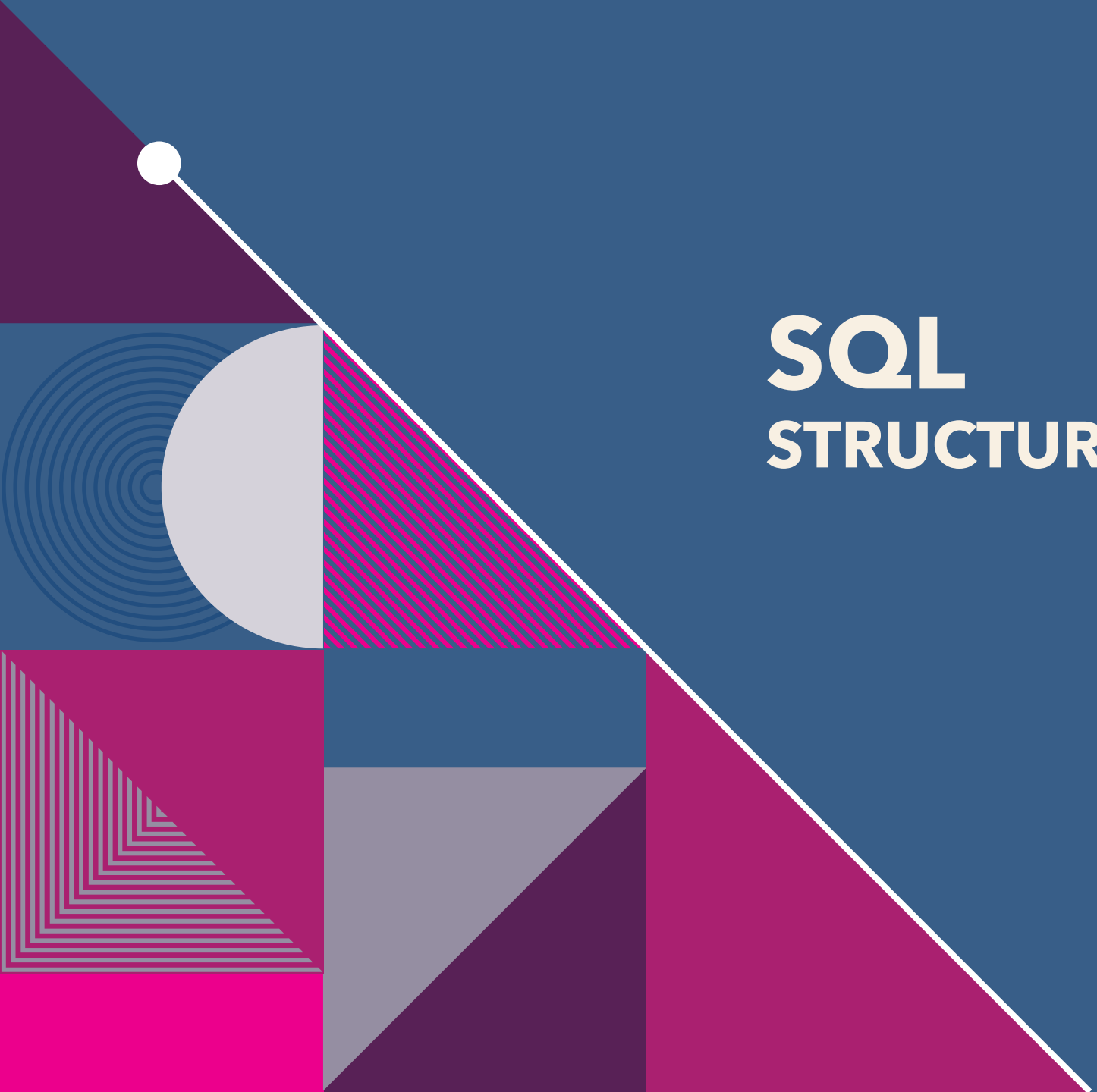


SQL
STRUCTURED QUERY LANGUAGE

SQL
PRÉREQUIS : OUTILS ET LOGICIELS

Pour la suite de cette formation, vous allez devoir installer quelques outils

1. Serveur de Base de Données

   - La solution « simple » sur **Windows** : Installer **Laragon.** Cela vous fournira un serveur LAMP fonctionnel (Apache, MySQL, PHP).

   - La solution d'avenir : utiliser **Docker**, un gestionnaire de conteneurs.
     C'est la solution que vous aurez le plus de chances de croiser en entreprise.

     La courbe d'apprentissage est un peu plus ardue que pour la solution précédente, mais lorsque vous avez une config fonctionnelle, c'est beaucoup plus efficace et souple.

     Docker est une solution **multi-plateformes**, fonctionnant aussi bien sous Windows, Linux ou Mac.

SQL
INSTALLATION DE DOCKER SOUS WINDOWS

1. Installer Docker pour Windows (https://www.docker.com/)
2. Installer WSL 2 (Windows Subsystem for Linux)

   - Ouvrir un terminal powershell et entrer les commande suivante :

     wsl.exe --update

     Puis :

     wsl --set-default-version 2

   - Ajouter votre utilisateur local au groupe docker-users
     - Ouvrez un terminal PowerShell en tant qu'administrateur et entre la commande suivante :
       - net localgroup docker-users "your-user-id" /ADD

   - Fermez votre session et rouvrez là

SQL
INSTALLATION DE DOCKER SOUS WINDOWS

- Créez un répertoire qui sera votre répertoire de travail, puis clonez-y le dépôt git suivant :
  git clone https://github.com/guirod/docker-lamp.git
  Ce dépôt contiendra l'essentiel pour la suite de votre formation (serveur MySQL, serveur Apache, PHP 8.1, serveur SMTP + mailcatcher) et sera mis à jour au besoin
- Quelques commandes utiles :
  - **Lancer et couper vos containers (serveurs) :**
    docker-compose up –d
    docker-compose down
  - **Lister les containers**
    docker container ls
  - **Obtenir une connexion en root sur le container**
    docker exec -it nom_du_container executable
  - **Exemple pour se connecter au container apache et lancer l'exécutable "bash"**
    docker exec -it docker-lamp-docker-lamp-php-1 bash

SQL
INSTALLATION D'UNE GUI (GRAPHIC USER INTERFACE)

- 2 principaux outils, je vous conseille d'installer les 2.
  - MySQL Workbench
    - https://www.mysql.com/fr/products/workbench/
    - Graphiquement plus agréable
    - Parfois un peu lourd et buggué pour certaines tâches (import et export notamment)
  - HeidiSQL
    - https://www.heidisql.com/
    - Un peu moins convivial, mais très puissant.
    - Compatible avec de nombreux SGBDr (MySQL/MariaDB, PostgreSQL, SQL Server)
  - Pour chacun de ces outils, configurez et sauvegarder votre connexion :
    - Host : localhost / 127.0.0.1
    - Username : root
    - Password : p@ssw0rd pour les utilisateurs Docker (configuré dans le fichier compose.yaml)

SQL
CONNEXION EN LIGNE DE COMMANDE

- Se connecter au container MySQL

  docker exec -it docker-lamp-docker-lamp-mysql-1 bash

- Se connecter avec son utilisateur

  mysql -hlocalhost -uroot -pp@ssw0rd (problème, le pass sera dans votre bash history)
  mysql –h localhost –u root –p (un prompt vous demandera votre mdp)

- Enjoy (Non. Mais parfois vous n'aurez pas le choix)

SQL
INTRODUCTION

SQL
INTRODUCTION

- Langage informatique normalisé
- N’est pas un langage procédural
- Créé par IBM en 1974
- Normalisé depuis 1986
  - Recommandation de l’ANSI
  - Norme ISO/CEI 9075 - Technologies de l'information - Langages de base de données SQL

