Requêtes SQL simples

I. Requête SELECT

```
-- Créer une nouvelle table "users"

CREATE TABLE users (

    id INT AUTO_INCREMENT PRIMARY KEY,

    username VARCHAR(50) UNIQUE NOT NULL,

    email VARCHAR(100) UNIQUE NOT NULL,

    age INT,

    password VARCHAR(255) NOT NULL

) ENGINE=InnoDB;

-- Insérer des données de test

INSERT INTO users (username, email, age, password) VALUES

('john', 'john@example.com', 25,
'$2y$10$QivErFPIjzQ4qu.4.Rok/uD5Fnlr.Mo3n0oXlVd5yZ6tRfmbgFk2W'), -- Mot de passe hashé :
password123

('emma', 'emma@example.com', 30,
'$2y$10$QivErFPIjzQ4qu.4.Rok/uD5Fnlr.Mo3n0oXlVd5yZ6tRfmbgFk2W'),

('sam', 'sam@example.com', 28,
'$2y$10$QivErFPIjzQ4qu.4.Rok/uD5Fnlr.Mo3n0oXlVd5yZ6tRfmbgFk2W');

-- Création de la table "orders"

CREATE TABLE orders (

    order_id INT AUTO_INCREMENT PRIMARY KEY,

    user_id INT,

    order_details VARCHAR(255),

    CONSTRAINT fk_user_id
```

FOREIGN KEY (user_id)

REFERENCES users(id)

ON DELETE CASCADE

) ENGINE=InnoDB;

-- Insertion d'exemples de données dans la table "orders"

INSERT INTO orders (user_id, order_details) VALUES

(1, 'Commande 1 - Détails'),

(2, 'Commande 2 - Détails'),

(3, 'Commande 3 – Détails');

Ce fichier SQL crée une table "users" avec les colonnes suivantes :

- id : Clé primaire auto-incrémentée
- username : Nom d'utilisateur unique, non nul
- email : Adresse e-mail unique, non nulle
- age : Âge de l'utilisateur (peut être nul)
- password : Mot de passe de l'utilisateur (stocké de manière sécurisée, non nul)

Il insère également trois enregistrements de test dans la table "users".

- **Sélectionner sans projection :**

```
SELECT * FROM users;
```

Cette requête sélectionnera toutes les colonnes de la table "users" pour tous les enregistrements. Cela inclura toutes les informations disponibles pour chaque utilisateur, telles que l'identifiant (id), le nom d'utilisateur (username), l'adresse e-mail (email), l'âge (age), et le mot de passe (password). Utiliser cette approche peut être utile lorsque vous avez besoin de récupérer toutes les données de la table sans filtres ou restrictions supplémentaires.

- **Sélectionner avec projection (uniquement les colonnes nécessaires) :**

```
SELECT username, email FROM users;
```

Supposons que nous voulions sélectionner uniquement les noms d'utilisateur (username) et les adresses e-mail (email) de tous les utilisateurs Cette requête sélectionnera uniquement les colonnes spécifiées dans la liste (username et email) pour tous les enregistrements de la table "users". Cela permet de limiter les données récupérées à celles qui sont pertinentes pour l'utilisation spécifique de la requête.

- **Avec des critères dans le WHERE (restriction) :**

```
SELECT * FROM users WHERE username = 'john';
```

- **Contraintes d'intégrité IN, BETWEEN, LIKE et IS NULL / IS NOT NULL :**

```
SELECT * FROM users WHERE age IN (20, 25, 30);
SELECT * FROM users WHERE age BETWEEN 20 AND 30;
SELECT * FROM users WHERE email LIKE '%@example.com';
SELECT * FROM users WHERE age IS NULL;
SELECT * FROM users WHERE email IS NOT NULL;
```

- **Jointures simples :**

Une jointure INNER JOIN récupère uniquement les lignes des deux tables qui ont une correspondance.

```
SELECT users.username, orders.order_details
FROM users
INNER JOIN orders ON users.id = orders.user_id;
```

Dans cette requête, nous utilisons INNER JOIN pour combiner les lignes de la table "orders" avec les lignes de la table "users" où les valeurs de la colonne "user_id" dans la table "orders" correspondent aux valeurs de la colonne "id" dans la table "users". Cela nous permet de récupérer les détails de chaque commande avec le nom d'utilisateur correspondant.

