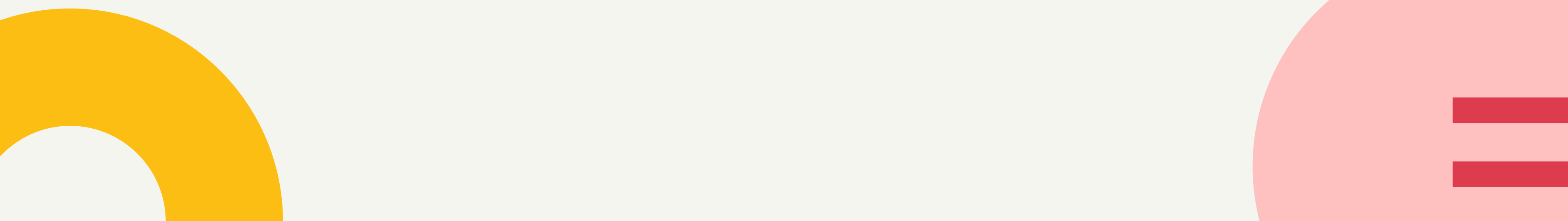
# UPDATE

La commande **UPDATE** permet d'effectuer des modifications sur des lignes existantes. Très souvent cette commande est utilisée avec **WHERE** pour spécifier sur quelles lignes doivent porter la ou les modifications.

## Syntaxe

La syntaxe basique d'une requête utilisant **UPDATE** est la suivante :

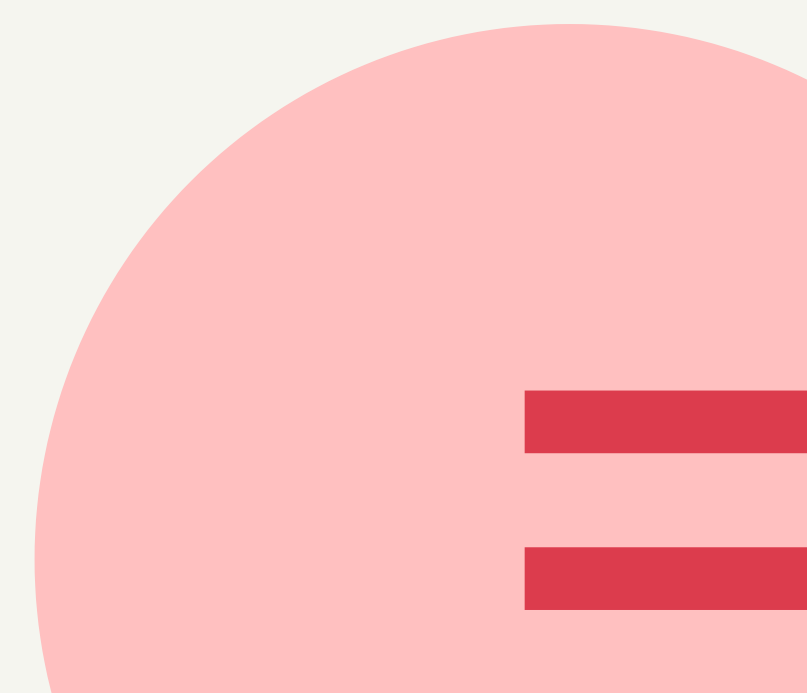

```
UPDATE table  
SET nom_colonne_1 = 'nouvelle valeur'  
WHERE condition
```





**pour spécifier en une seule fois plusieurs modification, il faut séparer les attributions de valeur par des virgules. Ainsi la syntaxe deviendrait la suivante :**

**UPDATE table**  
**SET colonne\_1 = 'valeur 1', colonne\_2 = 'valeur 2', colonne\_3 = 'valeur 3'**  
**WHERE condition**





# DELETE

**La commande DELETE en SQL permet de supprimer des lignes dans une table. En utilisant cette commande associé à WHERE il est possible de sélectionner les lignes concernées qui seront supprimées.**





**La syntaxe pour supprimer des lignes est la suivante :**

**DELETE FROM `table`  
WHERE condition**

**Attention : s'il n'y a pas de condition WHERE alors toutes les lignes seront supprimées et la table sera alors vide.**



# ALTER TABLE

**La commande ALTER TABLE en SQL permet de modifier une table existante. Idéal pour ajouter une colonne, supprimer une colonne ou modifier une colonne existante, par exemple pour changer le type.**

**Syntaxe de base :**

**D'une manière générale, la commande s'utilise de la manière suivante:**

**ALTER TABLE nom\_table  
instruction**

**Le mot-clé "instruction" ici sert à désigner une commande supplémentaire, qui sera détaillée ci-dessous selon l'action que l'ont souhaite effectuer : ajouter, supprimer ou modifier une colonne.**