SQL
INTRODUCTION

Le langage SQL permet principalement de faire du CRUD :

- **Create** : Ecrire, insérer des données
- **Read** : Lire des données
- **Update** : Modifier des données
- **Delete** : Supprimer des données

Il permet aussi de modifier la structure de la base de données :

- Créer, modifier, supprimer des bases de données
- Créer, modifier, supprimer des tables
- Créer, modifier, supprimer des utilisateurs et gérer leurs privilèges

SQL
BASE DE DONNÉES RELATIONNELLES

- 3 objectifs :
  - **Garantir l'intégrité des données**
    - Eviter l'altération des données (usure, pannes, erreurs, malveillance)
    - Eviter l'incohérence des données
      - La duplication des données : exemple 2 adresses différentes pour un utilisateur
      - Les valeurs aberrantes : exemple Age < 0
  - **Garantir l'indépendance entre données et traitements**
    - L'information correspondant à une donnée doit être compréhensible indépendamment
    - Si une donnée est calculée, on évitera de la stocker en BDD
  - **Traitements rationalisés**
    - Une fois les données définies, les traitements consistent principalement à ajouter, modifier, supprimer et consulter des données

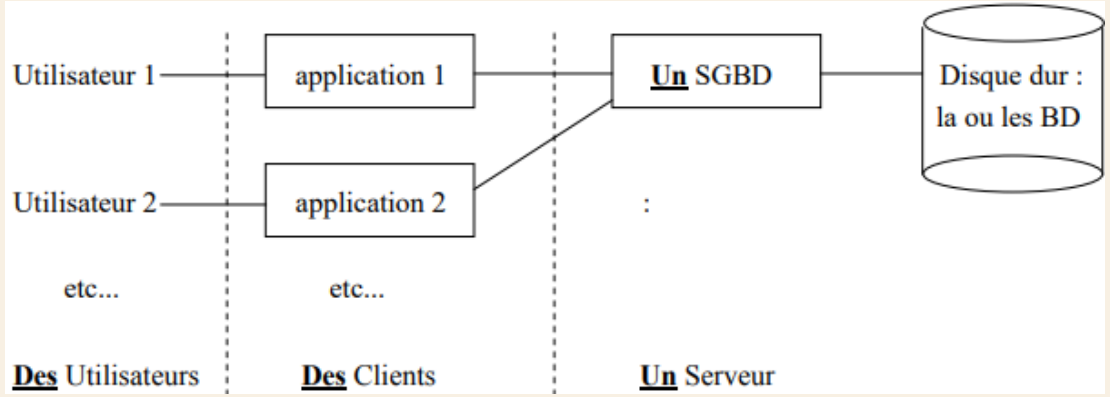
SQL
SGBDR : SYSTÈME DE GESTION DE BASE DE DONNÉES RELATIONNELLES

- Ensemble de logiciels faisant l'interface entre l'utilisateur et la BDD

- **Objectifs** :
  - Stockage et requêtage des données
  - Multi-utilisateurs => Gestion des droits d'accès, des collisions
  - Limiter la redondance (duplication de données) => Intégrité des données

SQL
SGBDR : SYSTÈME DE GESTION DE BASE DE DONNÉES RELATIONNELLES

- **Architecture**
  - Le plus souvent on utilise une architecture client/serveur



  - Une BDD est stockée sur 1 ou plusieurs disques durs
  - 1 seul SGBD par BDD
  - Plusieurs applications / API (clients) peuvent communiquer avec le SGBD

SQL
SGBDR : SYSTÈME DE GESTION DE BASE DE DONNÉES RELATIONNELLES

- L'objectif majeur du SGBD étant d'assurer l'intégrité, voici quelques moyens mis en œuvre
  - Conformité au modèle : Vérifier que les données saisies soient cohérentes
  - Accès concurrents : Eviter les incohérences dues à des modifications multiples au même moment (transactions)
  - Gestion sur panne : Garantir le maintien de la cohérence même quand une panne intervient
  - Autorisations d'accès

SQL
SGBDR : SYSTÈME DE GESTION DE BASE DE DONNÉES RELATIONNELLES

- Les SGBDr les plus connus :
  - MySQL (Oracle) / MariaDB (Fork open source de MySQL)
  - OracleDB
  - PostgreSQL : alternative open source à OracleDB
  - SQL Server (Microsoft)



SQL
ADMINISTRATION D'UNE BDD

SQL
CRÉATION DE BDD

```
CREATE DATABASE [IF NOT EXISTS] db_name [options…]…
options: [DEFAULT] {
        CHARACTER SET [=] charset_name |
        COLLATE [=] collation_name |
        ENCRYPTION [=] {'Y' | 'N'}
}
```

IF NOT EXISTS : Si cette option est présente, cette commande ne lèvera pas d'exception si la BDD existe déjà. Il n'y fera aucune modification.

Les options sont facultatives.
Exemple de requête :
CREATE DATABASE *db_name* CHARACTER SET *utf8mb4* COLLATE *utf8mb4_0900_ai_ci*

Pour plus d'informations : https://dev.mysql.com/doc/refman/8.0/en/create-database.html

SQL
SUPPRESSION ET MODIFICATION DE BDD

- SUPPRESSION

```
DROP DATABASE [IF EXISTS] db_name;
```

- MODIFICATION

```
ALTER DATABASE [db_name] alter_option ...
alter_option: { [DEFAULT] CHARACTER SET [=] charset_name |
                [DEFAULT] COLLATE [=] collation_name |
                [DEFAULT] ENCRYPTION [=] {'Y' | 'N'} |
                READ ONLY [=] {DEFAULT | 0 | 1} }
```

- SELECTIONNER UNE BDD : Nécessaire pour pouvoir effectuer des opérations (création ou consultation)

```
USE db_name;
```

- DIVERS

```
SHOW DATABASES;
```

SQL
GESTION DES DROITS ET UTILISATEURS

- **Sécurité**
  - Limiter au strict nécessaire les droits des utilisateurs exposés (application web)
  - Utiliser des mots de passe forts et uniques
  - Maintenir ses logiciels à jour

SQL
GESTION DES DROITS ET UTILISATEURS

- Création d'un utilisateur
  CREATE USER 'alice'@'localhost' IDENTIFIED BY 'P@ssw0rd';

- Afficher les utilisateurs
  SELECT * FROM mysql.user;

- Assigner les privileges
  GRANT type_of_permission ON database_name.table_name TO 'username'@'host';

  Example :
  GRANT CREATE, ALTER, DROP, INSERT, UPDATE, DELETE, SELECT, REFERENCES, RELOAD
  ON *.* TO 'alice'@'localhost' WITH GRANT OPTION;

- Révoquer un privilege
  REVOKE type_of_permission ON database_name.table_name FROM 'username'@'host'

SQL
GESTION DES DROITS ET UTILISATEURS

- Assigner tous les privileges à un utilisateur
  GRANT ALL PRIVILEGES ON database.table TO 'username'@'host';

- Appliquer les modifications:
  FLUSH PRIVILEGES;

- Afficher les privileges d'un utilisateur
  SHOW GRANTS FOR 'username'@'host';

- Supprimer un utilisateur
  DROP USER 'username'@'localhost';

SQL
EXERCICE

- Coder un script pour :
  Créer un autre utilisateur ‘bob’
  Créer une BDD pour bob lui assignant tous les privileges

- Ressource :
  https://www.digitalocean.com/community/tutorials/how-to-create-a-new-user-and-grant-permissions-in-mysql

SQL
STRUCTURE DES DONNÉES

SQL
LES TABLES

- Les tables sont les objets qui contiennent toutes les données d'une BDD.

