MySQL

niveau 1

Qu’est ce qu’une base de donnés

Une base de données est une collection organisée et structurée de données. Elle permet de stocker, gérer et récupérer des informations.

Elle est utilisée par les organisations comme méthode de stockage, de gestion et de récupération de l’information.

Les données sont organisées en lignes, colonnes et tableaux et sont indexées pour faciliter la recherche d'informations. Les données sont mises à jour, complétées ou encore supprimées au fur et à mesure que de nouvelles informations sont ajoutées.

Les données sont organisées dans des tables

id	prenom	nom	sexe	date_naissance	classe
1	Gabriel	Morin	Garçon	2007-10-04	3
2	Léa	Benoit	Fille	2005-12-30	1
3	Tiago	Morel	Garçon	2008-10-31	4
4	Emma	Roy	Fille	2011-06-04	6
5	Louise	Lacroix	Fille	2005-09-08	1
6	Aaron	Leclercq	Garçon	2012-06-12	7

Qu’est-ce qu’un SGBDR ?

Un SGBDR est un Système de Gestion de Base de Données Relationnelles.

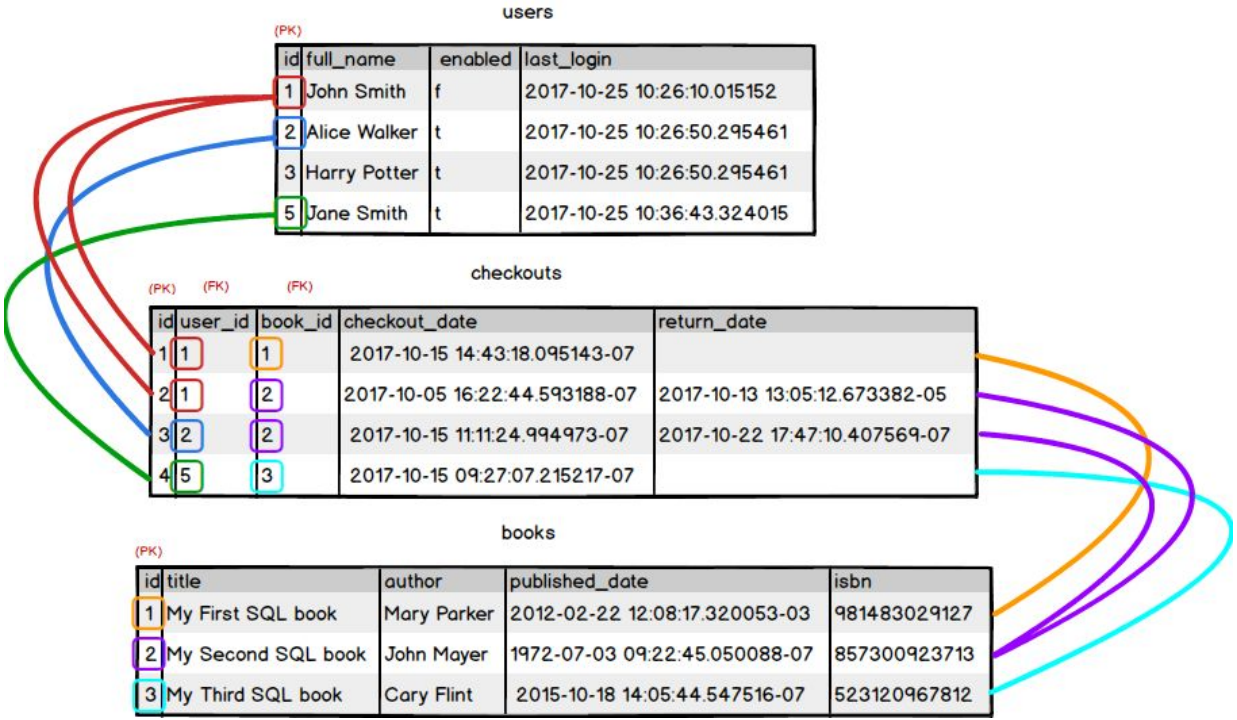
Il s'agit d'un logiciel qui permet de stocker, gérer et interroger les données stockées dans une base de données relationnelle.

Les bases de données relationnelles sont des bases de données qui organisent les données en tables, où chaque table contient des enregistrements (lignes) et chaque enregistrement contient des champs (colonnes).

Certaines tables peuvent avoir des relations avec d'autres tables. Cette relation se traduit par un champ (une colonne) contenant des données similaires dans 2 ou plusieurs tables.

Les acteurs majeurs des bases de données sont: Oracle , Microsoft (SQL Server), IBM (DB2) et PostgreSQL.

Relation entre les différentes tables de données



Architecture base de données

Une architecture de données comprend généralement trois types de modèle de données

- Le modèle conceptuel.
- Le modèle logique
- Le modèle physique.

Le modèle de données conceptuel offre une vue d'ensemble de ce que le système contiendra, de la façon dont il sera organisé et des règles de gestion impliquées.

Le modèle de données logique décrit les entités, les attributs, les relations et les contraintes de manière plus détaillée.

Le modèle de données physique spécifie comment les données sont stockées dans les structures de données sous-jacentes, telles que les tables, les colonnes, les indexes, etc.

Confomité ACID

Pour garantir la cohérence des données et l'intégralité des transactions, toutes les transactions réalisées sur une base de données doivent répondre aux exigences de la conformité ACID.

Atomicité (Atomicity)

L'atomicité garantit que chaque transaction est une unité indivisible.
Une transaction se fait en entier ou pas du tout.

Exemple de transaction suite à un virement:

Retirer 100 € du compte de John
Ajouter 100 € au compte de Jane

Confomité ACID

Cohérence (Consistency)

La cohérence garantit que chaque transaction amène la base de données d'un état valide à un autre état valide. Cela signifie que les contraintes et les règles de la base de données sont toujours respectées.

Exemples:

Un identifiant client doit être toujours unique (pour une table client).

Si une table des factures référence un client n°CL123, je dois pouvoir retrouver cette référence dans la table des clients.

Le prix d'un produit ne peut être négatif (contrainte de valeur pour le champ prix).