- **Jointure et contrainte en même temps :**

```
SELECT users.username, orders.order_details
FROM users
INNER JOIN orders ON users.id = orders.user_id;
WHERE users.id = 1;
```

Dans cette requête, l'INNER JOIN récupère uniquement les lignes des deux tables qui ont une correspondance. L'entrée pour l'utilisateur "claire" dans la table "users" est exclue car il n'y a pas de correspondance dans la table "orders", de même que la commande 4 qui n'a pas d'utilisateur correspond dans la table "users"».

- **Jointures LEFT JOIN :**

Une jointure LEFT JOIN récupère toutes les lignes de la table de gauche, ainsi que les lignes correspondantes de la table de droite s'il y en a. Si aucune correspondance n'est trouvée dans la table de droite, NULL est retourné.

```
SELECT users.username, orders.order_details
FROM users
LEFT JOIN orders ON users.id = orders.user_id;
```

Le LEFT JOIN récupère toutes les lignes de la table "users", ainsi que les lignes correspondantes de

la table "orders". Comme il n'y a pas de commande pour l'utilisateur "Claire" (user_id = 4) dans la table "orders", la colonne "order_details" pour cet utilisateur est NULL.

- **Jointures CROSS JOIN :**

La jointure CROSS JOIN retourne le produit cartésien des deux tables.

```
SELECT users.username, orders.order_details
FROM users
CROSS JOIN orders;
```

La jointure CROSS JOIN retourne le produit cartésien des deux tables, ce qui signifie qu'elle combine chaque ligne de la table "users" avec chaque ligne de la table "orders", créant ainsi toutes les combinaisons possibles entre les deux tables. On obtient donc un résultat où chaque nom d'utilisateur est associé à chaque détail de commande.

II. Requête INSERT

- **Insertion de donnée (avec clé primaire en auto-increment) :**

```
INSERT INTO users (username, email) VALUES ('John', 'john@example.com');
```

Supposons que nous avons une table "users" avec une clé primaire "id" en auto-incrémentation et des colonnes "username" et "email". Cette requête insère une nouvelle ligne dans la table "users" avec le nom d'utilisateur "John" et l'e-mail "john@example.com". La valeur de la colonne "id" sera automatiquement générée et incrémentée par le système.

- **Insertion de données ou clefs en doublons:**

```
INSERT INTO users (username, email) VALUES ('John', 'john@example.com');
INSERT INTO users (username, email) VALUES ('Jane', 'john@example.com');
```

Supposons que nous voulions insérer un utilisateur avec le même e-mail qu'un utilisateur existant. Comme l'e-mail est en double (déjà présent dans la table), cela provoquera une erreur de violation de la contrainte d'unicité.

Supposons que nous voulions insérer un utilisateur avec une clé primaire qui est déjà utilisée. Comme l'identifiant "1" est déjà utilisé par un autre utilisateur dans la table, cela provoquera une erreur de violation de la contrainte de clé primaire en doublon.

III. Requête UPDATE

- **Avec des critères très restrictifs (clé primaire = identifiant):**

```
UPDATE users SET username = 'jane' WHERE id = 1;
```

Supposons que nous voulons mettre à jour le nom d'utilisateur de l'utilisateur ayant l'identifiant "1". Cette requête mettra à jour le nom d'utilisateur de l'utilisateur dont l'identifiant est "1" avec la valeur "Jane".

- **Avec des critères plus larges:**
  UPDATE users **SET** email = 'new_email@example.com' **WHERE** age > 30;

Supposons que nous voulons mettre à jour l'adresse e-mail de tous les utilisateurs dont l'âge est supérieur à 30 ans. Cette requête mettra à jour l'adresse e-mail de tous les utilisateurs dont l'âge est supérieur à 30 ans avec la nouvelle adresse e-mail "new_email@example.com".

