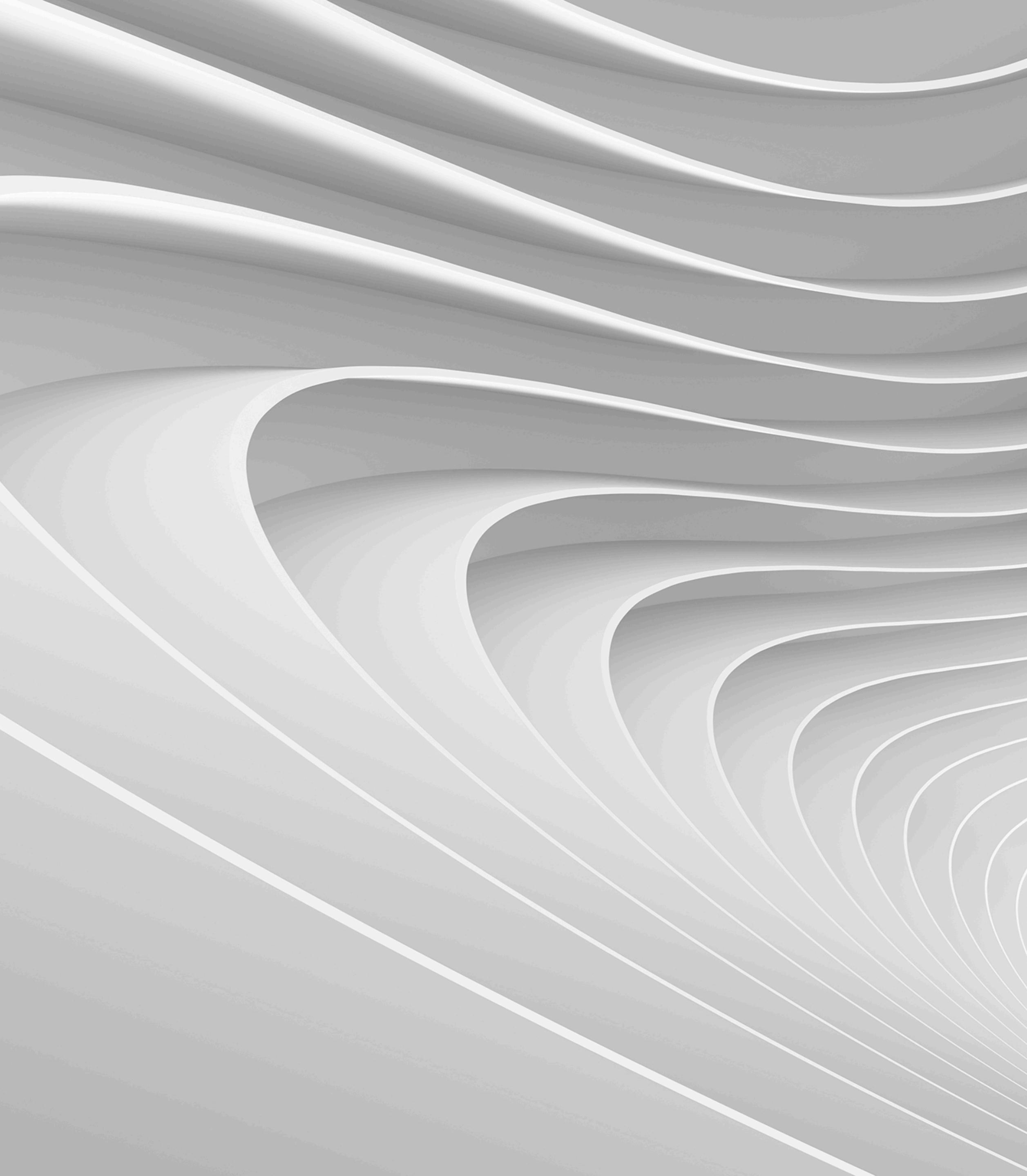


# SQL

## Structured Query Language

Hedi RIVAS

# Introduction



# Base de données

Une base de données est une collection organisée de données structurées, stockées de manière cohérente électroniquement.

Elle est conçue pour permettre la gestion, la manipulation et la récupération efficace des informations qu'elle contient.

# Base de données

## Les tables

Une base de données est généralement composée de tables, qui sont des structures de données rectangulaires organisées en colonnes (champs) et en lignes (enregistrements). Les tables sont reliées entre elles par des relations, ce qui permet d'établir des liens logiques entre les données.

# Base de données

## Les tables

Une base de données contient une ou plusieurs tables, dont les noms doivent être uniques au sein de la base de données. Une table contient des colonnes. Les colonnes contiennent les données.

# SGBDR

SGBDR : Système de Gestion de Base de Données Relationnelle

Fait référence à un logiciel utilisé pour stocker, gérer et interagir avec des données organisées selon le modèle de base de données relationnelle.