Confomité ACID

Isolation

L'isolation assure que les transactions s'exécutent indépendamment les unes des autres, sans interférence. Cela empêche les transactions concurrentes d'accéder ou de modifier les données d'autres transactions en cours.

Durabilité (Durability)

La durabilité garantit que les résultats d'une transaction réussie sont persistants et ne seront pas perdus. Une fois qu'une transaction est confirmée, ses effets sont enregistrés de manière permanente dans la base de données.

Les principaux objets d'un SGBDR

Les bases de données

Les bases de données sont des conteneurs qui regroupent des objets, tels que tables, vues, procédures stockées, etc.

Une instance d'un SGBDR peut contenir plusieurs bases de données.

Tables

Les tables sont des structures qui stockent des données dans une base de données. Elles sont composées de colonnes (champs) et de lignes (enregistrements). Elles sont similaires aux feuilles Excel.

Chaque colonne définit un type de données, tandis que chaque ligne représente un enregistrement contenant des valeurs spécifiques.

Les principaux objets d’un SGBDR

Vues (Views)

Les vues sont des requêtes pré-enregistrées qui permettent d'accéder à des données spécifiques d'une ou plusieurs tables. Elles simplifient l'accès aux données en fournissant une représentation virtuelle des données stockées dans la base de données.

Indexes

Les indexes sont des structures de données qui accélèrent la recherche et la récupération de données dans une table. Ils améliorent les performances en permettant au SGBD de localiser rapidement les enregistrements correspondant à une condition de recherche spécifique.

Les principaux objets d’un SGBDR

Procédures (Stored Procedures)

Les procédures stockées sont des blocs de code précompilés qui peuvent être appelés et exécutés à partir d'une application ou d'une requête SQL. Elles peuvent prendre des arguments en entrée. Elles permettent d'effectuer des opérations complexes de manière réutilisable.

Fonctions (Functions)

Les fonctions sont similaires aux procédures stockées, mais elles retournent une valeur. Elles peuvent prendre des arguments en entrée et renvoient un résultat basé sur ces arguments.

Les principaux objets d’un SGBDR

Déclencheurs (Triggers)

Les déclencheurs sont des instructions SQL ou PL/SQL qui sont exécutées automatiquement en réponse à certains événements tels que l'insertion, la mise à jour ou la suppression de données dans une table. Les déclencheurs permettent d'automatiser des actions en cas de modification des données.

Qu’est-ce que SQL

Le langage SQL (Structured Query Language) est un langage de requête permettant de stocker et de traiter des informations dans une base de données relationnelle.

Créé en 1974, normalisé depuis 1986, le langage est reconnu par la grande majorité des systèmes de gestion de bases de données relationnelles (SGBDR) du marché.

Catégorisation des commandes SQL

Les requêtes SQL peuvent être classifiées en 5 catégories

DDL – Data Definition Language. Ce sont des commandes qui sont utilisées pour créer ou modifier des structures en base de données (CREATE, DROP, ALTER, TRUNCATE, COMMENT, RENAME).

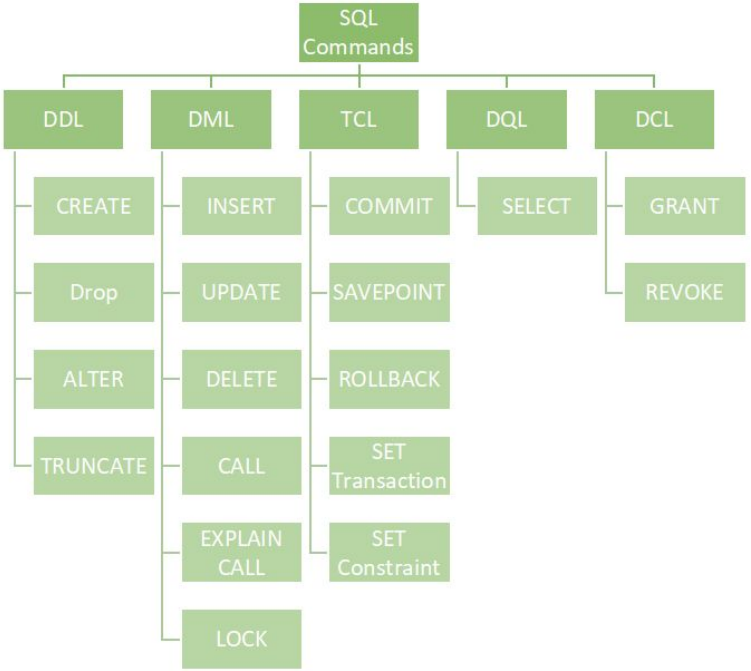
DML – Data Manipulation Language. Commandes utilisées pour la manipulation des données existantes dans la base (INSERT, UPDATE, DELETE).

DQL – Data Query Language. Commandes utilisées pour effectuer des requêtes en base de données (SELECT).

DCL – Data Control Language. Commandes utilisées pour gérer les droits et les permissions des utilisateurs (GRANT, REVOKE)

TCL – Transaction Control Language. Commandes utilisées pour le contrôle de l'exécution d'une transaction (BEGIN, COMMIT, ROLLBACK, SAVEPOINT).

Les 5 catégories des requêtes SQL



Première requête SQL

Voici un exemple d'une simple requête affichant les mots "Hello World!"

```
SELECT 'Hello World!' as Result;
```

Vous pouvez commenter vos fichiers de commandes SQL à l'aide des caractère suivant

- -- ou # pour commenter une ligne ou le reste de la ligne.
- Mettre les commentaires entre /* et */ pour les commentaires sur plusieurs lignes.

Exemple

```
/* Ce commentaire est
sur plusieurs lignes */
SELECT 'Hello World!' as Result;  -- affiche Hello World!
```

Introduction aux requêtes DDL

Les requêtes DDL, ou langage de définition de données, sont un ensemble de commandes SQL utilisées pour définir, modifier et gérer la structure des bases de données.

Les requêtes DDL permettent

- de créer des tables
- de modifier leur structure (ajouter, supprimer ou modifier des colonnes)
- de définir des contraintes d'intégrité (comme les clés primaires et les clés étrangères)
- de créer des indexes

Ainsi que d'autres opérations liées à la gestion de la structure de la base de données.