# Ajouter une colonne

## Syntaxe

L'ajout d'une colonne dans une table est relativement simple et peut s'effectuer à l'aide d'une requête ressemblant à ceci:

```
ALTER TABLE nom_table  
ADD nom_colonne type_donnees
```





## Supprimer une colonne

**Une syntaxe permet également de supprimer une colonne pour une table. Il y a 2 manières totalement équivalente pour supprimer une colonne:**

```
ALTER TABLE nom_table  
DROP nom_colonne
```





## **Modifier une colonne**

**Pour modifier une colonne, comme par exemple changer le type d'une colonne, il y a différentes syntaxes selon le SGBD.**

**MySQL**

**ALTER TABLE nom\_table**

**MODIFY nom\_colonne type\_donnees**





## renommer colonne

**Pour renommer une colonne, il convient d'indiquer l'ancien nom de la colonne et le nouveau nom de celle-ci.**

**Pour MySQL, il faut également indiquer le type de la colonne.**

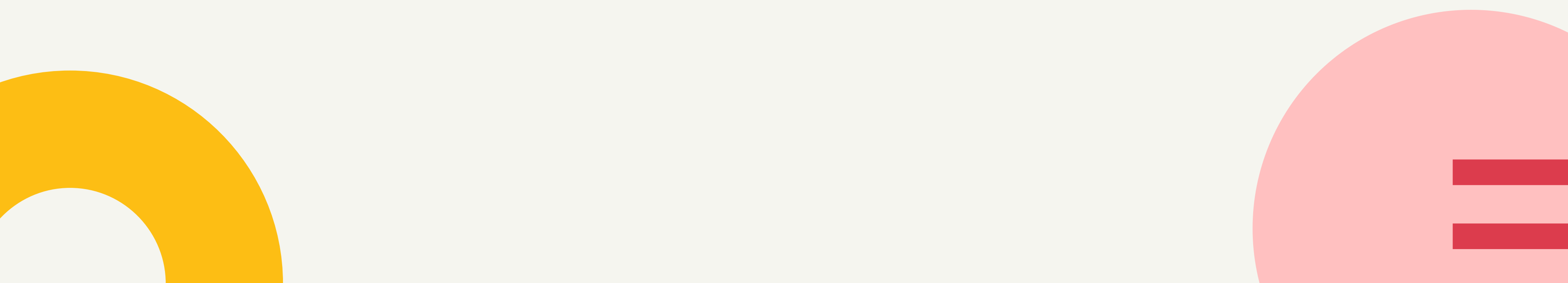
**ALTER TABLE nom\_table**  
**CHANGE** colonne\_ancien\_nom colonne\_nouveau\_nom type\_donnees





## SQL CASE

**Dans le langage SQL, la commande CASE ... WHEN ... permet d'utiliser des conditions de type "si / sinon" (cf. if / else) similaire à un langage de programmation pour retourner un résultat disponible entre plusieurs possibilités. Le CASE peut être utilisé dans n'importe quelle instruction ou clause, telle que SELECT, UPDATE, DELETE, WHERE, ORDER BY ou HAVING.**





