Présentation du SQL sous MySQL/MariaDB

1. Histoire du SQL

SQL (Structured Query Language) est un langage standardisé de gestion de base de données qui a été développé pour la première fois dans les années 1970 par IBM. Il a été conçu pour manipuler et récupérer des données stockées dans les systèmes de gestion de bases de données relationnelles. Le premier prototype, appelé SEQUEL (Structured English Query Language), a été développé pour le projet de système de gestion de bases de données relationnelles appelé System R chez IBM. Au fil du temps, le langage a été normalisé par l'ANSI et l'ISO, ce qui a conduit à sa large adoption dans l'industrie.

2. Liste de quelques SGBD en SQL

Les systèmes de gestion de bases de données (SGBD) en SQL sont nombreux et variés, chacun ayant des caractéristiques et des avantages spécifiques. Voici quelques-uns des principaux SGBD basés sur SQL :

1. **Oracle Database** : C'est l'un des systèmes de gestion de bases de données relationnelles les plus robustes et les plus utilisés, idéal pour les grandes entreprises avec de grandes quantités de données et des transactions complexes.

2. **Microsoft SQL Server** : Ce SGBD est très populaire dans les environnements Windows et offre une intégration facile avec d'autres produits Microsoft. Il est connu pour sa facilité d'utilisation et ses bonnes performances.

3. **MySQL** : Propriété d'Oracle Corporation, MySQL est très populaire dans le monde du web pour sa facilité d'utilisation et sa gratuité. Il est souvent utilisé pour les applications web et les sites Internet.

4. **PostgreSQL** : Connu pour sa conformité aux standards et son extensibilité, PostgreSQL est un système de gestion de base de données relationnelle objet très respecté et puissant.

5. **IBM Db2** : Db2 offre des capacités avancées de gestion de données et d'analyse pour les entreprises, avec des options disponibles pour le cloud ainsi que pour les déploiements sur site.

6. **SQLite** : Très léger, SQLite est une solution de base de données qui est souvent utilisée pour les applications mobiles et les petites applications de bureau. Il est unique en ce qu'il

est intégré directement dans une application.

7. **MariaDB** : Créé par les développeurs originaux de MySQL, MariaDB est souvent considéré comme son remplacement direct, offrant plus de fonctionnalités, une meilleure performance et une plus grande ouverture.

8. **SAP HANA** : C'est une plateforme in-memory qui permet le traitement de grandes quantités de données en temps réel. SAP HANA est particulièrement bien adaptée pour les environnements d'affaires qui utilisent d'autres logiciels SAP.

Chacun de ces SGBD a ses propres caractéristiques qui peuvent être mieux adaptées à certaines applications ou environnements que d'autres.

3. MySQL et MariaDB

MySQL est l'un des systèmes de gestion de bases de données relationnelles open source les plus populaires. Il a été créé par Michael Widenius ("Monty") et David Axmark dans les années 1990. MySQL est célèbre pour sa rapidité, sa robustesse et sa facilité d'utilisation. Il est largement utilisé pour les applications web et est devenu une partie intégrante de la pile technologique LAMP (Linux, Apache, MySQL, PHP/Python/Perl).

MariaDB est open source et a été forké de MySQL en 2009 par les mêmes développeurs qui ont créé MySQL, après que MySQL a été acquis par Oracle Corporation. MariaDB est conçu pour être compatible avec MySQL, tout en offrant de nouvelles fonctionnalités, des performances améliorées et des fonctionnalités remplacées qui n'étaient pas disponibles dans les versions gratuites de MySQL.

4. Fonctionnement Général

SQL fonctionne en analysant et en exécutant des instructions écrites dans le langage SQL. Ces instructions peuvent effectuer diverses tâches, telles que la création de tables, la mise à jour de données, la récupération de données, et la gestion des transactions. SQL utilise des déclarations pour définir des structures de données, des requêtes pour récupérer des données et des commandes pour gérer les données et les transactions.