Création et suppression d'une base de données

Syntaxe pour la création d'un base de données
CREATE DATABASE ma_bdd;

Syntaxe pour la suppression d'une base de données
DROP DATABASE IF EXISTS ma_bdd;

Le mot clé `IF EXISTS` est facultatif. Il permet de ne pas générer d'erreurs si la base de données n'existe pas.

La visualisation des base de données d'une instance MySQL
SHOW DATABASES;

Pour voir sur quelle base je suis actuellement connecté
SELECT DATABASE();

Visualisation des tables

Voici la requête SQL à utiliser pour visualiser toutes les tables d'une base de données

```
SHOW TABLES;
```

Si vous voulez voir toutes les tables commençant par `prefix_`, vous pouvez utiliser cette commande

```
SHOW TABLES LIKE 'prefix\_%';
```

- _ est un wildcard représentant exactement un caractère, d'où le caractère d'échappement `\`.
- % est un wildcard pouvant représenter 0, 1 ou plusieurs caractères.

Pour afficher la structure d'une table, vous pouvez utiliser le mot clé `DESCRIBE`.
DESCRIBE <nom_de_la_table>;

Création de tables

Voici la syntaxe de création d'une table simple

```
CREATE TABLE sample_table (
    id INT,
    amount DOUBLE,
    value DECIMAL(10, 2),
    is_active BOOLEAN,
    birthdate DATE,
    initial CHAR(1),
    name VARCHAR(20),
    description TEXT,
    binary_data VARBINARY(255)
);
```

Modification de tables

La commande à utiliser pour modifier une table est `ALTER TABLE`.

Modification d'un type de colonne

```
ALTER TABLE premiere ALTER COLUMN mon_texte TYPE varchar(10);
```

Pour ajouter une contrainte non nullable (NOT NULL)

```
ALTER TABLE client ALTER COLUMN id SET NOT NULL;
```

Pour ajouter une valeur par défaut à une colonne

```
ALTER TABLE client ALTER COLUMN id SET DEFAULT 0;
```

Modification de tables

Pour ajouter une clé primaire à une table existante

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

Pour ajouter une clé étrangère

```
ALTER TABLE orders
ADD FOREIGN KEY (customer_id) REFERENCES customers (id);
```

Ajout d'une colonne

```
ALTER TABLE sample_table ADD COLUMN nouv_col REAL;
```

Suppression d'une colonne

```
ALTER TABLE sample_table DROP COLUMN nouv_col;
```

Renommer une colonne

```
ALTER TABLE clients RENAME COLUMN email TO contact_email;
```

Suppression de tables

Voici la syntaxe pour la suppression d'une table

```
DROP TABLE sample_table;
```

Nous pouvons spécifier une option `CASCADE`

```
DROP TABLE sample_table CASCADE;
```

L'option `CASCADE` permet de supprimer la table ainsi que tous les objets qui dépendent de cette table.

Vider le contenu d’une table

Pour vider le contenu de toutes une table sans supprimer la structure de la table, vous pouvez utiliser l’instruction `TRUNCATE`.

La commande TRUNCATE supprime rapidement toutes les lignes d'un ensemble de tables. Car contrairement à la commande DELETE qui supprime les données ligne par ligne, la commande TRUNCATE supprime toutes les pages associées à une table donnée.

Exemple

```
TRUNCATE TABLE employee;
```

Les principaux types de données

Voici les principaux types de données de MySQL

INT: un entier (encodé sur 4 octets). D'autres variantes sont possibles: TINYINT (1 octet), SMALLINT (2 octets) et BIGINT (8 octets).

DOUBLE: un nombre réel (encodé sur 8 octets). On peut également utiliser REAL (encodé sur 4 octets).

DECIMAL(p, s): un nombre décimal ayant p chiffres significatifs (precision) et s chiffres après la virgule (scale). Essentiellement utilisé pour la finance pour avoir un calcul exact sans approximations.

TINYINT(1) ou **BOOLEAN**: un booléen.
Si la valeur contenue est égale à 0, le booléen a la valeur faux, sinon il a la valeur vraie.

Les principaux types de données

DATE: Contient une date (année, mois, jour).

DATETIME: Contient la date et l'heure (encodé sur 8 octets). TIMESTAMP (encodé sur 4 octets) ne peut contenir que les dates entre 1970 et 2038.

TIME: Contient une heure (heure, minute, seconde).

CHAR(n): Chaîne de caractères contenant un nombre fixe de caractères (n caractères).

VARCHAR(n): Chaîne de caractères contenant un nombre variable de caractères (max n caractères).

TEXT: Chaîne de caractères de n'importe quelle taille.

BINARY(n), **VARBINARY(n)** ou **BLOB**: Données binaires.

Les colonnes auto incrémentées

En MySQL, il existe un moyen pour auto incrémenter le type INT automatiquement.

La syntaxe à ajouter est: AUTO_INCREMENT
La syntaxe précédente, indique que la colonne est auto-générée. Cette colonne également appelée séquence ou identité n'a pas besoin d'être spécifiée lors de l'insertion de nouvelles données. La colonne d'identité est une colonne NOT NULL à laquelle est attachée une séquence implicite.

Exemple d'un identifiant auto généré

```
CREATE TABLE ma_table (
  id INT AUTO_INCREMENT,
  CONSTRAINT pk_ma_table_id PRIMARY KEY (id));
```

Gestion des contraintes

Lors de la création des tables en base de données, vous pouvez spécifier les contraintes suivantes

1. Contrainte **PRIMARY KEY**. Spécifie les champs qui identifient la ligne.
2. Contrainte **FOREIGN KEY**. Spécifie les champs qui référencent une ligne d'une autre table.
3. Contrainte **NOT NULL**. La valeur ne peut pas être à NULL.
4. Contrainte **UNIQUE**. La valeur doit être unique dans la table.
5. Contrainte **CHECK**. La valeur doit respecter certaines conditions.