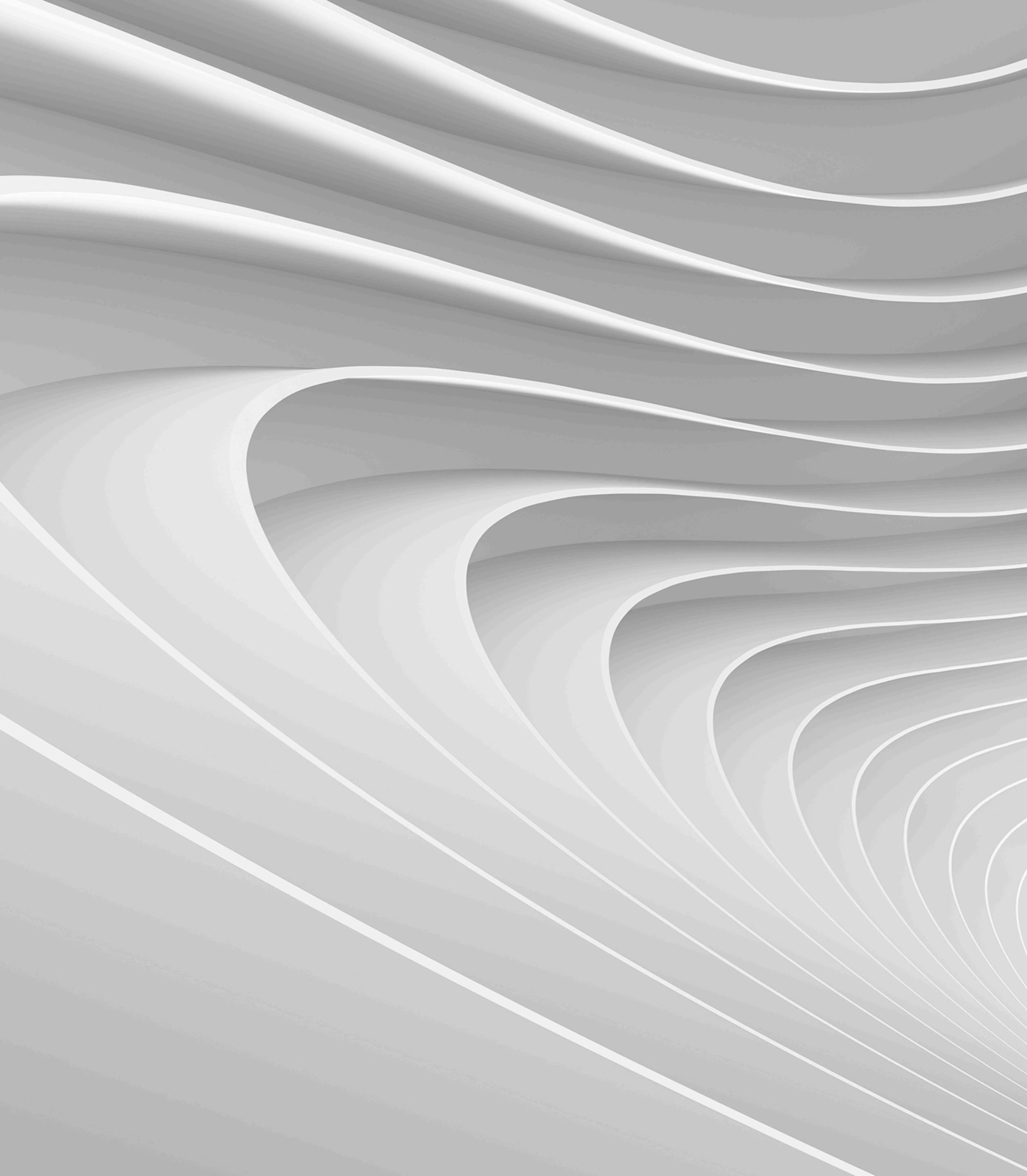
# MariaDB / MySQL

## Langage SQL

MySQL utilise le langage SQL (Structured Query Language) pour interagir avec les bases de données.

SQL est un langage standardisé largement utilisé pour la manipulation des données, permettant aux développeurs d'exécuter des requêtes, de créer, de modifier et de supprimer des données dans les tables.

ACID



# Principes ACID

Les principes ACID sont un ensemble de propriétés qui garantissent que les transactions de bases de données sont traitées de manière fiable. ACID est un acronyme pour Atomicity, Consistency, Isolation, et Durability.

Ces propriétés assurent que les bases de données restent précises et cohérentes même en cas de défaillances matérielles, logicielles ou humaines.

# Principes ACID

## Atomicité (Atomicity)

L'atomicité garantit que chaque transaction est traitée comme une unité indivisible.

Cela signifie que toutes les opérations dans une transaction sont exécutées avec succès ou aucune ne l'est. Si une partie de la transaction échoue, la transaction entière échoue, et la base de données est laissée inchangée.

# Principes ACID

## Cohérence (Consistency)

La cohérence garantit que la base de données passe d'un état valide à un autre état valide après une transaction.

Toutes les règles définies (comme les contraintes, les déclencheurs, etc.) sont respectées avant et après la transaction.

# Principes ACID

## Isolation

L'isolation garantit que les transactions concurrentes se déroulent de manière isolée et que les opérations d'une transaction ne sont pas visibles aux autres transactions jusqu'à ce qu'elles soient terminées.

Cela prévient les problèmes comme les lectures sales, les lectures non répétables et les lectures fantômes.

# Principes ACID

## Durabilité (Durability)

La durabilité garantit que les résultats d'une transaction sont permanents et persisteront même en cas de panne système.

Une fois qu'une transaction est validée (committed), ses modifications ne seront pas perdues.

# Principes ACID

## Conclusion

Les principes ACID sont cruciaux pour assurer l'intégrité, la fiabilité et la cohérence des transactions dans une base de données. Ils forment la base sur laquelle repose la gestion des transactions dans les systèmes de gestion de bases de données relationnelles (SGBDR), garantissant que les opérations sont exécutées de manière sécurisée et correcte, même en cas de défaillances.

# Pratique



# SQL

## Créer sa première base de données

Une fois connecté à MySQL, vous pouvez créer une nouvelle base de données en utilisant la commande **CREATE DATABASE**

Par exemple, pour créer une base de données appelée « my\_database », exéutez la commande suivante :

```
CREATE DATABASE my_database;
```

# SQL

Créer sa première base de données

Vous pouvez vérifier si la base de données a été créée avec succès en utilisant la commande :

```
SHOW DATABASES;
```

Pour commencer à travailler avec la base de données nouvellement créée :

```
USE mydatabase;
```