5. Moteurs de Table MySQL

MySQL utilise ce qu'on appelle des "moteurs de stockage" pour gérer la manière dont les données sont stockées, gérées et récupérées. Chaque moteur de stockage a ses propres

propriétés, optimisations et utilisations spécifiques. Voici quelques-uns des moteurs les plus courants :

- **MyISAM** : C'était le moteur par défaut avant MySQL 5.5. Il est connu pour sa rapidité dans les opérations de lecture, mais il ne supporte pas les transactions ni la récupération après un crash.
- **InnoDB** : C'est le moteur de stockage par défaut pour MySQL depuis la version 5.5. InnoDB supporte les transactions, la récupération après un crash et le verrouillage au niveau des lignes. Il est conçu pour maximiser la robustesse et l'intégrité des données.
- **Memory** : Ce moteur stocke les données en mémoire, ce qui offre un accès très rapide. Cependant, les données sont perdues lorsque la base de données est arrêtée, car elles ne sont pas stockées de manière persistante.
- **Archive** : Utilisé pour le stockage d'un grand nombre de données qui ne nécessitent pas de modification fréquente. Il est optimisé pour les insertions rapides et la compression des données.

Les moteurs de table de MySQL permettent aux utilisateurs de choisir la meilleure façon de stocker et de gérer leurs données en fonction de leurs besoins spécifiques, ce qui rend ce système extrêmement flexible et puissant pour une large gamme d'applications.

6. Documentation SQL : Guillemets et Syntaxe

6.1. Introduction

En SQL, l'usage des guillemets simples ('), doubles ("), et des accents graves (`) varie selon le système de gestion de base de données (SGBD) utilisé. Ces caractères sont utilisés pour identifier les éléments de la base de données, comme les noms de tables et de colonnes, et pour délimiter les chaînes de caractères.

6.2. Guillemets simples (')

- **Usage :** Utilisés pour délimiter les chaînes de caractères.
- **Exemple :**

```sql
SELECT * FROM utilisateurs WHERE nom = 'Dupont';
```

6.3. Guillemets doubles ("")

- **Usage :** Selon le standard SQL, les guillemets doubles sont utilisés pour identifier les identifiants (noms de table, de colonne, etc.) qui ne respectent pas les conventions normales de nommage ou qui contiennent des caractères spéciaux.
- **Exemple :**

```
SELECT "nomColonne" FROM "maTable" WHERE "nomColonne" = 'valeur';
```

- **Remarque :** Tous les SGBD ne suivent pas cette convention. Par exemple, MySQL utilise les guillemets doubles comme des guillemets simples pour délimiter les chaînes, à moins que le mode ANSI_QUOTES soit activé.

6.4. Accents graves (`)

- **Usage :** Utilisés spécifiquement dans MySQL et quelques autres SGBD pour entourer les identifiants.
- **Exemple :**

```
SELECT `nomColonne` FROM `maTable` WHERE `nomColonne` = 'valeur';
```

- **Remarque :** Cela permet de s'assurer que les mots réservés peuvent être utilisés comme noms de table ou de colonne sans provoquer d'erreurs.

6.5. Règles de syntaxe générale

1. **Sensibilité à la casse :** SQL n'est pas sensible à la casse pour les mots-clés, mais le traitement des identifiants peut l'être selon le SGBD et la configuration du système d'exploitation sous-jacent.
2. **Commentaires :**
   - Commentaires sur une ligne : Utilisez `-- commentaire` ou `# commentaire` (spécifique à MySQL).
   - Commentaires sur plusieurs lignes : Utilisez `/* commentaire */`.
3. **Terminaison des instructions :** Les instructions SQL doivent se terminer par un point-virgule (`;`), bien que certains outils et environnements puissent tolérer des instructions sans point-virgule en fin de requête.
4. **Nommage des identifiants :**

- Ne doivent pas commencer par un chiffre, mais peuvent être numériques si délimités correctement.
- Ne doivent pas contenir d'espaces. Utilisez des underscores `_` comme séparateurs.
- Évitez d'utiliser des mots réservés du SQL, à moins qu'ils ne soient entourés par des guillemets appropriés.