Clés primaires et clés étrangères

Une clé primaire (primary key), permet d'identifier une ligne de la table de façon unique.
Elle est introduite par le mot clé `PRIMARY KEY`.
Vous ne pouvez avoir qu'une seule clé primaire par table.
Les clés primaires assurent que la ou les colonnes sont uniques et non nullables.

La clé primaire peut-être déclarée lors de la création de la table

```
CREATE TABLE classe
  (classe_id INT AUTO_INCREMENT,
  niveau VARCHAR(12) NOT NULL,
  libelle VARCHAR(20) NOT NULL,
  CONSTRAINT pk_classe_id PRIMARY KEY (classe_id));
```

Ou elle peut être déclarée après la création de la table

```
ALTER TABLE classe ADD CONSTRAINT pk_classe_id PRIMARY KEY (classe_id);
```

Clés primaires et clés étrangères

Une clé étrangère (foreign key) est une contrainte qui garantit l'intégrité référentielle entre 2 tables.

Une clé étrangère identifie une ou plusieurs colonnes d'une table comme référençant une clé primaire d'une autre table.

```
CREATE TABLE eleve (
  eleve_id INT AUTO_INCREMENT,
  nom varchar(20) NOT NULL,
  prenom varchar(20) NOT NULL,
  classe_id int NOT NULL,
  CONSTRAINT pk_eleve_id PRIMARY KEY (eleve_id),
  CONSTRAINT fk_eleve_classe_id FOREIGN KEY (classe_id) REFERENCES classe(classe_id));
```

La clé étrangère peut être également déclarée après la création de la table

```
ALTER TABLE classe ADD CONSTRAINT fk_eleve_classe_id FOREIGN KEY (classe_id) REFERENCES classe(classe_id);
```

Contrainte NOT NULL

Par défaut les champs créés dans une table peuvent valoir NULL. Pour spécifier qu'un champ n'est pas nullable, vous devez utiliser le mot clé `NOT NULL`.

Vous pouvez spécifier qu'un champ est nullable à l'aide du mot clé `NULL`.

Exemple

```
CREATE TABLE Employees (
  employee_id INT AUTO_INCREMENT,
  first_name VARCHAR(50) NOT NULL,
  last_name VARCHAR(50) NOT NULL,
  hire_date DATE NULL,
  CONSTRAINT pk_employee_id PRIMARY KEY (employee_id));
```

Contrainte d'unicité

Vous pouvez spécifier qu'un champ ou plusieurs champs sont uniques pour une table avec le mot clé `UNIQUE`.

Exemple

```
CREATE TABLE students (
  student_id INT AUTO_INCREMENT,
  student_name VARCHAR(100) NOT NULL,
  email VARCHAR(100),
  CONSTRAINT pk_student_id PRIMARY KEY (student_id),
  CONSTRAINT uq_students_email UNIQUE(email));
```

Vous pouvez également spécifier qu'un ensemble de plusieurs champs soient uniques pour une table donnée.

```
ALTER TABLE students ADD CONSTRAINT uq_students_student_name_email UNIQUE(student_name, email);
```

Contraintes sur les valeurs

Vous pouvez pour chaque champ d'une table, spécifier une contrainte sur les valeurs possibles

```
CREATE TABLE product (
  product_id INT AUTO_INCREMENT,
  product_name VARCHAR(100) NOT NULL,
  price DECIMAL(10, 2) CHECK (price >= 0),
  stock_quantity INT CHECK (stock_quantity >= 0),
  CONSTRAINT pk_products_id PRIMARY KEY (product_id));
```

Vous pouvez également spécifier des contraintes impliquant plusieurs champs

```
ALTER TABLE eleve ADD CONSTRAINT ck_eleve_note_age CHECK ((note >= 0) AND (note <= 20) AND (age >=
10));
```

Introduction aux requêtes DML

Les requêtes DML, ou langage de manipulation de données, sont utilisées pour manipuler les données stockées dans une base de données.

Les requêtes DML permettent d'insérer de nouvelles données, de mettre à jour des données existantes et de supprimer des données.

Les requêtes DML modifient le contenu des tables et permettent aux utilisateurs d'interagir avec les données de la base.

Insertion des données dans une table

Lorsque la table possède des colonnes auto incrémentées `AUTO_INCREMENT` ou des colonnes attachées à des séquences, il n'est pas nécessaire de fournir des valeurs pour ces colonnes.

```
CREATE TABLE names (
    id INT AUTO_INCREMENT,
    nom VARCHAR(20),
    CONSTRAINT pk_names_id PRIMARY KEY (id)
);
```

Exemples d'insertions de valeurs.

```
INSERT INTO names VALUES (DEFAULT, 'James');
INSERT INTO names (nom) VALUES ('Jane');
INSERT INTO names (id, nom) VALUES (DEFAULT, 'John');
```

Insertion des données dans une table

Considérons la table suivante

```
CREATE TABLE dictionnaire (
    cle VARCHAR(20),
    valeur VARCHAR(20)
);
```

	cle character varying (20)	valeur character varying (20)
1	Bonjour	Hello
2	[null]	?
3	Pas de valeur	[null]

Voici les requêtes d'insertion

```
INSERT INTO dictionnaire VALUES ('Bonjour', 'Hello');
INSERT INTO dictionnaire VALUES (DEFAULT, '?');
INSERT INTO dictionnaire (cle) VALUES ('Pas de valeur');
```

Insertion des données dans une table

Nous pouvons également insérer plusieurs lignes à l'aide d'une commande `INSERT INTO`.

Exemple

```
INSERT INTO employees (name, age, salary, department_id)
   VALUES ('AB De-Villiers', 30, 5000.00, 1),
  ('Steve Smith', 28, 6000.00, 1),
  ('Michael Johnson', 35, 8000.00, 2),
  ('Ollie Robinson', 32, 5500.00, 2);
```

Insertion de plusieurs lignes à partir d'un SELECT sur une autre table

```
INSERT INTO Customers (CustomerName, City, Country)
  SELECT SupplierName, City, Country FROM Suppliers;
```

Mise à jour des données d'une table

Pour mettre à jour les données d'une table, vous pouvez la clause `UPDATE`.