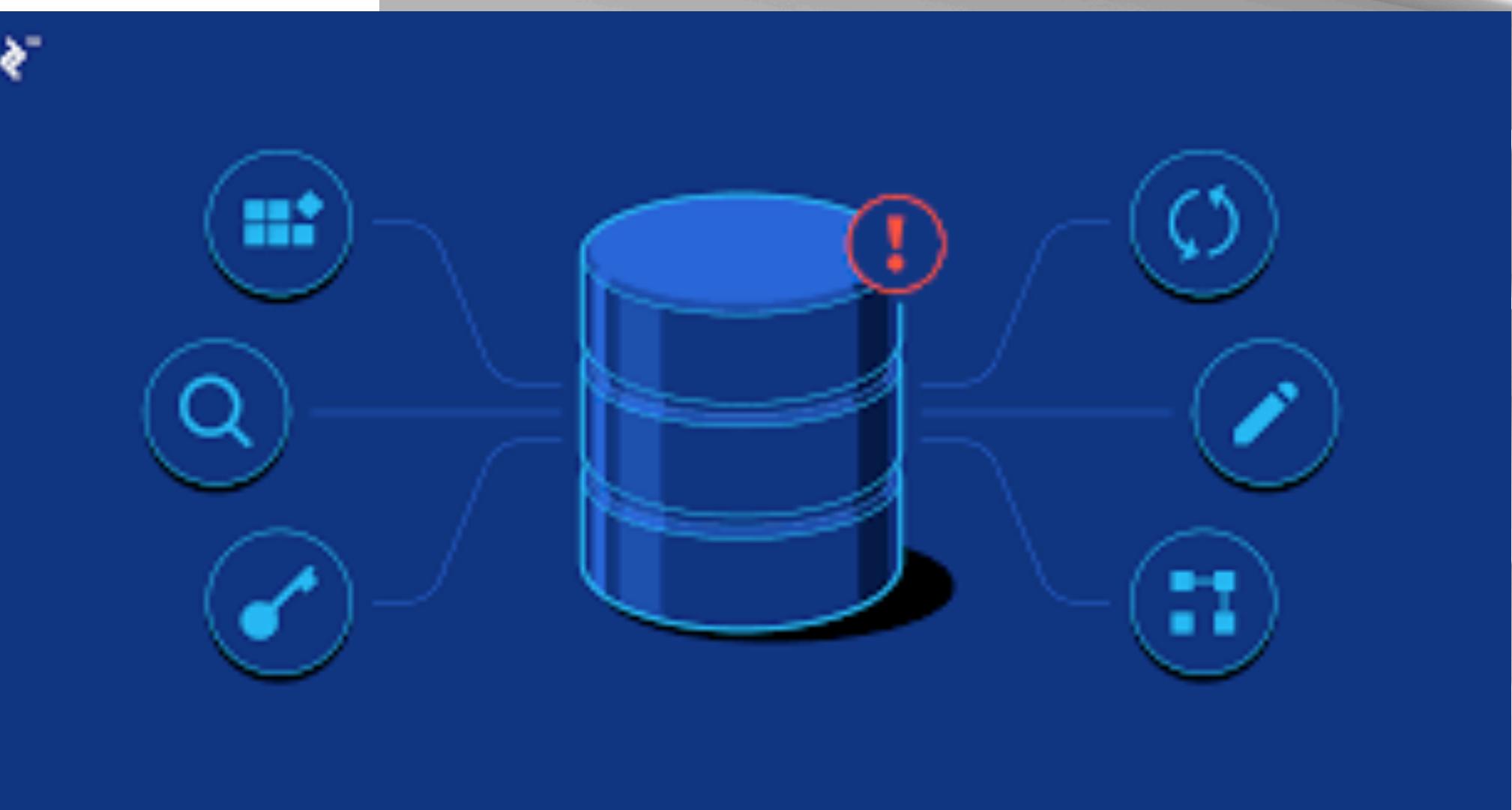
# SQL

## Créer sa première table

Une fois connecté à une base de donnée vous pouvez créer votre première table :

```
CREATE TABLE student (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nom VARCHAR(50),
    prénom VARCHAR(50),
    date_naissance DATE,
    adresse VARCHAR(100),
    email VARCHAR(100)
);
```

# Type de données



# SQL

## Types de données numériques

- INT : Entier signé.
- FLOAT : Nombre à virgule flottante simple précision.
- DOUBLE : Nombre à virgule flottante double précision.
- DECIMAL : Nombre décimal à précision fixe.

# SQL

## Types de données texte

- CHAR : Chaîne de caractère de longueur fixe.
- VARCHAR : Chaîne de caractère de longueur fixe.
- TEXT : Texte de longueur variable.

# SQL

## Types de données de date et d'heure

- DATE : Date au format « AAAA-MM-JJ ».
- TIME : Heure au format « HH:MM:SS ».
- DATETIME : Combinaison de date et d'heure au format « AAAA-MM-JJ HH:MM:SS ».
- TIMESTAMP : Horodatage au format « AAAA-MM-JJ HH-MM-SS »

# SQL

## Types de données booléennes

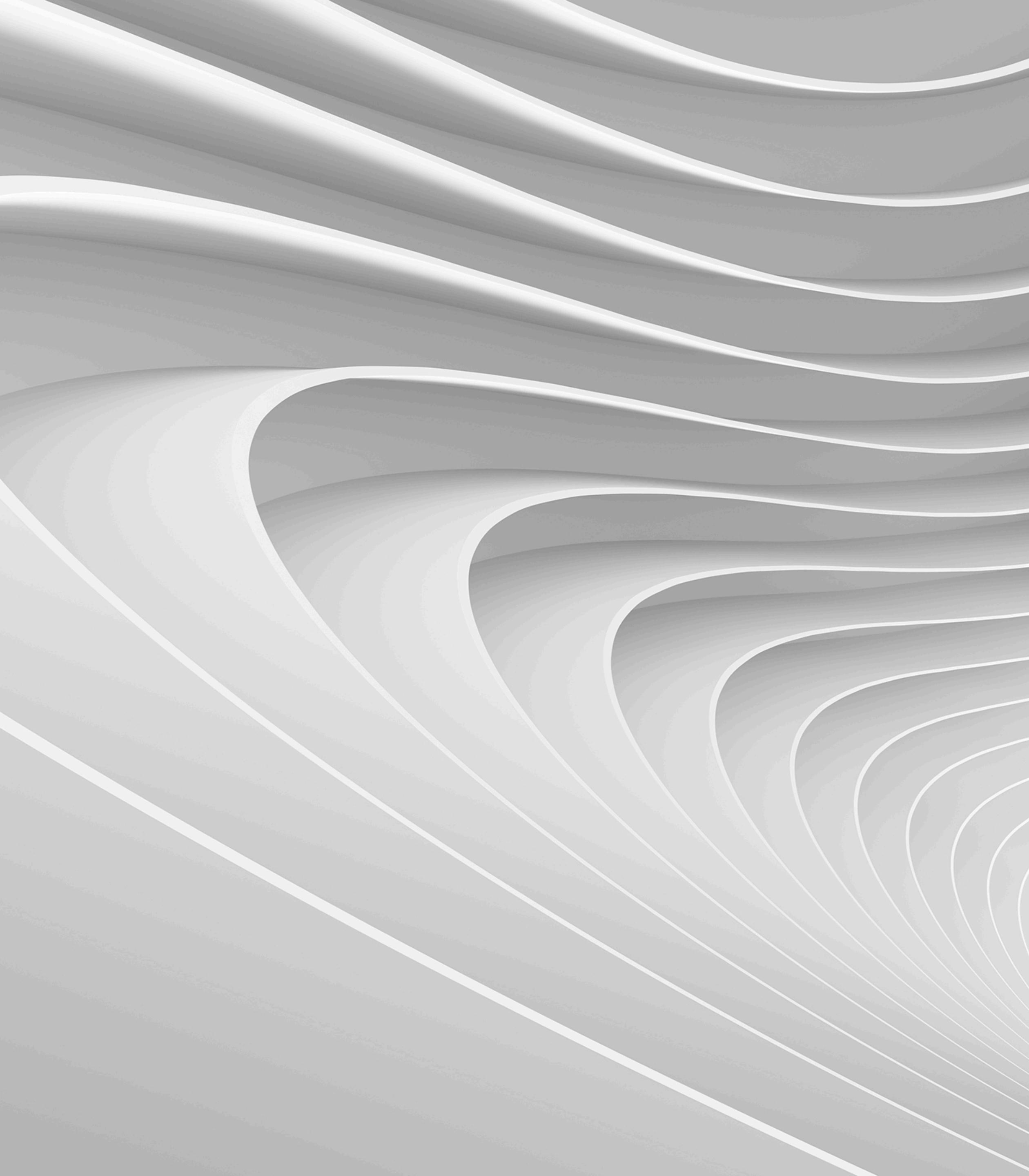
- DATE : Valeur booléenne (TRUE/FALSE) ou (0/1).

# SQL

## Types de données de clés

- PRIMARY KEY : Clé primaire, utilisée pour identifier de manière unique une ligne dans une table.
- FOREIGN KEY : Clé étrangère utilisée pour établir des relations entre les tables.

# Constraint



# SQL

## Les contraintes

En SQL, les contraintes sont utilisées pour appliquer les règles et des conditions aux données stockées dans les tables.



# SQL

## Les contraintes (clé primaire)

La contrainte **PRIMARY KEY** est utilisée pour identifier de manière unique chaque enregistrement dans une table.

Elle garantit que la valeur d'une colonne (ou d'un groupe de colonnes) est unique et non nulle.  
Une table ne peut avoir qu'une seule clé primaire.

# SQL

## Les contraintes (clé étrangère)

La contrainte **FOREIGN KEY** est utilisée établir une relation entre deux tables.