7. SQL concrètement

7.1. 1. Création de la Base de Données et des Tables

```
-- Supprime la base de données si elle existe.
DROP DATABASE IF EXISTS restaurant;
```

```
-- Création de la base de données.
CREATE DATABASE restaurant;
```

```
-- Sélection de la base.
USE restaurant;
```

```
-- Création de la table client.
CREATE TABLE client (
    id_client INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    nom VARCHAR(50),
    prenom VARCHAR(50),
    email VARCHAR(100)
);
```

```
-- Création de la table commande.
CREATE TABLE commande (
    id_commande INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    id_client INT,
    date_commande DATE,
    total DECIMAL(10, 2),
    FOREIGN KEY (id_client) REFERENCES client(id_client)
);
```

```sql
-- Création de la table categorie.
CREATE TABLE categorie (
    id_categorie INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    nom_categorie VARCHAR(50)
);
```

```sql
-- Création de la table plat.
CREATE TABLE plat (
    id_plat INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    nom_plat VARCHAR(50),
    id_categorie INT,
    prix DECIMAL(10, 2),
    FOREIGN KEY (id_categorie) REFERENCES categorie(id_categorie)
);
```

```sql
-- Données pour la table `client`
INSERT INTO client (nom, prenom, email) VALUES
('Martin', 'Lucie', 'lucie.martin@example.com'),
('Bernard', 'Julien', 'julien.bernard@example.com'),
('Kuong', 'Émilie', 'emilie.kuong@example.com'),
('Petit', 'Sophie', 'sophie.petit@example.com'),
('Robert', 'Christophe', 'christophe.robert@example.com');
```

```sql
-- Données pour la table `categorie`
INSERT INTO categorie (nom_categorie) VALUES
('Entrée'),
('Plat principal'),
('Dessert'),
('Boisson');
```

```sql
-- Données pour la table `plat`
INSERT INTO plat (nom_plat, id_categorie, prix) VALUES
('Salade Niçoise',        (SELECT categorie.id_categorie FROM categorie where
categorie.nom_categorie = 'Entrée'),        12.50),
('Steak Frites',        (SELECT categorie.id_categorie FROM categorie where
categorie.nom_categorie = 'Plat principal'),18.90),
('Mousse au Chocolat',    (SELECT categorie.id_categorie FROM categorie where
categorie.nom_categorie = 'Dessert'),        6.50),
('Poulet Basquaise',    (SELECT categorie.id_categorie FROM categorie where
categorie.nom_categorie = 'Plat principal'),16.50),
('Tarte Tatin',            (SELECT categorie.id_categorie FROM categorie where
categorie.nom_categorie = 'Dessert'),        8.00),
('Limonade Maison',        (SELECT categorie.id_categorie FROM categorie where
categorie.nom_categorie = 'Boisson'),        3.75);
```

```sql
-- Données pour la table `commande`
INSERT INTO commande (id_client, date_commande, total) VALUES
((SELECT client.id_client FROM client where client.email = 'lucie.martin@example.com'),
'2023-09-01',45.90),
((SELECT client.id_client FROM client where client.email = 'julien.bernard@example.com'),
'2023-09-02',24.25),
((SELECT client.id_client FROM client where client.email = 'emilie.kuong@example.com'),
'2023-09-03',19.75),
((SELECT client.id_client FROM client where client.email = 'sophie.petit@example.com'),
'2023-09-04',34.75),
((SELECT client.id_client FROM client where client.email =
'christophe.robert@example.com'),'2023-09-05',22.50);
```

7.2. 2. Requêtes SQL

```sql
-- 1. Sélectionne toutes les colonnes de la table client.
SELECT * FROM client;
```

```sql
-- 2. Sélectionne le nom et l'email des clients dont le nom commence par "A".
SELECT nom, email FROM client WHERE nom LIKE 'A%';
```

```sql
-- 3. Sélectionne le nombre total de clients.
SELECT COUNT(*) AS nombre_clients FROM client;
```

```sql
-- 4. Sélectionne le plat le moins cher.
SELECT * FROM plat ORDER BY prix LIMIT 1;
```

```sql
-- 5. Sélectionne les plats dont le prix est supérieur à 10 euros et dans une catégorie spécifique.
SELECT * FROM plat WHERE prix > 10 AND id_categorie = (SELECT id_categorie FROM categorie WHERE nom_categorie = 'plat principal');
```

```sql
-- 6. Sélectionne les commandes effectuées par un client donné (ID 1).
SELECT * FROM commande WHERE id_client = 1;
```

```sql
-- 7. Sélectionne le montant total de toutes les commandes.
SELECT SUM(total) AS montant_total FROM commande;
```

```sql
-- 8. Met à jour le nom d'un client (ID 1).
UPDATE client SET nom = 'Nouveau Nom' WHERE id_client = 1;
```

```sql
-- 9. Insère un nouveau plat dans la table plat.
INSERT INTO plat (nom_plat, prix, id_categorie) VALUES ('Nouveau plat', 15.99, (SELECT id_categorie FROM categorie WHERE nom_categorie = 'Entrée'));
```

```sql
-- 10. Supprime un plat de la table plat (ID 1).
DELETE FROM plat WHERE id_plat = 1;
```

```sql
-- 11. Sélectionne les plats en ordre décroissant de prix.
SELECT * FROM plat ORDER BY prix DESC;
```

```sql
-- 12. Sélectionne les clients ayant passé une commande avec un montant supérieur à 50 euros.
SELECT DISTINCT c.* FROM client c INNER JOIN commande cm ON c.id_client = cm.id_client WHERE cm.total > 50;
```

```sql
-- 13. Sélectionne les plats avec leur catégorie respective.
SELECT p.nom_plat AS plat, c.nom_categorie AS categorie FROM plat p INNER JOIN categorie c ON
p.id_categorie = c.id_categorie;
```

```sql
-- 14. Sélectionne les clients n'ayant jamais passé de commande.
SELECT * FROM client WHERE id_client NOT IN (SELECT DISTINCT id_client FROM commande);
```

```sql
-- 15. Sélectionne les plats et leur nombre de commandes respectif, triés par nombre de
commandes décroissant.
SELECT p.nom_plat AS plat, COUNT(cm.id_commande) AS nombre_commandes FROM plat p LEFT JOIN
commande cm ON p.id_plat = cm.id_plat GROUP BY p.id_plat ORDER BY nombre_commandes DESC;
```