Cette clause est de la forme suivante

```
UPDATE <nom de la table à mettre à jour>
SET <les nouvelles valeurs à attribuer>
WHERE <condition>;
```

Exemple

```
UPDATE matiere
SET libelle='Français'
WHERE id=1;
```

La clause WHERE est facultative, elle permet de restreindre les enregistrements à modifier.

Mise à jour des données d'une table

Nous pouvons mettre à jour les données d'une table en référençant les données d'une autre table.

Dans cet exemple toutes les notes de François Pignon sont fixées à 20.

```
UPDATE notes n
JOIN eleves e ON e.id = n.id_eleve
SET n.note = 20
WHERE e.nom = 'Pignon' AND e.prenom = 'François';
```

Supprimer des lignes d'une table

Pour supprimer des lignes d'une table, vous devez utiliser la clause DELETE FROM.

Cette clause est de la forme suivante

```
DELETE FROM nom_table
WHERE condition;
```

Exemple

```
DELETE FROM eleves
WHERE nom='Pignon' AND prenom='François';
```

Exemple de suppression avec référence à une autre table utilisant le mot clé JOIN.

```
DELETE n FROM notes n
JOIN eleves e ON e.id = n.id_eleve
WHERE e.nom = 'Pignon' AND e.prenom = 'François';
```

Introduction aux requêtes DQL

Une requête DQL (Data Query Language) est une requête SQL utilisée pour interroger et récupérer des données depuis une base de données. Elle ne modifie pas les données, mais elle permet de sélectionner, filtrer et trier les données selon les critères spécifiés.

La structure d'une commande DQL (Data Query Language) est la suivante

```
SELECT <liste des données à afficher>
FROM <liste des tables sur lesquelles effectuer des recherches>
WHERE <filtre sur les données à récupérer>
GROUP BY <liste des colonnes à grouper>
HAVING <condition sur les valeurs des fonctions d'agrégation retournées>
ORDER BY <liste des colonnes à ordonner>
LIMIT <nombre de lignes à afficher>
```

L’ordre des clauses est important. Seul la clause `SELECT` est obligatoire.

La clause SELECT

La clause SELECT spécifie les colonnes ou les valeurs que vous souhaitez récupérer dans le résultat de la requête.

La clause suivante retourne le résultat d'un calcul

```
-- retourne 7
SELECT 2*3 + 1;
```

La clause suivante retourne la version de MySQL

```
-- 8.0.35
SELECT VERSION();
```

La clause FROM

La clause FROM indique la table ou les tables à partir de laquelle vous souhaitez récupérer les données.

La clause suivante retourne tous les noms et prénoms de la table élève

```
SELECT nom, prenom
FROM eleve;
```

La structure est de la forme

```
SELECT colonne1, colonne2
FROM table1;
```

La clause FROM peut récupérer les données d'une ou de plusieurs tables.

La clause FROM

Si la clause FROM récupère les données de plusieurs table, la requête renverra par défaut le produit cartésien de toutes les tables

Exemple si la table `eleve` contient les prénoms ‘Gaston’ et ‘Harry’ et que la table `matiere` contient les libellés ‘Magie’, ‘Math’, ‘Histoire’ et ‘Français’.
La requête suivante retourne 8 lignes (2x4)

```
SELECT prenom, libelle FROM matiere, eleve;
```

```
 prenom | libelle
--------+----------
 Gaston | Magie
 Harry  | Magie
 Gaston | Math
 Harry  | Math
 Gaston | Histoire
 Harry  | Histoire
 Gaston | Français
 Harry  | Français
(8 rows)
```

La clause FROM

La clause SELECT peut retourner toutes les colonnes d'une table grâce au wildcard `*`.

Exemple

```
SELECT * FROM eleve;
```

```
id |    nom     | prenom | date_naissance
---+------------+--------+---------------
 1 | Pourquier  | Gaston | 1943-02-22
 2 | Potter     | Harry  | 1991-10-15
```

Vous pouvez renommer le nom d'une table

```
SELECT m.libelle FROM matiere AS m;
```

Le mot clé AS est facultatif.

La clause WHERE

La clause WHERE permet de spécifier les critères de filtrage pour les lignes à inclure dans le résultat. Les lignes qui ne satisfont pas la condition ne sont pas incluses.

Exemple

Dans la clause suivante, on ne récupère que les données ayant comme valeur 1 à la colonne `id`

```
SELECT * FROM eleve WHERE id=1;
```

La condition après le WHERE peut contenir plusieurs contraintes à l'aide des mots clé `AND`, `OR` et `NOT`.

```
SELECT * FROM eleve WHERE age > 18 AND (nom = 'Pourquier' OR prenom = 'Gaston');
```

La clause WHERE

Dans la condition après le WHERE on peut utiliser des opérateurs de comparaison.

Voici les différents opérateurs de comparaison que l'on peut utiliser dans une clause where

<	inférieur à
>	supérieur à
<=	inférieur ou égal à
>=	supérieur ou égal à
=	égal à
<>	différent de
!=	différent de

Vous pouvez également sélectionner une valeur entre 2 bornes à l'aide des mots clé BETWEEN et AND.
Exemple

```
WHERE note BETWEEN 5 AND 15
```

L’opérateur LIKE

L’opérateur LIKE est utilisé pour effectuer des comparaisons partielles de chaînes de caractères en utilisant des modificateurs de correspondance tels que `%` et `_`.

- % pour représenter zéro, un, ou plusieurs caractères.
- _ pour représenter un seul caractère.

Exemple

```
SELECT nom, prenom
FROM employes
WHERE nom LIKE 'Sm%';
```

Les opérateurs IN, NOT IN

L’opérateur IN est utilisé pour filtrer les résultats en fonction d'une liste de valeurs spécifiées. Si la liste spécifiée contient des doublons, les valeurs en double seront ignorées.

L’opérateur NOT IN est utilisé pour exclure les résultats qui correspondent à une liste spécifiée de valeurs.

Exemple

```
SELECT nom, prenom
FROM employes
WHERE departement IN ('Ventes', 'Marketing');
```

Cela renverra les employés dont le département est soit 'Ventes' soit 'Marketing'.