- **Avec référence aux autres colonnes:**
  UPDATE users **SET** username = **SUBSTRING_INDEX**(email, '@', 1);

Supposons que nous voulons mettre à jour le nom d'utilisateur pour qu'il soit égal à l'adresse e-mail de chaque utilisateur. Cette requête mettra à jour le nom d'utilisateur de chaque utilisateur en extrayant le nom d'utilisateur à partir de leur adresse e-mail (en utilisant la fonction SUBSTRING_INDEX pour obtenir la partie de l'adresse e-mail avant le caractère "@").

IV. Requête DELETE

- **Avec des critères très restrictifs (clé primaire = identifiant):**
  DELETE FROM users **WHERE** id = 1;

Supposons que nous voulons supprimer l'utilisateur ayant l'identifiant "1". Cette requête supprimera l'utilisateur dont l'identifiant est "1" de la table "users".

- **Avec des critères plus larges:**
  DELETE FROM users **WHERE** age > 30;

Supposons que nous voulons supprimer tous les utilisateurs dont l'âge est supérieur à 30 ans. Cette requête supprimera tous les utilisateurs dont l'âge est supérieur à 30 ans de la table "users".

- **Avec du delete cascade:**

Supposons que nous avons une contrainte de clé étrangère avec l'option "ON DELETE CASCADE" configurée, ce qui signifie que lorsque l'enregistrement parent est supprimé, les enregistrements enfants correspondants dans une table liée seront également supprimés.

Si la table "orders" a une clé étrangère "user_id" qui fait référence à la table "users" avec l'option "ON DELETE CASCADE" configurée, cette requête supprimera également toutes les commandes associées à l'utilisateur dont l'identifiant est "1" dans la table "orders".

***ALTER TABLE** orders*
***ADD CONSTRAINT** fk_user_id*
***FOREIGN KEY** (user_id)*
***REFERENCES** users(id)*
***ON DELETE CASCADE**;*

- **Avec une contrainte d'intégrité:**

Supposons que nous voulons supprimer un utilisateur mais que la contrainte d'intégrité

référentielle empêche la suppression parce qu'il existe des commandes associées à cet utilisateur dans la table "orders". Si la contrainte d'intégrité référentielle est configurée pour empêcher la suppression d'un utilisateur tant qu'il existe des commandes associées dans la table "orders", cette requête générera une erreur de violation de la contrainte d'intégrité.

```
ALTER TABLE orders
ADD CONSTRAINT fk_user_id
FOREIGN KEY (user_id)
REFERENCES users(id)
ON DELETE RESTRICT;
```

V. Organiser les données

- **ORDER BY:**

```
SELECT * FROM users
ORDER BY age ASC;
```

Les utilisateurs sont triés par ordre croissant d'âge. Utiliser ORDER BY permet d'organiser les résultats de la requête selon une ou plusieurs colonnes spécifiées, dans l'ordre croissant (ASC) par défaut ou dans l'ordre décroissant (DESC) si spécifié.

- **GROUP BY:**

```
SELECT user_id, COUNT(*) AS total_orders
FROM orders
GROUP BY user_id;
```

La clause GROUP BY est utilisée pour regrouper les résultats en fonction des valeurs d'une colonne spécifiée. Par exemple, pour compter le nombre de commandes par utilisateur. Cela retournera le nombre total de commandes pour chaque utilisateur.

- **HAVING:**

```
SELECT user_id, COUNT(*) AS total_orders
FROM orders
GROUP BY user_id
HAVING COUNT(*) >= 3
```

La clause HAVING est utilisée pour filtrer les résultats d'une requête GROUP BY en fonction d'une condition spécifiée. Par exemple, pour afficher uniquement les utilisateurs ayant passé au moins 3 commandes. Cela retournera les utilisateurs qui ont passé au moins 3 commandes.