8. SQL (légèrement) avancé ...

8.1. Jointures Variées (INNER JOIN, LEFT JOIN, RIGHT JOIN, CROSS JOIN)

```sql
-- INNER JOIN récupère les enregistrements correspondants dans les deux tables.
SELECT client.nom, commande.date_commande
FROM client
INNER JOIN commande ON client.id_client = commande.id_client;
```

```sql
-- LEFT JOIN inclut tous les enregistrements de la table de gauche et les correspondants de la
table de droite.
SELECT client.nom, commande.date_commande
FROM client
LEFT JOIN commande ON client.id_client = commande.id_client;
```

```sql
-- RIGHT JOIN n'est pas supporté dans toutes les bases de données, donc vérifiez la compatibilité.
-- Elle inclut tous les enregistrements de la table de droite et les correspondants de la table de gauche.
-- Si votre système de gestion de base de données ne supporte pas RIGHT JOIN, utilisez LEFT JOIN avec l'ordre des tables inversé.
SELECT commande.date_commande, client.nom
FROM commande
LEFT JOIN client ON commande.id_client = client.id_client;
```

```sql
-- CROSS JOIN produit le produit cartésien des deux tables.
SELECT client.nom, plat.nom_plat
FROM client
CROSS JOIN plat;
```

8.2. Fonctions SQL Avancées (GROUP BY, HAVING, DISTINCT)

```sql
-- Utilisation de GROUP BY avec HAVING pour filtrer les résultats.
SELECT categorie.nom_categorie, COUNT(*) AS Nombre_plats
FROM plat
JOIN categorie ON plat.id_categorie = categorie.id_categorie
GROUP BY categorie.nom_categorie
HAVING COUNT(*) > 5;
```

```sql
-- Utilisation de DISTINCT pour éviter les doublons.
SELECT DISTINCT client.prenom
FROM client;
```

8.3. Fonctions de Date et d'Heure

```sql
-- Extrait l'année de la date de commande.
SELECT id_commande, YEAR(date_commande) AS Annee_commande
FROM commande;
```

```sql
-- Calcule la différence en jours entre deux dates.
SELECT DATEDIFF(CURDATE(), date_commande) AS Jours_Depuis_commande
FROM commande;
```

8.4. Utilisation de Fonctions d'Aggrégation (SUM, AVG, MIN, MAX)

```sql
-- Calcule le total des ventes, la commande moyenne, le montant maximum et minimum des commandes.
SELECT SUM(total) AS Total_Ventes, AVG(total) AS Moyenne_commande, MIN(total) AS commande_Min, MAX(total) AS commande_Max
FROM commande;
```

8.5. Sous-requêtes et Opérations sur les Sets (IN, NOT IN, EXISTS)

```sql
-- Utilise IN pour sélectionner les clients ayant passé une commande de plus de 100 euros.
SELECT nom, prenom FROM client
WHERE id_client IN (SELECT id_client FROM commande WHERE total > 100);
```

```sql
-- Utilise EXISTS pour vérifier l'existence de certaines commandes.
SELECT nom FROM client c
WHERE EXISTS (SELECT 1 FROM commande co WHERE co.id_client = c.id_client AND total > 150);
```