Les opérateurs IS NULL, IS NOT NULL

Les opérateurs IS NULL et IS NOT NULL sont utilisés en SQL pour vérifier si une colonne contient des valeurs nulles ou non nulles. Une valeur nulle, signifie l'absence de valeurs dans une colonne.

Ces opérateurs peuvent être utilisés dans des conditions de la clause WHERE pour restreindre les résultats en fonction de la présence ou de l'absence de valeurs nulles.

Exemple

```
SELECT nom, prenom
FROM employes
WHERE date_de_naissance IS NULL;
```

Les opérateurs IS NULL, IS NOT NULL

La comparaison des valeurs nulles en SQL doit être effectuée à l'aide de l'opérateur IS NULL et non pas `= NULL`.

Dans MySQL une comparaison à l'aide de l'opérateur `=` et une opérande NULL retournera toujours faux.

L'exemple suivant retourne tous les employés n'ayant pas renseigner leurs date de naissance (date_naissance est nulle).

```
SELECT nom, prenom FROM employe WHERE date_naissance IS NULL;
```

Une clause WHERE avec `date_naissance = NULL` ne retourne aucune valeur.

Les fonctions d’agrégations

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 d’une colonne sur un ensemble d’enregistrements.

Les fonctions d’agrégations

Pour retourner le nombre de colonnes d’une table, vous pouvez utiliser la requête suivante

```
SELECT count(*) AS nb_colonnes FROM notes;
```

Pour retourner le nombre de valeurs non nulles d’un champ spécifique, vous pouvez utiliser la commande suivante

```
SELECT count(nom_du_champ) AS nb FROM nom_table;
```

Exemple

```
SELECT count(note) AS nb_colonnes FROM notes;
```

Vous pouvez également utiliser les autre fonctions d'agrégation

```
SELECT min(note) AS min_note, max(note) AS max_note, avg(note) AS avg_note FROM notes;
```

La clause GROUP BY

Attention si dans une clause SELECT vous mélanger des fonctions d'agrégation (COUNT, SUM, MIN, MAX, AVG) avec d'autres champs, vous devez utiliser ces autres champs dans une clause GROUP BY. Autrement vous aurez un message d'erreur.

Le mot-clé GROUP BY est utilisé dans SQL pour regrouper les résultats de la requête en fonction des valeurs d'une ou plusieurs colonnes. Lorsqu'une clause GROUP BY est utilisée, les résultats sont regroupés en enregistrements distincts basés sur les valeurs des colonnes spécifiées.

Exemple

```
SELECT count(*), nom
FROM eleve
GROUP BY nom;
```

La clause GROUP BY

La clause GROUP BY est utilisée pour regrouper les lignes en fonction des valeurs de certaines colonnes. Cette clause est souvent utilisée avec des fonctions d'agrégation telles que COUNT, SUM, MIN, MAX, AVG, …

Exemple

```
SELECT colonne1, colonne2, FONCTION_AGGREGATION(colonne3)
FROM nom_de_la_table
GROUP BY colonne1, colonne2;
```

La requête précédente regroupe les valeurs de la fonction d'agrégation par `colonne1` et `colonne2`.

La clause HAVING

La clause HAVING est utilisée en conjonction avec GROUP BY pour filtrer les résultats agrégés basés sur une condition spécifiée.

Alors que la clause WHERE filtre les lignes avant le regroupement, la clause HAVING filtre les résultats après le regroupement.

Exemple

```
SELECT departement, AVG(salaire) as salaire_moyen
FROM employes
GROUP BY departement
HAVING AVG(salaire) > 50000;
```

Dans cet exemple, la clause HAVING filtre les résultats pour inclure uniquement les départements ayant un salaire moyen supérieur à 50 000.

La clause ORDER BY

La commande ORDER BY en SQL est utilisée pour trier les résultats d'une requête en fonction des valeurs d'une ou plusieurs colonnes, dans un ordre ascendant (par défaut) ou descendant.

Vous pouvez spécifier l'ordre de tri à l'aide des mots clés `ASC` et `DESC`.

- ASC (par défaut) : Tri ascendant (du plus petit au plus grand).
- DESC : Tri descendant (du plus grand au plus petit).

Exemple

```
SELECT nom, prenom, salaire
FROM employes
ORDER BY salaire DESC;
```

La clause ORDER BY

Il est également possible de trier par plusieurs colonnes en spécifiant l'ordre pour chaque colonne.

Par exemple, trier par salaire décroissant, puis par nom de manière ascendante.

```
SELECT nom, prenom, salaire
FROM employes
ORDER BY salaire DESC, nom ASC;
```

Cela affichera les employés triés par salaire décroissant. En cas d'égalité au niveau du salaire, le tri sera effectué par nom de manière ascendante.

La clause LIMIT

La clause LIMIT permet de limiter le nombre de lignes renvoyées dans le résultat. C'est utile pour paginer les résultats ou pour limiter la quantité de données renvoyées.

Exemple

```
SELECT * FROM eleves LIMIT 100;
```

Pour récupérer les 100 premières lignes d'une table.

La clause LIMIT peut être associée à un OFFSET pour sauter un nombre de lignes qui nous intéressent pas.
LIMIT nombre_de_lignes_a_renvoyer OFFSET nombre_de_lignes_a_ignorer

Exemple

```
SELECT * FROM eleves LIMIT 100 OFFSET 50;
```

La commande précédente renvoie les lignes 51 à 150.

Les jointures

Vous avez vu précédemment qu'une clause FROM récupérant les données de plusieurs tables, renvoyait le produit cartésien de ces 2 tables.