- Données organisées dans un tableau à deux dimensions
  - Les lignes sont des tuples, enregistrements ou n-uplets
  - Les colonnes sont des attributs

SQL
LES TABLES

id	nom	prénom	profession	code postal	ville
1	Durand	Michel	Directeur	75016	Paris
2	Dupond	Karine	Secrétaire	92000	Courbevoie
3	Mensoif	Gérard	Commercial	75001	Paris
4	Monauto	Alphonse	Commercial	75002	Paris
5	Emarre	Jean	Employé	75015	Paris
6	Abois	Nicole	Secrétaire	95000	St Denis
7	Dupond	Antoine	Assistant commercial	75014	Paris

SQL
LES TYPES DE DONNÉES

- **Numériques exacts** :
  - **Entiers signés** :
    - INTEGER 4 octets => -2147483648 à +2147483647
    - SMALLINT (2 octets), BIGINT (8 octets)
  - **Décimaux signés** :
    - NUMERIC, DECIMAL
- **Numériques approximatifs** (attention, ils portent bien leur nom : https://floating-point-gui.de/)
  - REAL, DOUBLE PRECISION, FLOAT
- **Chaînes de caractères**
  - Longueur fixe : CHAR
  - Longueur variable : VARCHAR
  - Objets de type caractère : CLOB

SQL
LES TYPES DE DONNÉES

- **Chaînes binaires**
  - Nombre d'octets fixe : BINARY
  - Longueur variable : VARBINARY
  - Binary Large Object : BLOB
- **Booléens**
  - 3 valeurs : true, false, unknown
- **Dates et heures**
  - DATE
  - TIME/TIMESTAMP WITH/WITHOUT TIMEZONE
- **DIVERS (dépendent du SGBDr)**
  - XML, ARRAY, INTERVAL

SQL
OPÉRATIONS SUR LES TABLES

- Création

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    (create_definition,...)
    [table_options]
    [partition_options]
```

Exemple :

```
CREATE TABLE users (
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    name VARCHAR(100),
    email VARCHAR(255) NOT NULL,
    birthdate DATE
);
```

Pour plus de renseignements, notamment sur les options, voir la doc.
https://dev.mysql.com/doc/refman/8.0/en/create-table.html
https://mariadb.com/kb/en/create-table/

SQL
OPÉRATIONS SUR LES TABLES

- Modification :
  - Ajouter/Modifier/Supprimer un attribut
  - Ajouter/Modifier/Supprimer une contrainte
  - Modifier des propriétés de la table

ALTER TABLE *tbl_name*
[*alter_option* [, *alter_option*] ...]
[*partition_options*]

Exemple :
ALTER TABLE *users*
ADD *firstname VARCHAR(100)* AFTER *name;*

Pour plus de renseignements, notamment sur les options, voir la doc.
https://dev.mysql.com/doc/refman/8.0/en/alter-table.html
https://mariadb.com/kb/en/alter-table/

SQL
INDEXES

- Dans chaque table, on peut définir un ou plusieurs indexes.
- Un ROWID est créé pour chaque enregistrement
- Avantage : Requêtes plus rapides
- Inconvénient : Peuvent surcharger le système donc à utiliser avec précaution
- Exemple :

  Création d'un index : Notez que l'on peut le définir sur une ou plusieurs colonnes

  ```
  CREATE INDEX index_name ON table_name ( column1, column2, ...);
  ```

  Suppression d'un index

  ```
  ALTER TABLE table_name DROP INDEX index_name;
  ```

SQL
CONTRAINTES

Les contraintes sont des règles appliquées aux attributs d'une table.
Elles permettent d'améliorer l'intégrité des données de la base.

- Les contraintes les plus communes sont :

  - **NOT NULL** : Contraint à avoir une valeur différente de NULL.
  - **DEFAULT** : Fournit une valeur par défaut à la colonne.
  - **UNIQUE** : Ne permet pas que 2 enregistrements aient la même valeur pour une colonne (email)
  - **CHECK** : Active une condition pour qu'un enregistrement soit valide.
  - **INDEX**
  - **PRIMARY KEY** : La clé primaire permet d'identifier de manière unique chaque enregistrement. Une clé primaire peut être composée, elle fait dans ce cas référence à plusieurs attributs de la table.
  - **FOREIGN KEY** : Une clé étrangère permet de relier 2 tables.

SQL
CONTRAINTES - CLÉS

- **Clé primaire :**
  Une clé primaire permet d'identifier chaque enregistrement d'une table.
  Par définition, elle doit être UNIQUE et NOT NULL.
  Une clé primaire peut être simple (définie par un seul attribut) ou composée de plusieurs attributs.

- **Clé étrangère :**
  Une clé étrangère est une contrainte qui permet d'associer plusieurs tables entre elles tout en assurant l'intégrité des données.
  Une clé étrangère fait référence à une clé primaire d'une autre table.

  A noter :
  - Il est impossible de faire référence à une clé primaire si celle-ci n'existe pas dans la BDD.
  - Il n'est par défaut pas possible de supprimer un enregistrement si sa clé primaire est référencée comme clé étrangère dans une autre table.

SQL
CONTRAINTES – ON DELETE / ON UPDATE

- **Clé étrangère** :

  Si un enregistrement est référencé dans une autre table, le SGBD empêchera la modification ou suppression de la clé primaire.
  Si l'on souhaite forcer la modification / suppression, il est nécessaire de le renseigner lors de la création de la clé secondaire.

  Pour cela on doit utiliser les instructions ON DELETE et ON UPDATE

  - ON DELETE : A la suppression, effectue une action sur tous les enregistrements enfants.
  - ON UPDATE : A la modification, reporte le changement sur tous les enregistrements enfants.

  L'usage le plus courant est d'utiliser ON DELETE CASCADE ou ON UPDATE CASCADE.
  Toutefois, c'est un comportement assez destructeur ce qui n'est pas toujours souhaitable.
  Dans ce cas, on peut utiliser des alternatives, telles que :

  ON DELETE|UPDATE [SET NULL|SET DEFAULT|NO ACTION|CASCADE]

SQL
CONTRAINTES – EXEMPLES

```
CREATE TABLE employes(
        id     INT          NOT NULL,
        nom                 VARCHAR (100)    NOT NULL,
        email               VARCHAR (100)    NOT NULL UNIQUE,
        age                 INT  NOT NULL CHECK (age >= 18),
        salaire  DECIMAL (18, 2) DEFAULT 3000.00,
        PRIMARY KEY (id)
);
CREATE TABLE conges (
        id     INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
        date_debut    DATE    NOT NULL,
        date_fin          DATE    NOT NULL,
        id_employe INT,
        FOREIGN KEY (id_employe) REFERENCES employes(id) ON DELETE CASCADE
        -- Pour pouvoir nommer la contrainte, utiliser la syntaxe suivante
        CONSTRAINT fk_id_employe FOREIGN KEY (id_employe) REFERENCES employes(id)
);
```

SQL
CONTRAINTES – SUPPRESSION

- En ciblant la contrainte (ne fonctionne pas toujours) :

```
ALTER TABLE table_name
    ALTER COLUMN column_name DROP CONSTRAINT;
```

Ex :
```
ALTER TABLE employe
    ALTER COLUMN salaire DROP DEFAULT;
```

- En ciblant la contrainte par son alias (à privilégier)