Elle garantit que les valeurs d'une colonne (ou d'un groupe de colonnes) d'une table correspondent aux valeurs d'une colonne de référence dans une autre table.

# SQL

## Les contraintes (unique)

La contrainte **UNIQUE** garantit que les valeurs d'une colonne (ou d'un groupe de colonnes) dans une table sont uniques, à l'exception de la valeur NULL.

Cela signifie qu'aucune autre ligne de la table ne peut avoir la même valeur pour la colonne spécifiée(s).

# SQL

## Les contraintes (not null)

La contrainte **NOT NULL** garantit qu'une colonne ne peut pas contenir de valeurs nulles.

Cela signifie que chaque enregistrement doit avoir une valeur non nulle pour la colonne spécifiée.

# SQL

## Les contraintes (default)

La contrainte **DEFAULT** est utilisée pour attribuer une valeur par défaut à une colonne si aucune valeur n'est spécifiée lors de l'insertion d'un nouvel enregistrement.

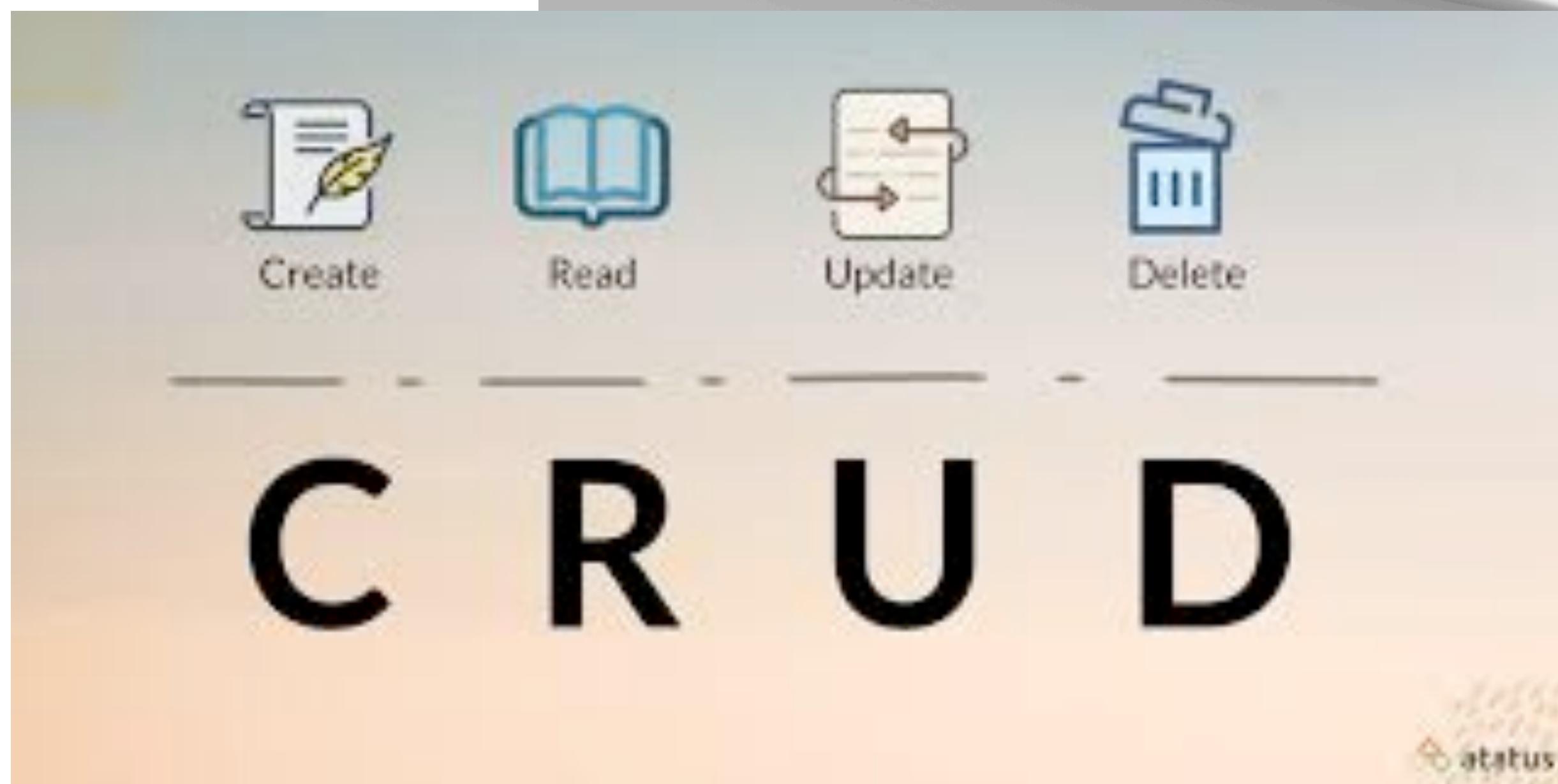
# SQL

## Les contraintes

Lors de la création d'une table, vous pouvez spécifier ces contraintes en utilisant la syntaxe appropriée. Par exemple, pour les contraintes PRIMARY KEY, FOREIGN KEY, NOT NULL et UNIQUE :

```
CREATE TABLE nom_de_la_table (
    id INT PRIMARY KEY,
    email VARCHAR(50) UNIQUE NOT NULL,
    colonne_ref INT,
    FOREIGN KEY (colonne_ref) REFERENCES
autre_table(colonne_reference));
```

# CRUD



# SQL

## Insérer des données

Pour insérer des données dans une table, vous pouvez utiliser la commande **INSERT INTO** :

```
INSERT INTO nom_de_la_table (colonne1, colonne2, ...)
VALUES (valeur1, valeur2, ...);
```

```
1 INSERT INTO `utilisateur` (`nom`, `prenom`, `email`)
2 VALUES ('Durantay', 'Quentin', 'quentin@gmail.com');
```