Exemple

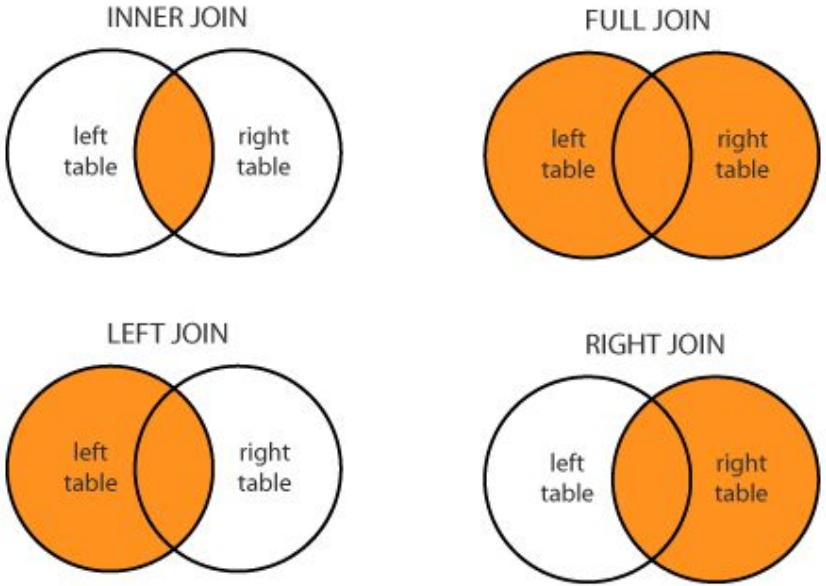
```
SELECT prenom, libelle FROM matiere, eleve;
```

Généralement lorsque nous récupérons des données de plusieurs tables, nous avons des relations entre les données des tables récupérées.
Ces relations sont généralement spécifiées par des clés primaires `PRIMARY KEY` et des clés étrangères `FOREIGN KEY`.

Les colonnes PRIMARY KEY et FOREIGN KEY vont nous permettre de joindre les données de 2 ou plusieurs tables pour ne récupérer que les données qui nous intéressent.

Les jointures

Voici les principaux types de jointures utilisés



La jointure INNER JOIN

La commande INNER JOIN (jointure interne) renvoie uniquement les lignes pour lesquelles il existe une correspondance dans les deux tables. Cela signifie que seules les lignes avec des valeurs correspondantes dans les colonnes spécifiées seront incluses dans le résultat.

Prenoms l'exemple suivant dans lequel nous possédons 2 tables

eleves (id, prenom, nom)

notes (id, note, matiere, #id_eleve)

La clé étrangères notes.id_eleve référence la colonne `id` de la table eleves.

ALTER TABLE notes ADD FOREIGN KEY (id_eleve) REFERENCES eleves(id);

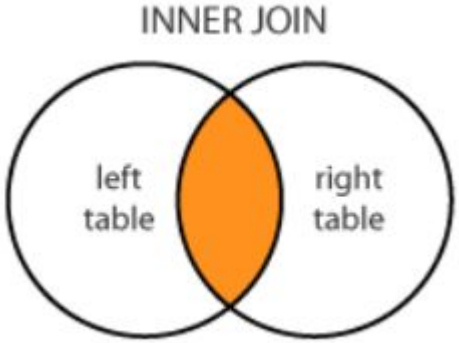
Pour récupérer les notes de l'élève Gaston Pourquier nous devons joindre les tables `eleves` et `notes` et sélectionner l'élève qui nous intéresse afin de ne pas récupérer les notes des autres élèves.

La jointure INNER JOIN

Pour ne récupérer que les notes de l'élève "Gaston Pourquier".

Nous pouvons utiliser la requête suivante

```
SELECT n.matiere, n.note
FROM eleves e
INNER JOIN notes n ON n.id_eleve = e.id
WHERE e.prenom='Gaston' AND e.nom='Pourquier';
```



La jointure INNER JOIN

Autre exemple avec les tables Utilisateurs et Commandes.

Utilisateurs (id, nom, prenom)
Commandes (id, ref_commande, #utilisateur_id)

Où la colonne `utilisateur_id` de la table `Commandes` référence la colonne `id` de la table utilisateur.

Pour avoir la référence des commandes pour chaque utilisateur, nous pouvons exécuter la commande suivante

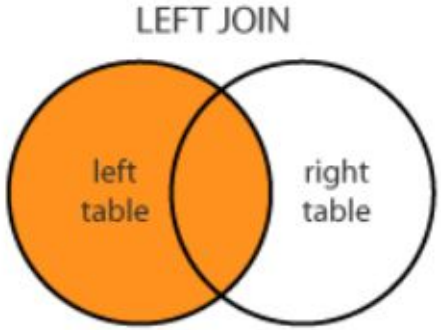
```
SELECT u.nom, u.prenom, c.ref_commande
FROM Utilisateurs u
INNER JOIN Commandes c ON u.id = c.utilisateur_id;
```

La jointure LEFT JOIN

La jointure LEFT JOIN (Jointure à Gauche) renvoie toutes les lignes de la table de gauche (première table spécifiée) et les lignes correspondantes de la table de droite (deuxième table spécifiée). Si aucune correspondance n'est trouvée dans la table de droite, des valeurs NULL sont renvoyées pour les colonnes de la table de droite.

Exemple: nous voulons obtenir la liste de tous les utilisateurs et, le cas échéant, leurs commandes associées.

```
SELECT u.nom, u.prenom, c.ref_commande
FROM Utilisateurs u
LEFT JOIN Commandes c ON u.id = c.utilisateur_id;
```

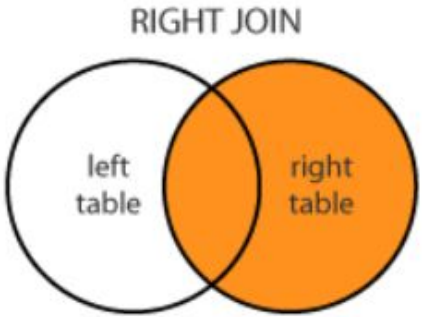


La jointure RIGHT JOIN

RIGHT JOIN (Jointure à Droite) est similaire à une jointure à gauche, mais elle renvoie toutes les lignes de la table de droite et les lignes correspondantes de la table de gauche. Si aucune correspondance n'est trouvée dans la table de gauche, des valeurs NULL sont renvoyées pour les colonnes de la table de gauche.