```
ALTER TABLE table_name DROP CONSTRAINT constraint_alias;
```

SQL
ASSOCIATIONS

Lorsque l'on modélise notre BDD, on va identifier différents types d'associations qui vont lier les tables entre elles.

Bien concevoir ses associations est indispensable pour satisfaire les conditions des formes normales et assurer l'intégrité des données.

On dénombre **5 types de relations** :

- One-to-One
- One-to-Many / Many-to-One
- Many-to-Many
- Self-Reference (association récursive)

SQL
ASSOCIATIONS – ONE TO ONE

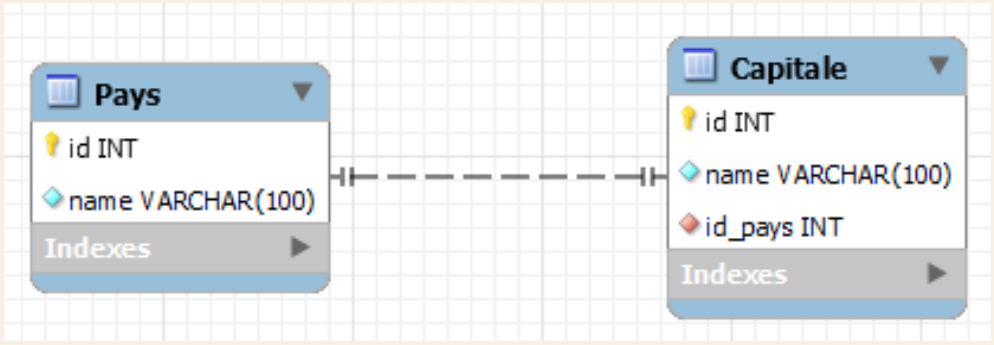
Permet d'associer un et un seul enregistrement d'une table T1 à un et un seul enregistrement d'une table T2.

Implémentée par l'utilisation d'une **clé étrangère** :

- Veiller à ajouter des contrôles (code et/ou BDD) pour assurer l'intégrité des données (ex : contrainte d'unicité sur la clé étrangère)
- Cette association sera rarement utilisée, car généralement, la donnée peut directement être ajoutée dans la table « parent »

Exemples :

- Citoyen <-> Passeport
- Capitale <-> Pays



SQL
ASSOCIATIONS – ONE TO MANY / MANY TO ONE

One-to-Many :

Associe un enregistrement d'une table T1 à de multiples enregistrements d'une table T2

Many-to-One :

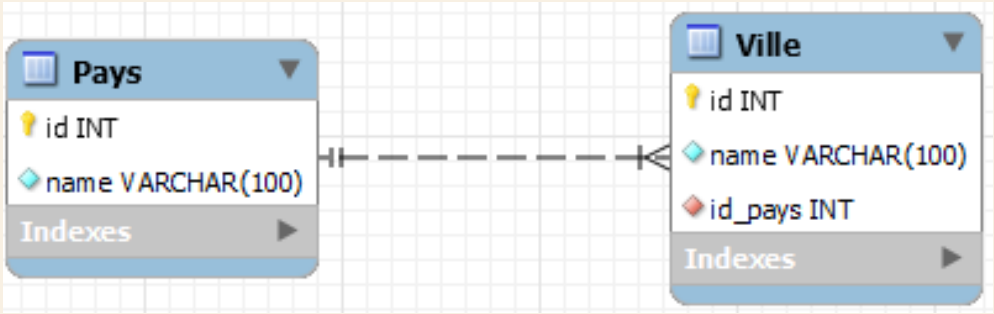
Associe de multiples enregistrements d'une table T1 à un unique enregistrement d'une table T2

Implémentée par l'utilisation d'une clé étrangère.

- Veiller à ajouter des contrôles (code et/ou BDD) pour assurer l'intégrité des données.

Exemple :

- Pays <-> Ville : Un pays possède plusieurs villes, une ville n'appartient qu'à un seul pays



SQL
ASSOCIATIONS – MANY TO MANY

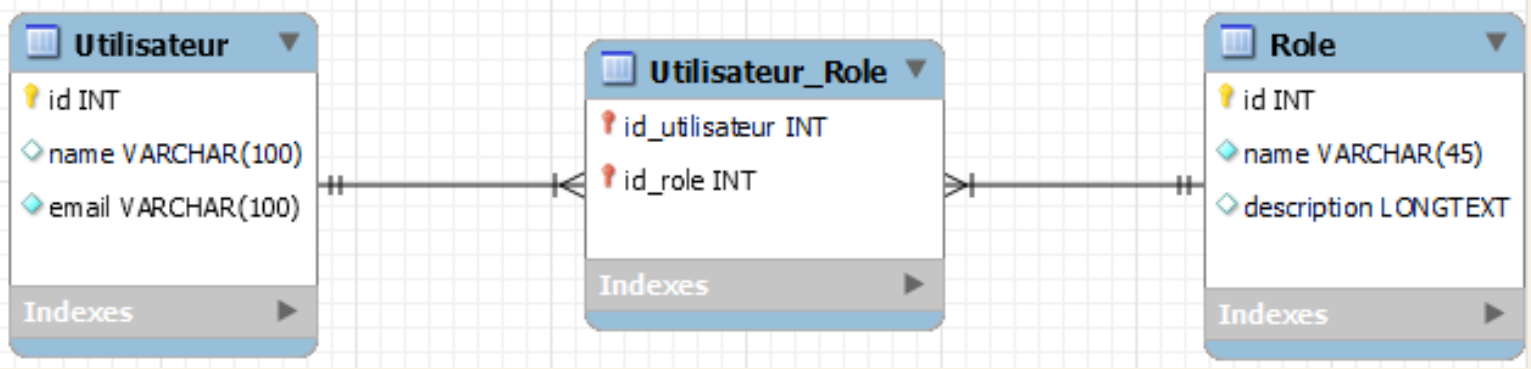
Associe un enregistrement d'une table T1 à de multiples enregistrements de la table T2 et vice versa.

Pour implémenter une association Many-to-Many, il est nécessaire de passer par une **table d'association**.

La table d'association peut aussi être **porteuse de donnée**, par exemple ci-dessous, la table Utilisateur_Role pourrait indiquer la date d'expiration du rôle.

Exemple :

- Utilisateur <-> Rôle : un utilisateur peut avoir plusieurs rôles et un rôle peut être possédé par plusieurs utilisateurs.



SQL
ASSOCIATIONS – SELF REFERENCE

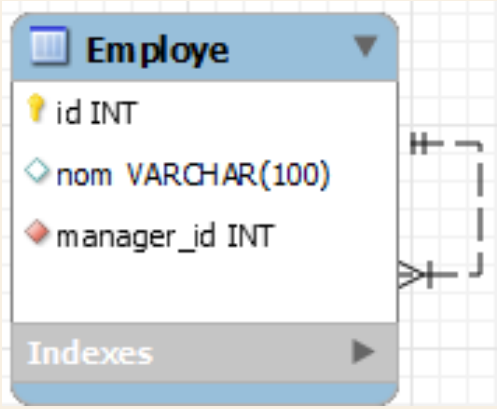
Une association Self-Reference permet d'associer un enregistrement d'une table T à un autre enregistrement d'une table T.

L'association sera souvent une association One-to-Many ou Many-to-One.

C'est notamment utilisé lorsque l'on souhaite gérer une arborescence.

Exemples :

- Catégories <-> Catégorie enfant
- Employé <-> Manager



SQL
RESSOURCES

SQL
RESSOURCES / DOCUMENTATION

- Site officiel :
  https://www.mysql.com/
  https://mariadb.com/

- Documentation officielle :
  https://dev.mysql.com/doc/refman/8.0/en/
  https://mariadb.com/kb/en/documentation/

- Tuto W3Schools :
  https://www.w3schools.com/sql/default.asp

- MySQL Tutorial :
  https://www.mysqltutorial.org/

- Slide sur l'optimisation :
  https://www.slideshare.net/ZendCon/joinfu-the-art-of-sql-tuning-for-mysql-presentation

SQL
TRAVAUX PRATIQUES

SQL
TRAVAUX PRATIQUES

Pour ce TP, il vous sera demandé de fournir un fichier de script SQL répondant aux différentes questions.
Pour les questions 5 et 6, n’hésitez pas à ajouter des commentaires afin d’expliquer votre logique.

N’hésitez pas à demander le PDF de l’énoncé du TP

SQL
TRAVAUX PRATIQUES

1. **Commencer par créer une BDD** “**shop_db**” avec les paramètres par défaut (Moteur de stockage InnoDB et le charset par défaut (DEFAULT COLLATION, en principe UTF-8)

2. **Créer 2 utilisateurs** avec des droits différents
   - Un utilisateur “admin” qui possède **tous les privilèges** sur la BDD **shop_db**
   - Un utilisateur “developer” qui possède uniquement les privilèges **ALTER, CREATE, DROP, INDEX, UPDATE** sur la BDD **shop_db**

SQL
TRAVAUX PRATIQUES

3. **Créer les tables suivantes**
   - Une table **Customer** avec les propriétés suivantes
     - id : INT : la primary key avec incrémentation automatique
     - username : VARCHAR(16) non-nullable et unique
     - email : VARCHAR(255) non-nullable et unique
     - password : VARCHAR(32) non-nullable
     - create_time : TIMESTAMP

   - Une table **Address** avec les propriétés suivantes :
     - id : INT : la primary key avec incrémentation automatique
     - road_number : INT
     - road_name : VARCHAR(100) non-nullable
     - zip_code : CHAR(5) non-nullable
     - city_name: VARCHAR(100) non-nullable
     - country_name : VARCHAR(100) non-nullable

SQL
TRAVAUX PRATIQUES

3. **Créer les tables suivantes**
   - Une table **Order** avec les propriétés suivantes :
     - id : INT : la primary key avec incrémentation automatique
     - ref : VARCHAR(45) unique et non-nullable
     - date : DATE non-nullable
     - shipping_cost : DECIMAL(6,2) avec 0.00 pour valeur par défaut
     - total_amount : DECIMAL(6,2) avec 0.00 pour valeur par défaut

   A l'exécution, il est possible que vous ayez une erreur. En effet, le mot "order" est un mot réservé en SQL. Il est donc nécessaire de l'échapper dans la requête en utilisant le symbole backquote « ` ».

   - Une table **Product** avec les propriétés suivantes
     - ref : CHAR(20) primary key
     - name : VARCHAR (100) non-nullable
     - price : DECIMAL(6,2) non-nullable
     - description : LONGTEXT
     - stock : INT avec 0 pour valeur par défaut

SQL
TRAVAUX PRATIQUES

3. **Créer les tables suivantes**
   - Une table **Order_Product** qui est une table associative de type Many-to-Many entre les tables Order et Product
     - ref_product : Foreign key vers table product
     - id_order : Foreign key vers la table order
     - quantity : INT avec 0 pour valeur par défaut

SQL
TRAVAUX PRATIQUES

4. Modification de la table Address

Si on analyse la table Address, on peut constater qu'elle ne nous permettra pas de respecter les formes normales.

En effet, les attributs concernant la rue, le pays ou la ville vont engendrer de nombreuses duplications de données.

Une bonne pratique pour gérer ce cas est d'extraire ces données dans des tables spécifiques.

Nous allons donc modifier la table Address pour y extraire les données zip_code et city_name dans une table City et la donnée country_name dans une table Country.

Les associations seront les suivantes :

- Une adresse n'est composée que d'une seule ville
- Une ville peut apparaître dans plusieurs adresses.
- Une ville appartient à un seul pays.
- Un pays possède plusieurs villes.

SQL
TRAVAUX PRATIQUES

5. Ajout d'une table Categorie

Nous souhaitons qu'un Product puisse appartenir à plusieurs catégories de produits.

Une catégorie possède un nom (VARCHAR), une description (LONGTEXT), et peut ou non être associée à une catégorie Parente (parent_category_id).

Créer les différentes requêtes permettant d'implémenter ces modifications.

SQL
TRAVAUX PRATIQUES

5. Mise en place d'un système de gestion des stocks et magasins

- Les magasins font office d'entrepôt, le stock de chaque produit est donc rattaché au magasin.
- Les magasins ont un nom et une adresse

Proposez le script permettant l'implémentation de cette modification

SQL
INSERTION ET MODIFICATION DE DONNÉES

SQL
INSERTION DE DONNÉES

Maintenant que nous avons une structure de BDD cohérente, nous pouvons commencer à insérer des données dans notre base.

Pour insérer des données, nous allons utiliser la commande « INSERT INTO».

Nous allons détailler sa syntaxe.

SQL
INSERT INTO

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
       [INTO] tbl_name
       [PARTITION (partition_name [, partition_name] ...)]
       [(col_name [, col_name] ...)]
       { {VALUES | VALUE} (value_list) [, (value_list)] ... }
       [AS row_alias[(col_alias [, col_alias] ...)]]
       [ON DUPLICATE KEY UPDATE assignment_list]
```