## Syntaxe

**CASE α**

**WHEN 1 THEN 'un'**

**WHEN 2 THEN 'deux'**

**WHEN 3 THEN 'trois'**

**ELSE 'autre'**

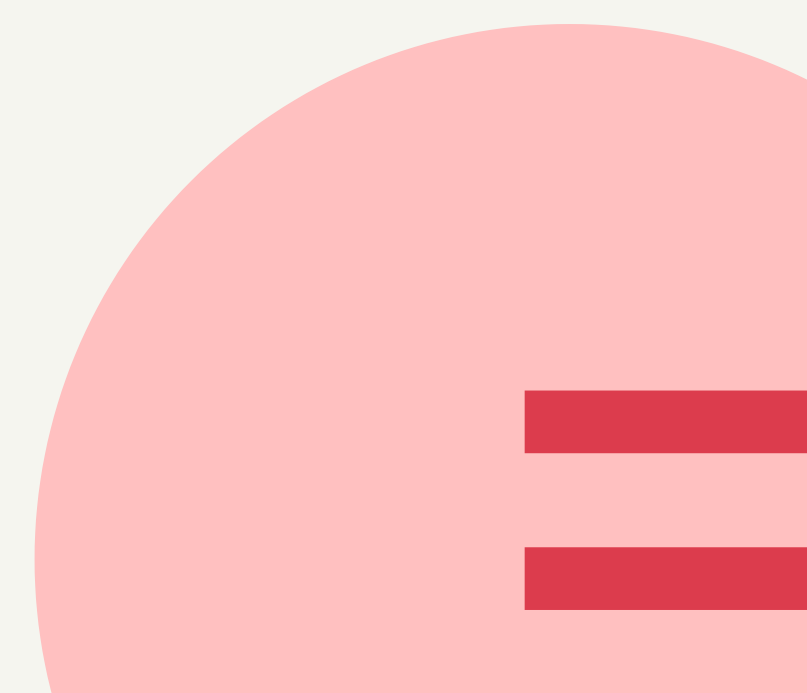

**END**



## Exemple :

**Il est possible d'effectuer une requête qui va afficher un message personnalisé en fonction de la valeur de la marge. Le message sera différent selon que la marge soit égale à 1, supérieur à 1 ou inférieure à 1. La requête peut se présenter de la façon suivante:**

```
SELECT id, nom, marge_pourcentage,  
CASE  
  WHEN marge_pourcentage=1 THEN 'Prix ordinaire'  
  WHEN marge_pourcentage>1 THEN 'Prix supérieur à la normale'  
  ELSE 'Prix inférieur à la normale'  
END  
FROM `achat`
```

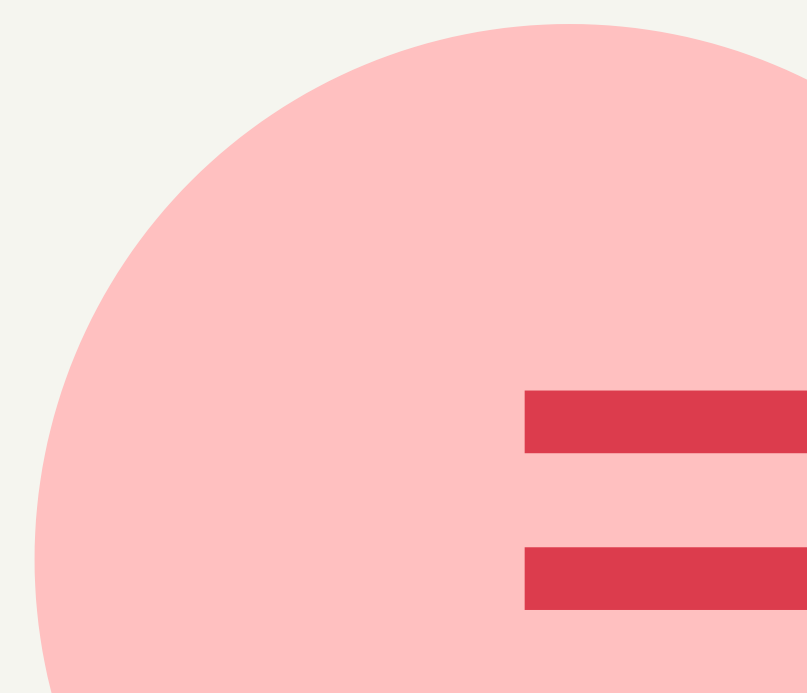





## Utilisation avec update



```
UPDATE `achat`  
SET `quantite` = (  
  CASE  
    WHEN `surcharge` < 1 THEN `quantite` + 1  
    WHEN `surcharge` > 1 THEN `quantite` - 1  
    ELSE quantite  
  END  
)
```







## **ORDER BY**

**La commande ORDER BY permet de trier les lignes dans un résultat d'une requête SQL. Il est possible de trier les données sur une ou plusieurs colonnes, par ordre ascendant ou descendant.**

### **Syntaxe**

**Une requête où l'on souhaite filtrer l'ordre des résultats utilise la commande ORDER BY de la sorte :**

```
SELECT colonne1, colonne2  
FROM table  
ORDER BY colonne1
```





**SELECT colonne1, colonne2, colonne3**

**FROM table**

**ORDER BY colonne1 DESC, colonne2 ASC**

**il n'est pas obligé d'utiliser le suffixe "ASC" sachant que les résultats sont toujours classés par ordre ascendant par défaut. Toutefois, c'est plus pratique pour mieux s'y retrouver, surtout si on a oublié l'ordre par défaut.**





**Pour récupérer par exemple la liste des utilisateurs par ordre alphabétique du nom de famille, il est possible d'utiliser la requête suivante :**

```
SELECT *  
FROM utilisateur  
ORDER BY nom
```



## LIMIT

**La clause LIMIT est à utiliser dans une requête SQL pour spécifier le nombre maximum de résultats que l'ont souhaite obtenir. Cette clause est souvent associé à un OFFSET, c'est-à-dire effectuer un décalage sur le jeu de résultat.**





## Limit et Offset avec *MySQL*

La syntaxe avec *MySQL* est légèrement différente :

```
SELECT *  
FROM table  
LIMIT 5, 10;
```

Cette requête retourne les enregistrements 6 à 15 d'une table. Le premier nombre est l'OFFSET tandis que le suivant est la limite.