# SQL

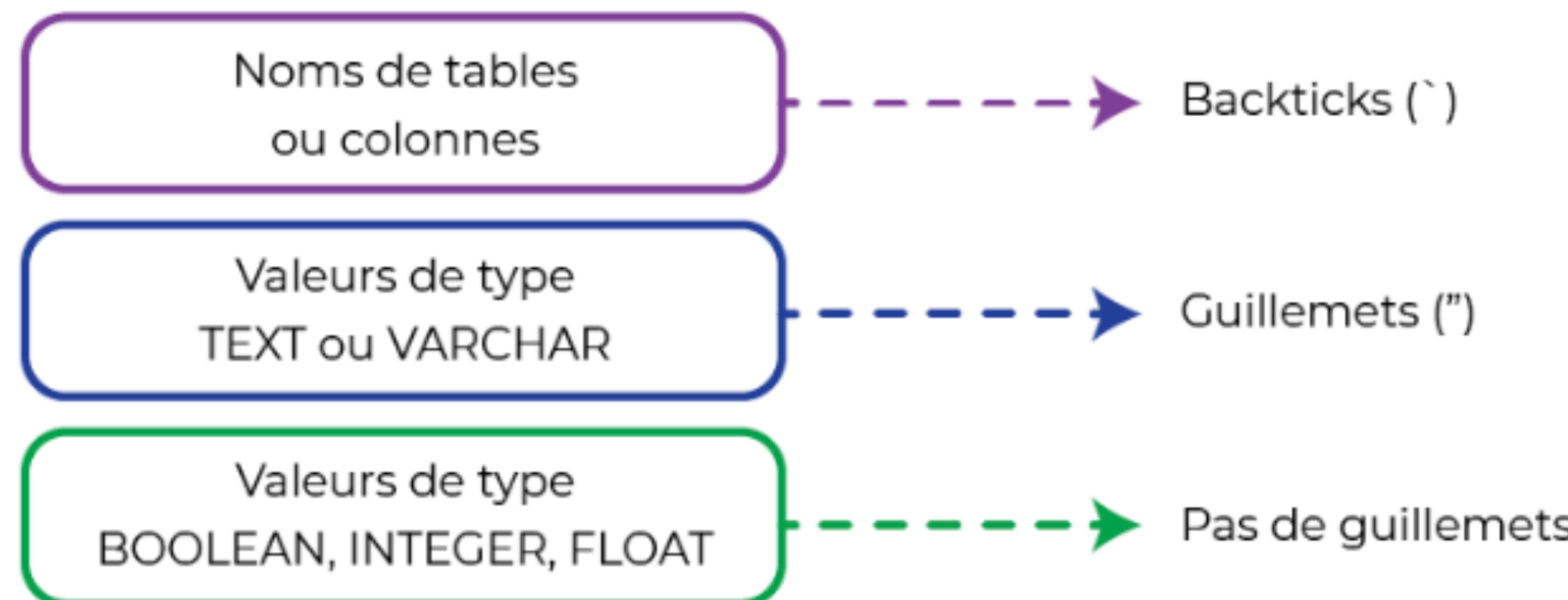
## Insérer des données - plusieurs éléments

```
1 INSERT INTO `utilisateur` (`nom`, `prenom`, `email`)
2 VALUES
3 ('Doe', 'John', 'john@yahoo.fr'),
4 ('Smith', 'Jane', 'jane@hotmail.com'),
5 ('Dupont', 'Sebastien', 'sebastien@orange.fr'),
6 ('Martin', 'Emilie', 'emilie@gmail.com');
```

```
1 INSERT INTO `aliment` (`nom`, `marque`, `sucre`, `calories`, `graisses`, `proteines`, `bio`)
2 VALUES
3 ('poire', 'monoprix', 27.5, 134, 0.2, 1.1, FALSE),
4 ('pomme', 'monoprix', 19.1, 72, 0.2, 0.4, FALSE),
5 ('oeuf', 'carrefour', 0.6, 82, 5.8, 6.9, TRUE),
6 ('lait d\'amande', 'bjorg', 4.5, 59, 3.9, 1.1, TRUE);
```

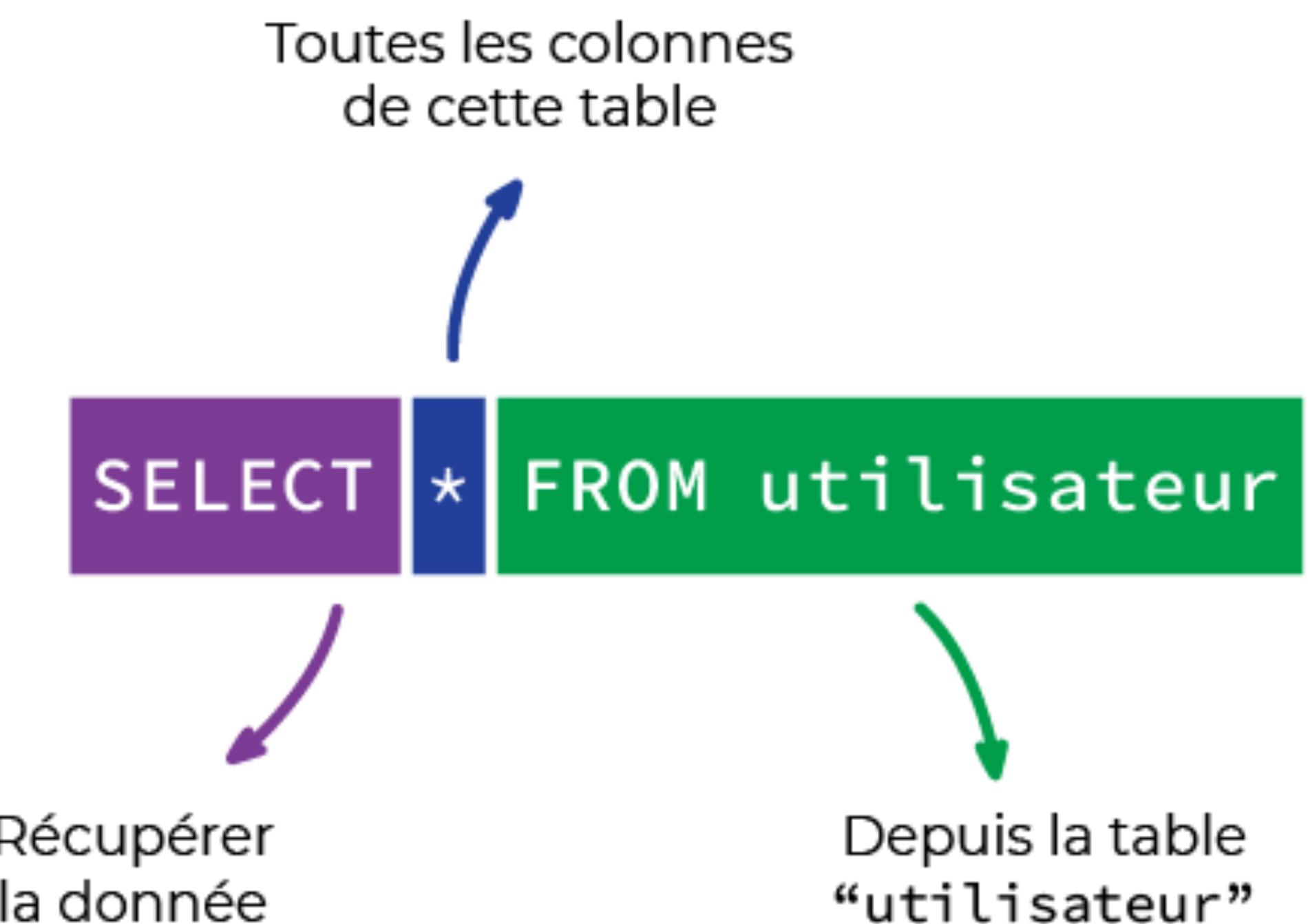
# SQL

Vous vous demandez peut-être pourquoi certaines valeurs sont entre guillemets simples, d'autres avec des backticks et certaines sans rien 🤔 ...