Voici le lien vers la documentation officielle :
https://dev.mysql.com/doc/refman/8.0/en/insert.html

Vous pourrez avoir des détails sur les différentes options.

SQL
INSERT INTO

Comme précédemment, la syntaxe semble barbare, mais la syntaxe que nous utiliserons le plus souvent sera beaucoup plus digeste :

```
INSERT INTO tbl_name(col_1, col_2, ...)
VALUES (value_col_1, value_col_2, …);
```

- **tbl_name** correspond au nom de notre table.
- **col_1, col_2, …** : les noms de colonne pour lesquelles nous souhaitons préciser les valeurs de notre enregistrement
- **value_col_1, value_col_2, … :** les valeurs que nous voulons insérer.

Notez qu'il est possible de ne pas préciser la liste des colonnes, mais dans ce cas il faudra envoyer les valeurs pour toutes les colonnes de la table, dans l'ordre. Je ne suis personnellement pas fan de cette syntaxe.

SQL
INSERT INTO

Il est également possible de faire de l'insertion « de masse », en envoyant plusieurs listes de valeurs, séparées d'une virgule. On procéderait de la façon suivante :

```
INSERT INTO tbl_name(col_1, col_2)
VALUES (value_col_1_e1, value_col_2_e1), (value_col_1_e1,
value_col_2_e1), (value_col_1_e1, value_col_2_e1);
```

SQL
INSERT INTO : EXEMPLE

- Insertion d'enregistrement unique :

```
INSERT INTO users(`name`,firstname,email)
VALUES('Cartman', 'Eric', 'eric@cartman.com');
```

- Insertion multiple d'enregistrements :

```
INSERT INTO users (`name`,firstname,email)
VALUES
    ('McCormick', 'Kenny', 'kenny@mccormick.com'),
    ('Broflovski', 'Kyle', 'kyle@broflovski.com'),
    ('Marsh', 'Stan', 'stan@marsh.com'),
    ('Garrison', 'Herbert', 'herbert@garrison.com');
```

SQL

UPDATE : MODIFICATION D'ENREGISTREMENTS

UPDATE [LOW_PRIORITY] [IGNORE] *table_reference*
    SET *assignment_list*
    [WHERE *where_condition*]
    [ORDER BY ...]
    [LIMIT *row_count*]

Documentation : https://dev.mysql.com/doc/refman/8.0/en/update.html

SQL
UPDATE : MODIFICATION D'ENREGISTREMENTS

Ici encore, le plus souvent nous utiliserons une syntaxe allégée :

UPDATE *table_reference*
    SET *col1 = val_col1, col2 = val_col2, …*
    [WHERE *where_condition*]

La clause "where" va nous permettre de cibler des enregistrements particuliers selon des critères.