8.6. Insertions, Mises à jour, et Suppressions de données

```sql
-- Insertion d'un nouveau client.
INSERT INTO client (nom, prenom, email) VALUES ('Dupont', 'Jean', 'jean.dupont@example.com');
```

```sql
-- Mise à jour de l'email d'un client.
UPDATE client SET email = 'nouveau.email@example.com' WHERE id_client = 101;
```

```sql
-- Suppression d'un client.
DELETE FROM client WHERE id_client = 101;
```

8.7. Transactions (START TRANSACTION, COMMIT, ROLLBACK)

Les transactions en SQL sont des séquences d'opérations de gestion de base de données qui sont traitées de manière logique et indivisible. Elles sont essentielles pour maintenir l'intégrité des données, en particulier dans les environnements où plusieurs utilisateurs ou applications accèdent et modifient simultanément la base de données. Une transaction en SQL commence par une commande de démarrage et se termine par un commit ou un rollback.

8.7.1. Principe des Transactions

1. **Atomicité** : Chaque transaction est atomique, ce qui signifie qu'elle est indivisible. Toutes les opérations au sein de la transaction sont effectuées avec succès, ou aucune n'est appliquée. Si une opération échoue, toutes les modifications précédentes dans la transaction sont annulées (rollback).

2. **Cohérence** : Une transaction transforme la base de données d'un état valide à un autre état valide, en préservant l'intégrité des données. Toutes les règles d'intégrité doivent être respectées.

3. **Isolation** : Chaque transaction doit être isolée des autres transactions. Les modifications effectuées dans une transaction ne doivent pas être visibles par les autres transactions avant que la transaction ne soit terminée (commit).

4. **Durabilité** : Une fois qu'une transaction a été validée (commit), les modifications qu'elle a introduites dans la base de données doivent être permanentes, même en cas de panne du système.

8.7.2. Exemple de Transaction en SQL

Pour illustrer comment les transactions fonctionnent en pratique, imaginons une base de données de gestion de comptes bancaires où les transactions sont cruciales pour assurer que tous les transferts de fonds sont effectués de manière sécurisée et cohérente.

```sql
-- Démarrage de la transaction
START TRANSACTION;
```

```sql
-- Tentative de transfert de 100 euros du compte 001 vers le compte 002
-- Débit du compte 001
UPDATE Comptes SET solde = solde - 100 WHERE numero_compte = '001';
```

```sql
-- Crédit du compte 002
UPDATE Comptes SET solde = solde + 100 WHERE numero_compte = '002';
```

```
-- Vérification que le compte débiteur a suffisamment de fonds (ne doit pas être négatif)
SELECT solde INTO @solde FROM Comptes WHERE numero_compte = '001';
IF @solde < 0 THEN
    -- Si le solde est insuffisant, annulation de la transaction
    ROLLBACK;
ELSE
    -- Si tout est en règle, validation de la transaction
    COMMIT;
END IF;
```

8.7.3. Commentaires sur l'Exemple

- **START TRANSACTION** démarre la transaction.
- Les commandes **UPDATE** sont utilisées pour transférer les fonds entre les comptes.
- La commande **SELECT INTO** récupère le solde après le débit pour vérifier si le compte ne passe pas en négatif.
- **IF** permet de tester si le solde est inférieur à zéro. Si c'est le cas, la transaction est annulée avec **ROLLBACK**; sinon, elle est validée avec **COMMIT**.
- **ROLLBACK** annule toutes les modifications effectuées dans la transaction.
- **COMMIT** applique toutes les modifications de manière permanente dans la base de données.
- **@solde** cré une variable

    1. Portée et Durée de vie : Les variables utilisateur comme @solde ont une portée limitée à la session dans laquelle elles sont définies. Elles perdent leur valeur une fois la session terminée.
    2. Sécurité des transactions : L'utilisation de cette variable dans le contexte d'une transaction bancaire est cruciale pour assurer l'intégrité des opérations financières et prévenir les erreurs telles que les découverts non autorisés.

Cet exemple montre l'utilisation essentielle des transactions pour gérer des opérations financières complexes de manière sécurisée et fiable, garantissant ainsi l'intégrité et la cohérence des données dans des systèmes transactionnels critiques.