# SQL

## Sélectionner des données



# SQL

## Sélectionner des données

Pour sélectionner des données d'une table, vous pouvez utiliser la commande **SELECT** :

```
SELECT * FROM nom_de_la_table;  
SELECT colonne1, colonne2, ... FROM nom_de_la_table;  
SELECT colonne1, colonne2, ... FROM nom_de_la_table  
WHERE condition;
```

# SQL

## Mettre à jour des données

Mettre à jour  
cette table

Colonne(s) et valeur(s)  
à modifier

```
UPDATE `utilisateur` SET `email` = valeurdelemail
```

```
WHERE `id`
```



Filtrer pour ajouter des conditions

# SQL

## Mettre à jour des données

Pour mettre à jour des données dans une table, vous pouvez utiliser la commande **UPDATE** :

```
UPDATE nom_de_la_table  
SET colonne1 = nouvelle_valeur1, colonne2 =  
nouvelle_valeur2, ...  
WHERE condition;
```

# SQL

## Mettre à jour des données

Voici un exemple concret :

```
UPDATE employes  
SET salaire = 5000, departement = 'RH'  
WHERE id = 1;
```

!! Il est tout à fait possible d'utiliser **UPDATE** sans filtres (sans **WHERE**). Néanmoins, la commande modifierait tous les objets de notre table. C'est très rarement ce que l'on souhaite. 😞

# SQL

## Supprimer des données

Pour supprimer des données d'une table, vous pouvez utiliser la commande **DELETE** :

```
DELETE FROM nom_de_la_table  
WHERE condition;
```

Assurez-vous d'utiliser la clause **WHERE** avec soin pour spécifier les conditions de suppression de manière précise.

**!!** Sinon, vous risquez de supprimer des données indésirables ou de supprimer toutes les lignes de la table par erreur.

# SQL

## Alias sur des champs

Pour utiliser un alias sur une colonne et ainsi renommer temporairement le nom de cette dernière dans le résultat d'une requête, il est possible d'utiliser deux syntaxes.

On peut utiliser la commande **AS** ou non

```
SELECT firstname AS prenom FROM produit;
```

```
SELECT firstname prenom FROM produit;
```

# SQL

## Supprimer des doublons

Pour supprimer les doublons des résultats d'une requête en SQL, il est nécessaire d'utiliser la commande **SELECT DISTINCT** :

```
SELECT DISTINCT prenom FROM etudiant;
```

# SQL

## Opérateurs de comparaisons

Il existe de nombreux opérateurs de comparaisons utilisables au sein de la commande **WHERE**. Le tableau ci-dessous liste les opérateurs les plus couramment utilisés et que nous aborderons dans ce cours :

- =
- != ou <>
- > ou >=
- < ou <=
- IS NULL ou IS NOT NULL

# SQL

## AND et OR

Les opérateurs logiques **AND** et **OR** peuvent être utilisés en complément de la commande **WHERE** pour combiner des conditions :

```
SELECT une_colonne FROM une_table WHERE une_condition AND  
une_autre_condition;  
  
SELECT une_colonne FROM une_table WHERE une_condition OR  
une_autre_condition;
```

# SQL

## BETWEEN

En SQL, l'opérateur **BETWEEN** permet de sélectionner des enregistrements en fonction d'une intervalle. Cet opérateur s'utilise avec la commande **WHERE**. L'intervalle peut être constitué de chaînes de caractères, de nombres ou de dates :

```
SELECT une_colonne FROM une_table WHERE une_colonne  
BETWEEN "valeur_1" AND "valeur_2";
```

# SQL

## IN et NOT IN

En SQL, l'opérateur **IN** permet de sélectionner des enregistrements si la valeur d'une colonne est comprise dans une liste de valeurs. La commande **IN** permet d'éviter d'avoir à recourir à de multiples **OR**. Cet opérateur s'utilise avec la commande **WHERE** :

```
SELECT une_colonne FROM une_table WHERE une_colonne IN  
("valeur_1", "valeur_2", "valeur_3");
```

# SQL

## LIKE

En SQL, l'opérateur **LIKE** permet de faire une recherche suivant un modèle sur les valeurs d'une colonne. Une nouvelle fois, cet opérateur s'utilise avec la commande **WHERE**. C'est un opérateur extrêmement puissant qui offre de nombreuses possibilités :

```
SELECT * FROM clients WHERE email LIKE "%@yeah.com";
```

# SQL

## LIKE

- `%gmail.com` → Texte se terminant par “gmail.com”
- `gmail.com%` → Texte commençant par “gmail.com”
- `%gmail.com%` → Texte contenant “gmail.com” au début ou à la fin

# SQL

## Alias sur une table

Cette méthode permet d'attribuer un autre nom à une table dans une requête **SQL**. Cela peut aider à avoir des noms plus courts, plus simples et plus facilement compréhensibles.

```
SELECT * FROM produit AS p;
```

```
SELECT * FROM produit p;
```

# SQL

## Critères de tri

Utilisez la clause **ORDER BY** pour trier les résultats en fonction d'une colonne spécifiée. Vous pouvez spécifier l'ordre de tri en utilisant « **ASC** » (par défaut pour un tri croissant) ou « **DESC** » (pour un tri décroissant).

```
SELECT nom, prenom, salaire  
FROM employes  
WHERE departement = 'RH'  
ORDER BY salaire DESC;
```

# SQL

## Limit

La clause "**LIMIT**" est une instruction utilisée dans MySQL pour limiter le nombre de résultats renvoyés par une requête.

Elle permet de spécifier un nombre maximum de lignes à récupérer à partir du début des résultats.

```
SELECT * FROM employes  
LIMIT 5;
```

# SQL

## Limit offset

La clause "LIMIT" peut également être utilisée avec un second argument, spécifiant l'index de départ à partir duquel récupérer les résultats. La syntaxe est la suivante ::

```
SELECT * FROM employees  
LIMIT 5, 2;
```