Lorsque la clause "where" est absente d'une requête d'update, toutes les données de la table seront modifiées en consequence.

Il faut donc être très prudent lors de l'execution de ce genre de requête.

SQL
UPDATE : EXEMPLE D'UTILISATION

Ajout d'une valeur à la colonne « birthdate » de tous les users.

```
UPDATE users
SET birthdate = '1997-08-13';
```

Modification de l'adresse mail d'un utilisateur en le recherchant par son nom et prénom.

```
UPDATE users
SET email = 'eric.cartman@south-park.com'
WHERE `name` = 'Cartman' AND firstname = 'Eric';
```

Pour visualiser les résultats :

```
SELECT * FROM users;
```

SQL

DELETE : SUPPRESSION D'ENREGISTREMENTS

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name [[AS] tbl_alias]
    [PARTITION (partition_name [, partition_name] ...)]
    [WHERE where_condition]
    [ORDER BY ...]
    [LIMIT row_count]
```

Documentation : https://dev.mysql.com/doc/refman/8.0/en/delete.html

SQL
DELETE : SUPPRESSION D'ENREGISTREMENTS

Ici encore, la commande allégée ressemblera à :

DELETE FROM *tbl_name* WHERE *where_condition*

Là aussi, la clause WHERE est facultative. Si elle n'est pas renseignée (et que votre serveur MySQL le permet), toutes les données de la table seront supprimées.

Il faut donc être extrêmement prudent lors de l'utilisation de cette commande.

SQL
DELETE : EXEMPLES

- Suppression d'un enregistrement, ciblé par la primary key :

```
DELETE FROM users WHERE id = 1;
```

- *Suppression d'enregistrements dans une plage d'id :*

```
DELETE FROM users WHERE id BETWEEN 1 AND 3;
```

SQL
DELETE : EXEMPLES

- Nous avons vu que la commande DELETE pouvait purger totalement une table.
- Cela peut parfois être utile. Toutefois, cela ne va pas reset le compteur “auto_increment” utilisé dans les identifiants.
- Si l’on souhaite reset le compteur d’auto_increment, il conviendra d’utiliser la commande “TRUNCATE” :

```
TRUNCATE TABLE users;
```

SQL
OPÉRATEURS LOGIQUES

SQL
OPÉRATEURS LOGIQUES

Nous avons commencé à manipuler quelques opérateurs logiques dans les clauses WHERE.

MySQL permet l'utilisation de nombreux opérateurs logiques. Nous allons en étudier quelques-uns.

SQL
OPÉRATEURS

Opérateur	Description	Datatype concerné
=	Opérateur d'égalité	Tous
<>, !=	Différent	Tous
>	Supérieur	Numérique
<	Inférieur	Numérique
>=	Supérieur ou égal	Numérique
<=	Inférieur ou égal	Numérique
BETWEEN x and y	Compris entre x et y	Numérique, dates
IN(liste de valeurs)	Appartient à la liste de valeurs	Tous
IS NULL	Valeur est null	Tous

SQL
OPÉRATEURS

Opérateur	Description	Datatype concerné
NOT	Opérateur logique de négation Renvoie true il évalue false, et renvoie false si il évalue true Exemple : IS NOT NULL	Opérateur logique
AND	Opérateur **ET logique** Ex : A AND B = true si A = true ET B = true	Opérateur logique
OR	Opérateur **OU logique** EX : A OR B = true si A = true ET/OU B = true	Opérateur logique
XOR	Opérateur **OU exclusif** EX : A XOR B = true si A = true OU B = true mais pas si les deux sont true	Opérateur logique

SQL
OPÉRATEUR « LIKE »

- L'opérateur like est utilisé pour rechercher une chaine de caractères dans une autre.
- Cet opérateur est très utilisé pour filtrer un tableau de résultats.
- Il ne prend pas en compte la casse. Ainsi : « Cheval » LIKE « CHeVaL » vaudra true
- On l'utilise en combinaison avec le symbole '%', qui représente 0, 1 ou n caractères

Opérateur	Description	Datatype concerné
X LIKE Y	La chaine de caractères X devra être identique à Y, à l'exception de la casse.	Chaines de caractères
X LIKE %Y	La chaine Y devra se terminer par la chaine X	Chaines de caractères
X LIKE Y%	La chaine Y devra commencer par la chaine X	Chaines de caractères
X LIKE %Y%	La chaine X devra être comprise dans la chaine Y	Chaines de caractères

SQL
OPÉRATEUR « LIKE » : EXEMPLES

Une démo vaut plus que de long discours.

Je vous invite donc à aller regarder et manipuler l'opérateur LIKE pour connaitre un peu mieux son fonctionnement, il vous sera très utile.

Documentation et exemples :

- https://www.w3schools.com/mysql/mysql_like.asp
- https://sql.sh/cours/where/like

SQL
PRÉCÉDENCE DES OPÉRATEURS

- Comme en mathématiques, les opérateurs n'ont pas tous la même priorité lors de l'évaluation d'une opération logique.
- Voici la liste des opérateurs, de la priorité la plus basse à la plus haute.

:=
||, OR, XOR
&&, AND
BETWEEN, CASE, WHEN, THEN, ELSE
=, <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN
|
&
<<, >>
-, +
*, /, DIV, %, MOD
^
- (unary minus), ~ (unary bit inversion)
- NOT, !
- BINARY, COLLATE

SQL
REQUÊTES DE SÉLECTION

SQL
SELECT

La commande SELECT est la commande qui permettre de nous retourner les enregistrements de nos bases de données.

Cette commande est complète et complexe. Comme précédemment, voici l'extrait de la documentation MySQL.

Cette fois-ci, je vous épargne la documentation officielle, assez indigeste.

Je vous invite toutefois à aller y jeter un œil :

https://dev.mysql.com/doc/refman/8.0/en/select.html

SQL
SELECT

- **Commande SELECT « basique »**

```
SELECT nom_colonne FROM nom_table;
```

Cette commande retournera pour toutes les lignes (enregistrements) de la table, la valeur de la colonne « nom_colonne ».

- **Récupérer la valeur de plusieurs colonnes** : pour cela il suffit de lister toutes les colonnes que nous souhaitons récupérer. Exemple :

```
SELECT `name`, firstname FROM users;
```

- **Récupérer la valeur de toutes les colonnes** pour tous les enregistrements de la table :

```
SELECT * FROM users;
```

SQL
SELECT

- Retourner des enregistrements en supprimant les doublons : SELECT DISTINCT

```
SELECT DISTINCT `name` FROM users;
```

- Voyons maintenant une partie des différentes commandes que peut comprendre un SELECT. Attention, leur ordre est important !