Exemple: nous voulons obtenir la liste de toutes les commandes et, le cas échéant, les noms des utilisateurs correspondants.

```
SELECT u.nom, u.prenom, c.ref_commande
FROM Utilisateurs u
RIGHT JOIN Commandes c ON u.id = c.utilisateur_id;
```

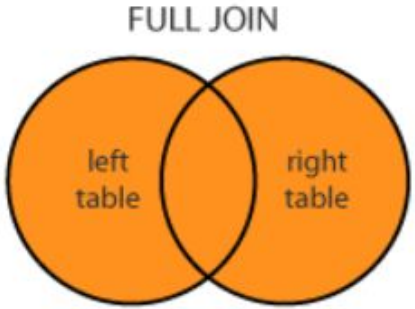


La jointure FULL JOIN

La jointure FULL JOIN ou FULL OUTER JOIN (Jointure complète) renvoie toutes les lignes des deux tables, en combinant les lignes correspondantes et en renvoyant des valeurs NULL lorsque des correspondances ne sont pas trouvées.

Exemple: nous voulons obtenir la liste de tous les utilisateurs et de toutes les commandes, en montrant les correspondances entre les deux.

```
SELECT u.nom, u.prenom, c.ref_commande
FROM Utilisateurs u
FULL JOIN Commandes c ON u.id = c.utilisateur_id;
```



Les sous-requêtes

Les sous-requêtes sont des requêtes SQL imbriquées à l'intérieur d'une requête principale. Elles sont utilisées pour extraire des données plus complexes et pour effectuer des opérations basées sur le résultat de ces sous-requêtes.

Elles peuvent être utilisées dans la clause SELECT, FROM ou WHERE.

Les sous-requêtes peuvent être

- non corrélées
- corrélées

Les sous-requêtes

Sous-requêtes dans la clause SELECT

Utilisées pour récupérer une valeur unique à partir d'une sous-requête.

```
SELECT column1, (SELECT MAX(column2) FROM table2) AS max_value
FROM table1;
```

Sous-Requêtes dans la Clause FROM

Pour créer une table temporaire à partir du résultat de la sous-requête.

```
SELECT cl.nom, cl.niveau, mc.moyenne
FROM (SELECT nom_classe, AVG(note) AS moyenne FROM eleve GROUP BY nom_classe) AS mc
INNER JOIN classe cl ON cl.nom = mc.nom_classe;
```

Les sous-requêtes

Sous-Requêtes dans la Clause WHERE

Filtrage basé sur le résultat de la sous-requête.

```
SELECT CustomerID
FROM Customer
WHERE TerritoryID =
    (SELECT TerritoryID
     FROM SalesPerson
     WHERE BusinessEntityID = 276);
```

Les sous-requêtes

On peut distinguer deux types de sous-requêtes : les sous-requêtes corrélées et les sous-requêtes non corrélées.

Les sous-requêtes non corrélées

Les sous-requêtes non corrélées sont évaluées une seule fois, indépendamment des lignes de la requête principale.

Le résultat de la sous-requête est calculé une fois, et ce résultat est ensuite utilisé dans la requête principale.

```
SELECT column1
FROM table1
WHERE column2 = (SELECT MAX(column2) FROM table1);
```

Les sous-requêtes

En utilisant le mot clé `WITH`, les sous-requêtes non corrélées peuvent également être sorties de la requête principale pour une meilleurs lisibilité.

Exemple

```
WITH mc AS (SELECT nom_classe, AVG(note) AS moyenne FROM eleve GROUP BY nom_classe)
SELECT cl.nom, cl.niveau, mc.moyenne
FROM mc
INNER JOIN classe cl ON cl.nom = mc.nom_classe;
```

Les sous-requêtes

Sous-Requêtes Corrélées

Dans ce cas la sous-requête fait référence aux colonnes de la requête principale.

Les sous-requêtes corrélées sont évaluées pour chaque ligne de la requête principale.

```
SELECT column1
FROM table1 t1
WHERE column2 = (SELECT MAX(column2) FROM table1 t2 WHERE t2.category = t1.category);
```

Les opérateurs EXISTS et NOT EXISTS

Les opérateurs `EXISTS` et `NOT EXISTS` sont utilisés pour vérifier l'existence ou l'absence de résultats dans une sous-requête.

Ces opérateurs sont souvent utilisés avec la clause WHERE pour filtrer les résultats en fonction de la présence ou de l'absence de résultats dans une sous-requête.

Si la sous-requête `EXISTS` renvoie au moins un résultat, la condition est évaluée comme vraie.

Exemple

```
SELECT nom, prenom
FROM employes
WHERE EXISTS (SELECT 1 FROM ventes WHERE ventes.id_employe = employes.id);
```

Cela retourne les employés pour lesquels il existe au moins une entrée dans la table des ventes correspondant à l'employé.

Les opérateurs EXISTS et NOT EXISTS

L’opérateur NOT EXISTS est utilisé pour vérifier si une sous-requête ne renvoie aucun résultat.

Si la sous-requête ne renvoie aucun résultat, la condition NOT EXISTS est évaluée comme vraie.

Exemple

```
SELECT nom, prenom
FROM employes
WHERE NOT EXISTS (SELECT 1 FROM congés WHERE congés.id_employe = employes.id);
```

Retourne les employés pour lesquels il n'existe aucune entrée dans la table des congés correspondant à l'employé.

La clause UNION

La clause UNION est utilisé pour combiner les résultats de deux ou plusieurs requêtes, en éliminant les doublons.

Cela signifie que si un enregistrement apparaît dans les résultats des deux requêtes, il ne sera inclus qu'une seule fois dans le résultat final.

Exemple

```
SELECT nom, prenom
FROM employes
WHERE departement = 'Ventes'
UNION
SELECT nom, prenom
FROM employes
WHERE departement = 'Marketing';
```

La clause UNION ALL

La clause UNION ALL, combine les résultats de deux requêtes ou plusieurs requêtes sans éliminer les doublons.

Cela signifie que si un enregistrement apparaît dans les résultats des deux requêtes, il sera inclus dans le résultat final autant de fois qu'il apparaît dans chaque requête.

Exemple

```
SELECT nom, prenom FROM employes WHERE departement = 'Ventes'
UNION ALL
SELECT nom, prenom FROM employes WHERE departement = 'Marketing';
```

Dans cet exemple, les résultats des deux requêtes seront combinés, et les doublons ne seront pas éliminés. Si un employé appartient aux deux départements, il apparaîtra dans le résultat final autant de fois qu'il apparaît dans chaque requête.