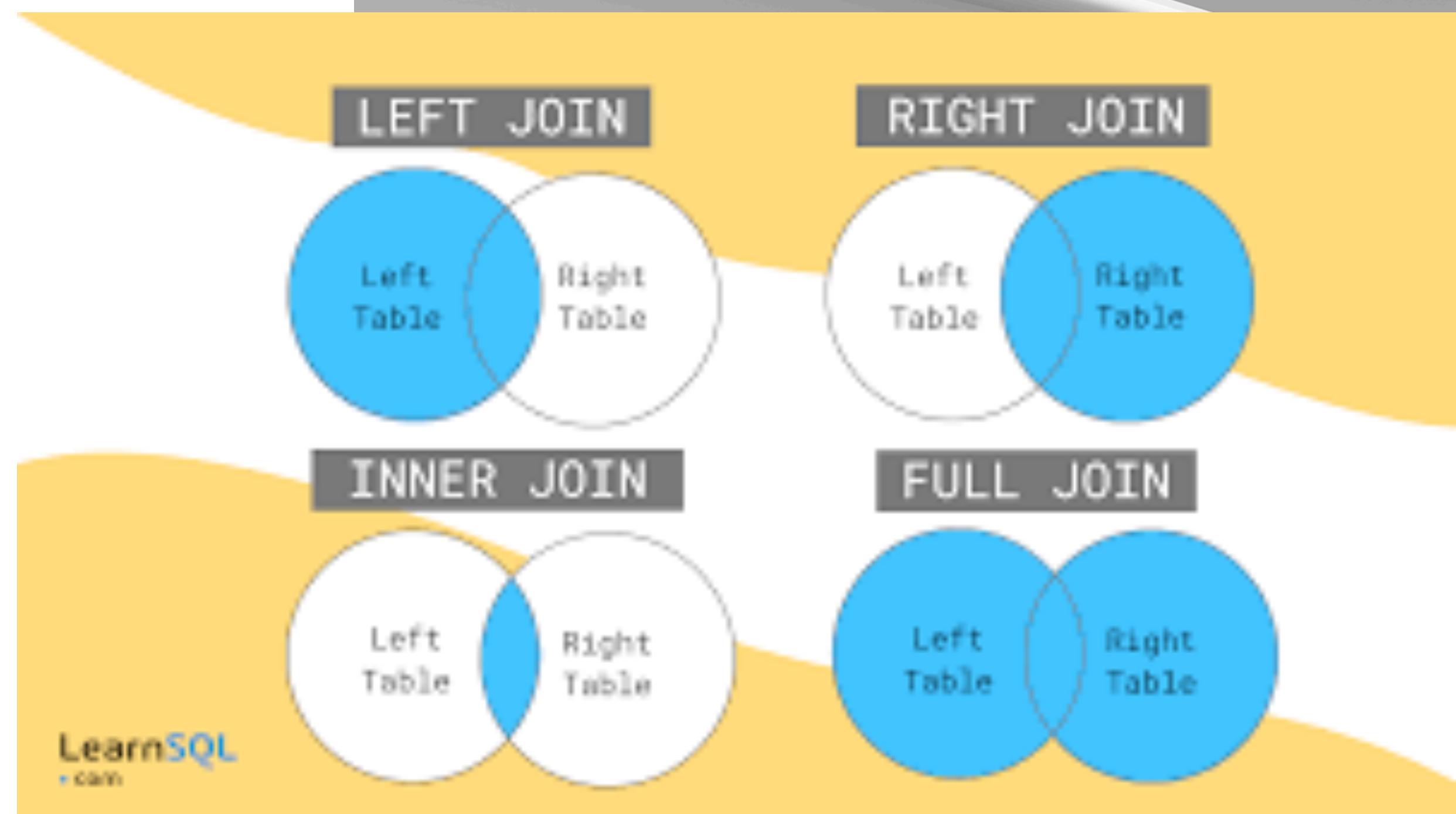
# SQL

## Limit offset

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()** compter le nombre d'enregistrements d'une table ou d'une colonne distincte
- **AVG()** calculer la moyenne sur un ensemble d'enregistrements
- **SUM()** calculer la somme sur un ensemble d'enregistrements
- **MAX()** récupérer la valeur maximum d'une colonne sur un ensemble d'enregistrements
- **MIN()** récupérer la valeur minimum

# Jointure



# SQL

## JOIN

```
SELECT * FROM `utilisateur`
```

→ Sélectionner tous les utilisateurs

```
JOIN `langue`
```

→ Joindre les langues

```
ON `utilisateur`.`langue_id` = `langue`.`id`;
```

→ Relation entre tous les utilisateurs ayant configuré dans une langue spécifique

# SQL

## JOIN

Grâce à cette commande, vous allez pouvoir expliquer à MySQL comment joindre deux tables selon un identifiant qu'elles ont en commun.

Partons du principe que :

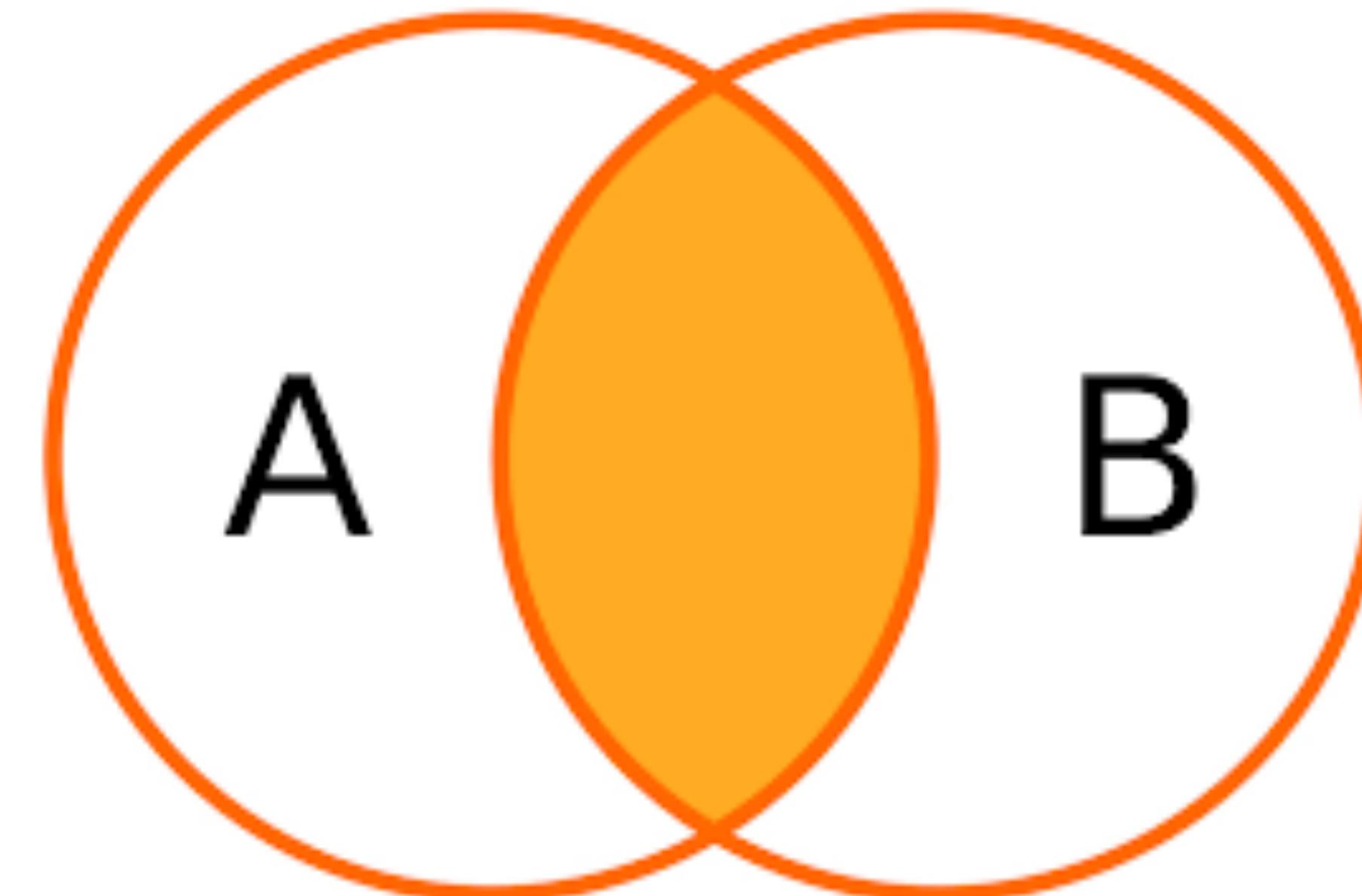
- La langue\_id du premier utilisateur est le français ;
- L'id du français est 1.