```
SELECT *
FROM table
WHERE condition
GROUP BY expression
HAVING condition
{ UNION | INTERSECT | EXCEPT }
ORDER BY expression
LIMIT count
OFFSET start
```

SQL
SELECT

- **Clause WHERE**

C'est la clause que vous utiliserez le plus. Elle permet de filtrer les résultats de la requête en fonction de certains critères.

Par exemple :

```
SELECT * FROM Personne
WHERE date_naissance BETWEEN '2000-07-02' AND '2014-10-3';
```

Cette requête retournera toutes les personnes dont la date de naissance est comprise entre le 02/07/2000 et le 03/10/2014.

Il est évidemment possible d'utiliser plusieurs critères dans une clause WHERE :

```
SELECT * FROM Personne
WHERE date_naissance BETWEEN '2000-07-02' AND '2014-10-3'
AND (email LIKE '%@gmail.com' OR email LIKE '%@hotmail.fr');
```

Cette requête retournera toutes les personnes dont la date de naissance est comprise entre le 02/07/2000 et le 03/10/2014, et qui possèdent une adresse gmail.com ou hotmail.fr (notez l'utilisation des parenthèses).

SQL
SELECT

- **Clause ORDER BY**

Cette clause permet de trier les résultats selon certaines colonnes. Pour choisir l'ordre du tri, nous utiliserons les paramètres ASC (croissant) et DESC (décroissant).

Par exemple :

```
SELECT * FROM Personne
ORDER BY date_naissance ASC;
```

Cette requête retournera tous les enregistrements de la table Personne, triés par la date de naissance (ordre croissant).

On peut aussi trier selon plusieurs colonnes :

```
SELECT * FROM users
ORDER BY `name` DESC, firstname DESC;
```

Ici, nous trions les résultats, en priorité par nom (ordre décroissant), puis par prénom (décroissant également).

SQL
SELECT

- **Clauses LIMIT et OFFSET**

La clause LIMIT permet de limiter le nombre de résultats retournés.

La clause OFFSET permet de retourner les résultats, à partir d'un index déterminé par l'offset.

La conjonction de ces 2 commandes est notamment utilisée pour gérer la pagination d'un tableau de résultats.

```
SELECT * FROM users
LIMIT 5;                    // Retourne les 5 1ers résultats uniquement
                            // Nous retournera les enregistrements des id 1 à 5

SELECT * FROM users
LIMIT 5
OFFSET 5;                   // Retourne les 5 1ers résultats, à partir du 5ème
                            // Retournera les résultats des enregistrements 6 à 10
```

SQL
SELECT : FONCTIONS D'AGGRÉGATION

- Les fonctions d'agrégation sont les fonctions qui d'une façon ou une autre, regrouper plusieurs lignes en une seule.

- Les fonctions d'agrégation les plus utilisées sont les suivantes :
  - **AVG(nom_colonne)** : calcule la moyenne de plusieurs enregistrements
  - **COUNT(nom_colonne)** : retourne le nombre total de lignes retournées
  - **MAX(nom_colonne)** : retourne la valeur maximale d'une colonne
  - **MIN(nom_colonne)** : retourne la valeur minimale d'une colonne
  - **SUM (nom_colonne)** : retourne la somme des enregistrements d'une colonne

SQL
SELECT : FONCTIONS D'AGGRÉGATION

- **Exemples avec injection de données aléatoires :**

```
CREATE TABLE test (
id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
val INT
);

INSERT INTO test(val)
VALUES (round(RAND()*100)),(round(RAND()*100)),(round(RAND()*100)),
(round(RAND()*100)),(round(RAND()*100)),(round(RAND()*100)),
(round(RAND()*100)),(round(RAND()*100)),(round(RAND()*100)),
(round(RAND()*100)),(round(RAND()*100)),(round(RAND()*100));

SELECT COUNT(*) FROM test;
SELECT SUM(val) FROM test;
SELECT AVG(val) FROM test;
SELECT MIN(val) FROM test;
SELECT MAX(val) FROM test;
```

SQL
SELECT : FONCTIONS D'AGGRÉGATION

- **Clause GROUP BY :**
- Lorsque l'on souhaite récupérer d'autres données de notre table, et utiliser une fonction d'agrégation, il sera nécessaire d'utiliser la clause GROUP BY, qui va regrouper les résultats selon une colonne.
- Imaginons la requête suivante :

```
SELECT nomclient, SUM(product_price) FROM orders;
```

- Si nous exécutons cette commande, nous aurons une erreur SQL disant qu'il manque la clause GROUP BY. En effet, la fonction SUM retournant une seule ligne de résultat, mais pas le « SELECT nomclient » (il y a plusieurs clients), MySQL ne sait pas quel client nous retourner.

- Il va donc falloir lui indiquer clairement que nous souhaitons regrouper les résultats par nomclient. Du coup, la SUM calculée, sera celle concernant chaque client et non la SUM globale de la table.

```
SELECT nomclient, SUM(product_price) AS total_order
FROM orders
GROUP BY nomclient;
```

SQL
SELECT : FONCTIONS D'AGGRÉGATION

- **Créer un alias : clause AS**

- Si on reprend la requête précédente, on peut aussi noter l'utilisation de la clause AS.
- Cette clause permet de définir un alias à la colonne récupérée. Ici, on renomme donc « SUM(product_price) » en « total_order »

```
SELECT nomclient, SUM(product_price) AS total_order
FROM orders
GROUP BY nomclient;
```

- Ce nouveau nom de colonne sera celui qui sera retourné à notre logiciel d'administration de BDD (MySQL Workbench, HeidiSQL), mais sera aussi celui retourné à notre programme PHP.

SQL
SELECT : FONCTIONS D'AGGRÉGATION

- **Conditions sur les agrégations : clause HAVING**

- Retraçons le cheminement d'une requête SQL avec agrégation :
  1. Sélection des colonnes qui seront retournées
  2. Filtrage des lignes retournées par la clause WHERE
  3. Agrégation des résultats selon une fonction
- Une requête SQL ne pouvant posséder 2 clauses WHERE, comment faire un filtre sur le retour d'une fonction d'agrégation ?
- Imaginons que nous souhaitions retourner uniquement les clients dont le montant total de la commande soit > 500. Pour cela, on va utiliser la clause HAVING, qui est l'équivalent d'une clause WHERE qui sera exécutée à la fin de notre requête, après l'agrégation des résultats.

```
SELECT nomclient, SUM(product_price) AS total_order
FROM orders
GROUP BY nomclient
HAVING total_order > 500;
```

SQL
SELECT : FONCTIONS D'AGGRÉGATION

- **Exemple GROUP BY et HAVING**

```
CREATE TABLE test2 (
id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
nomclient VARCHAR(50),
val INT
);

INSERT INTO test2(nomclient,val)
VALUES
('client_1', round(RAND()*100)),('client_1', round(RAND()*100)),('client_1', round(RAND()*100)),
('client_1', round(RAND()*100)),('client_1', round(RAND()*100)),('client_1', round(RAND()*100)),
('client_1', round(RAND()*100)),('client_1', round(RAND()*100)),('client_1', round(RAND()*100)),
('client_2', round(RAND()*100)),('client_2', round(RAND()*100)),('client_2', round(RAND()*100)),
('client_2', round(RAND()*100)),('client_2', round(RAND()*100)),('client_2', round(RAND()*100)),
('client_2', round(RAND()*100)),('client_2', round(RAND()*100)),('client_2', round(RAND()*100));
```

SQL
SELECT : FONCTIONS D'AGGRÉGATION

- **Exemple GROUP BY et HAVING**

```
SELECT nomclient,SUM(val) AS total_order
FROM test2
GROUP BY nomclient;