Vous allez spécifier à MySQL de joindre les tables “utilisateur” et “langue” en lui précisant que l'id de langue et langue\_id de l'utilisateur doivent être égaux !

# SQL

## INNER JOIN

Dans le langage SQL la commande **INNER JOIN**, est un type de jointure très commune pour lier plusieurs tables entre-elles dans une même requête. Cette commande retourne les enregistrements lorsqu'il y a **au moins une ligne dans chaque colonne** listé dans la condition



# SQL

## INNER JOIN

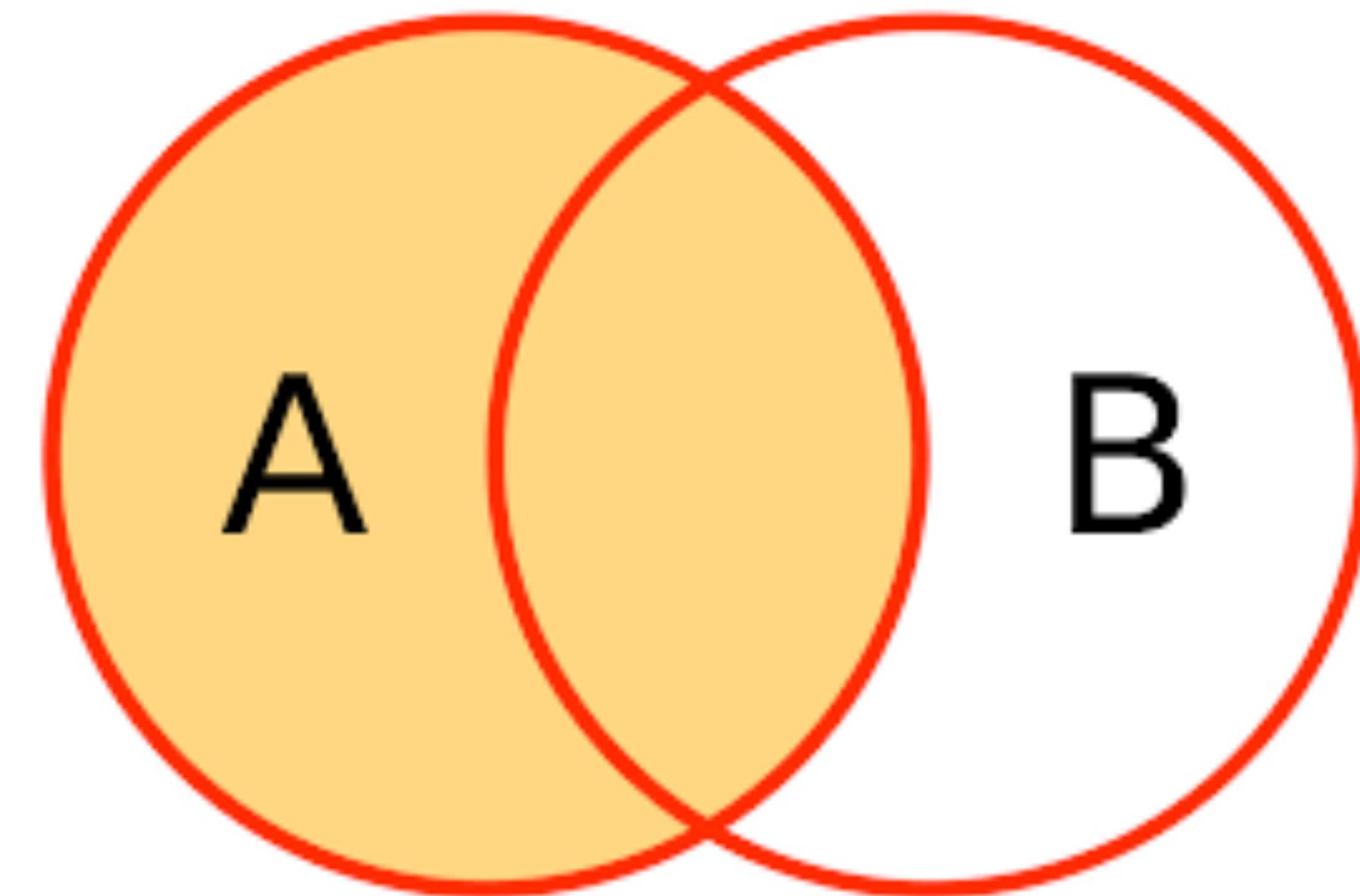
Cette commande retourne les enregistrements lorsqu'il y a au moins une ligne dans chaque colonne listé dans la condition.

```
SELECT *
FROM table_1
INNER JOIN table_2
ON table_1.une_colonne = table_2.autre_colonne;
```

# SQL

## LEFT JOIN

Cette commande retourne tous les enregistrements de la table première table, celle de gauche (left), avec la correspondance dans la deuxième table si la condition est respectée.



# SQL

## LEFT JOIN

En d'autres termes, ce type de jointure permet de retourner tous les enregistrements d'une table avec les données liées d'une autre table si elles existent.

```
SELECT *
FROM table_1
LEFT JOIN table_2
ON table_1.primary_key = table_2.foreign_key;
```

# SQL

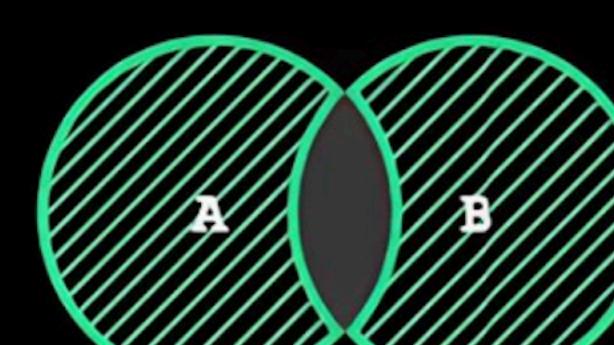
## Autres jointures



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



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



```
1 SELECT *
2 FROM A
3 FULL JOIN B ON A.key = B.key
4 WHERE A.key IS NULL OR
5 B.key IS NULL
```

# SQL

## Autres jointures

The diagram consists of four vertically stacked Venn diagrams, each showing two overlapping circles labeled 'A' and 'B'. The top two diagrams show circle A on the left and circle B on the right, while the bottom two show circle B on the left and circle A on the right. In all diagrams, the intersection of circles A and B is shaded with diagonal lines, representing the joined data.

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

```
1 SELECT *
2 FROM A
3 LEFT JOIN B ON A.key = B.key
4 WHERE B.key IS NULL
```

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

```
1 SELECT *
2 FROM A
3 RIGHT JOIN B ON A.key = B.key
4 WHERE B.key IS NULL
```

# SQL

## Lectures complémentaires

**<https://sql.sh/>** : site officiel SQL

**mariadb** : site officiel MariaDb

**LeetCode 50 SQL** : s'entraîner au SQL

MERCI ! 🙌