SELECT nomclient,SUM(val) AS total_order
FROM test2
GROUP BY nomclient
HAVING total_order < 500;          // Adaptez la valeur en fonction des données de votre table
```

SQL
JOINTURES

SQL
JOINTURES

Les jointures en SQL permettent d'associer plusieurs tables dans une même requête.

Cela permet d'exploiter la puissance des bases de données relationnelles pour obtenir des résultats qui combinent les données de plusieurs tables de manière efficace.

Exemples d'utilisation :

- Imaginons qu'une base de 2 données possède une table "utilisateur" et une table "adresse" qui contient les adresses de ces utilisateurs. Avec une jointure, il est possible d'obtenir les données de l'utilisateur et de son adresse en une seule requête.
- On peut aussi imaginer qu'un site web possède une table pour les articles (titre, contenu, date de publication …) et une autre pour les rédacteurs (nom, date d'inscription, date de naissance …). Avec une jointure il est possible d'effectuer une seule recherche pour afficher un article et le nom du rédacteur. Cela évite d'avoir à afficher le nom du rédacteur dans la table "article".

SQL
TYPES DE JOINTURES

Il existe différents types de jointures. Voici les jointures les plus communes :

- **INNER JOIN** : jointure interne pour retourner les enregistrements quand la condition est vrai dans les 2 tables. C'est l'une des jointures les plus communes.
- **LEFT JOIN (ou LEFT OUTER JOIN)** : jointure externe pour retourner tous les enregistrements de la table de gauche (LEFT = gauche) même si la condition n'est pas vérifié dans l'autre table.
- **RIGHT JOIN (ou RIGHT OUTER JOIN)** : jointure externe pour retourner tous les enregistrements de la table de droite (RIGHT= droite) même si la condition n'est pas vérifié dans l'autre table.
- **FULL JOIN (ou FULL OUTER JOIN)** : jointure externe pour retourner les résultats quand la condition est vrai dans au moins une des 2 tables.
- **FULL JOIN (ou FULL OUTER JOIN)** : jointure externe pour retourner les résultats quand la condition est vrai dans au moins une des 2 tables.
- **SELF JOIN :** permet d'effectuer une jointure d'une table avec elle-même comme si c'était une autre table.

SQL
TYPES DE JOINTURES

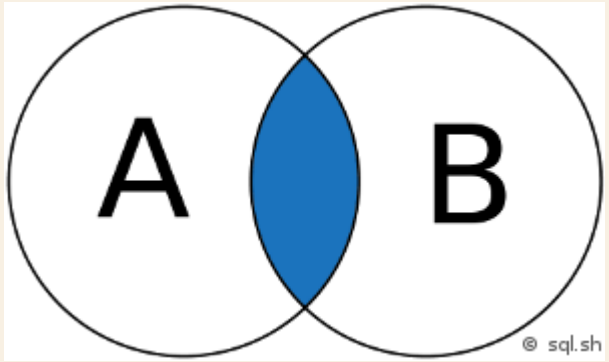
Voici quelques jointures moins utilisées. Nous ne les détaillerons pas ici.

Pour plus d'informations, allez voir ici : https://sql.sh/cours/jointures

- **CROSS JOIN** : Produit cartésien de 2 tables : va joindre chaque ligne d'une table avec chaque ligne d'une autre table => renvoie de nombreux résultats.
- **NATURAL JOIN** : jointure naturelle entre 2 tables s'il y a au moins une colonne qui porte le même nom entre les 2 tables SQL
- **UNION JOIN** : jointure d'union

SQL
INNER JOIN

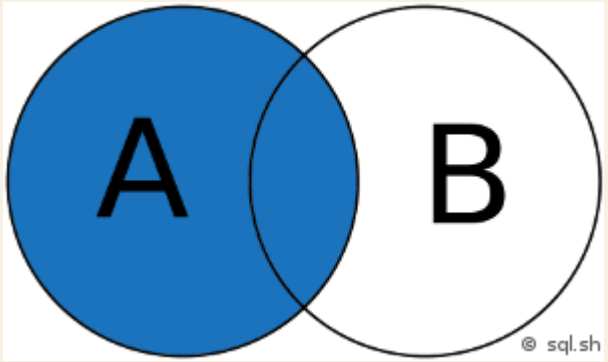
Cette jointure retournera l'intersection de 2 tables.




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

SQL
LEFT JOIN

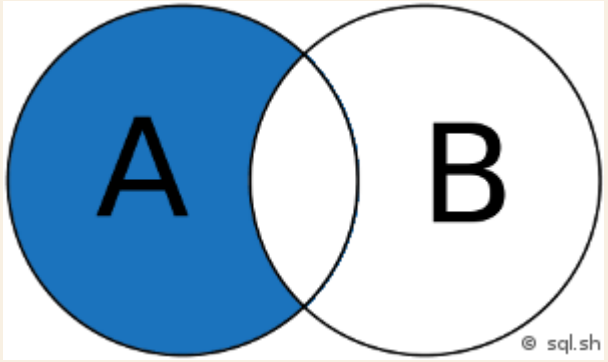
Retourne tous les résultats de A et l'intersection des résultats de A et B




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

LEFT JOIN sans intersection :

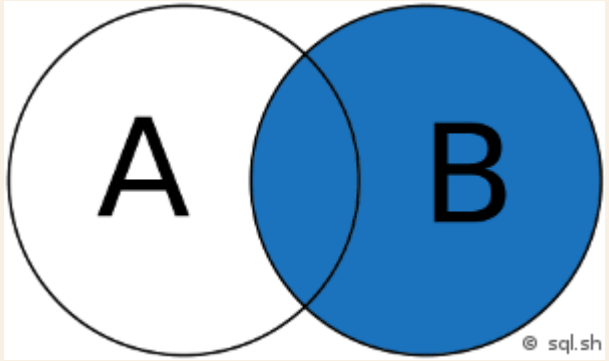
Retourne tous les résultats de A excluant l'intersection de A avec B.




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

SQL
RIGHT JOIN

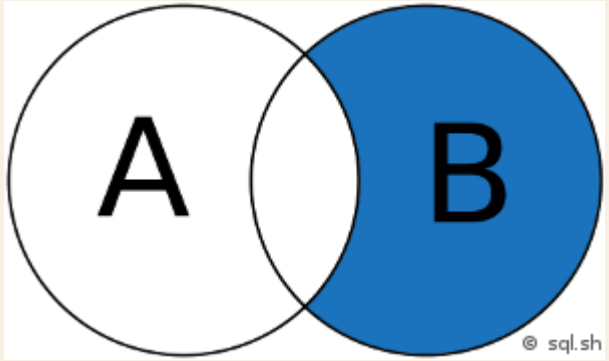
Retourne tous les résultats de B et l'intersection des résultats de A et B




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

RIGHT JOIN sans intersection :

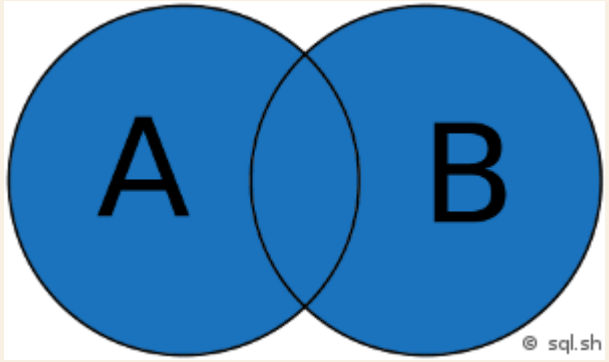
Retourne tous les résultats de B excluant l'intersection de A avec B.




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

SQL
FULL JOIN

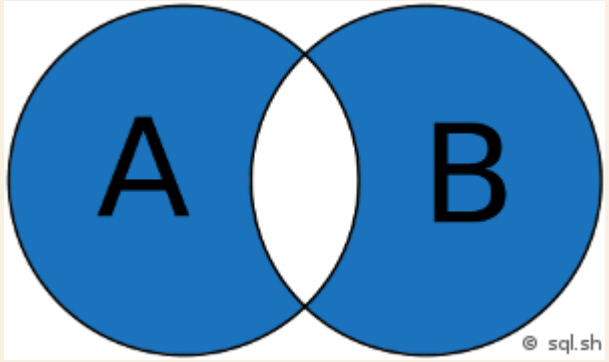
Retourne tous les résultats de B et l'intersection des résultats de A et B




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

FULL JOIN sans intersection :

Retourne tous les résultats de A et B excluant l'intersection de A avec B.




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