MySQL

niveau 2

Introduction

MySQL est un système de gestion de bases de données relationnelles (SGBDR). C'est un logiciel open-source soutenu par Oracle Corporation.

Ce logiciel occupe une place de choix parmi les outils de gestion de bases de données, étant largement utilisé tant par le grand public, notamment dans le domaine des applications web, que par des professionnels.

Il se positionne en concurrent direct de solutions telles qu'Oracle, PostgreSQL et Microsoft SQL Server.

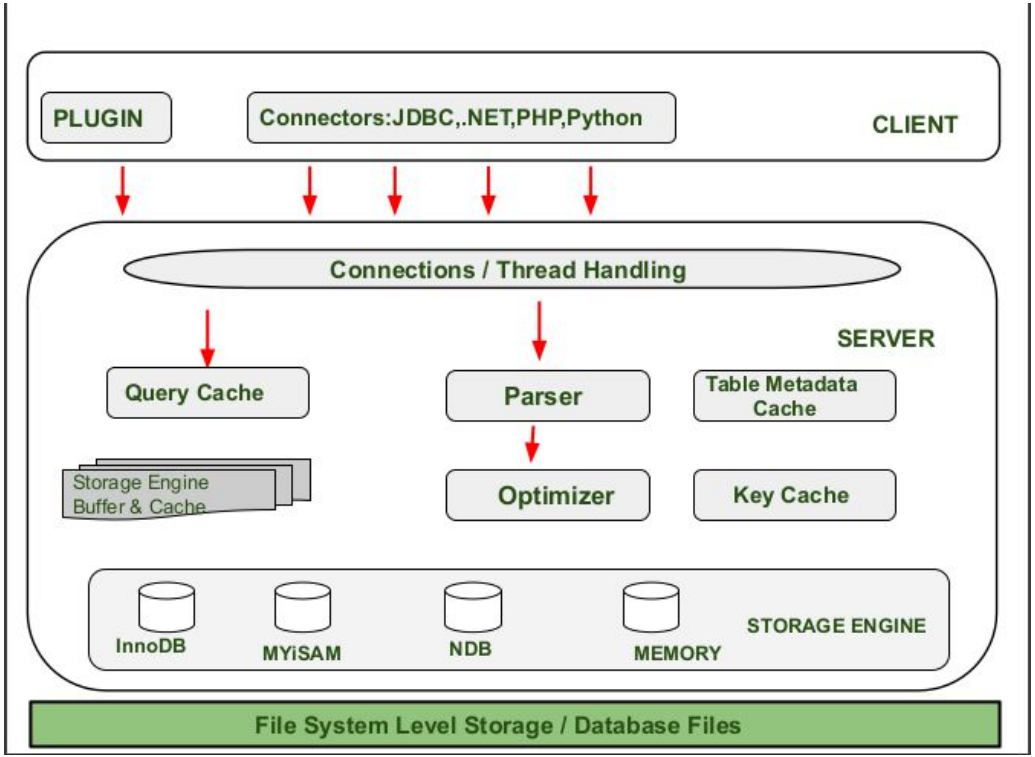
En 2009, MySQL a été acquis par Oracle Corporation, renforçant ainsi sa position sur le marché des bases de données relationnelles.

Module 1: Maîtrise du schéma interne

Module 1: Maîtrise du schéma interne

- Exploration approfondie de la structure interne d'une base de données.

Architecture de MySQL



Architecture de MySQL

Connections
Gère la connexion des utilisateurs. Cette partie s'occupe de l'authentification des utilisateurs. Un thread est créé pour chaque connexion.

Parser
C'est un composant qui construit une structure de données (parse tree) à partir de la requête de l'utilisateur. Le erreurs de syntax sont détectées à ce niveau.

Optimizer
S'exécute après le parsing pour trouver le meilleurs plan d'exécution.

Query cache
C'est un cache contenant des précédentes requêtes et leurs résultats. Si une nouvelle requête est similaire à celle du cache, le résultat peut être récupéré du cache.

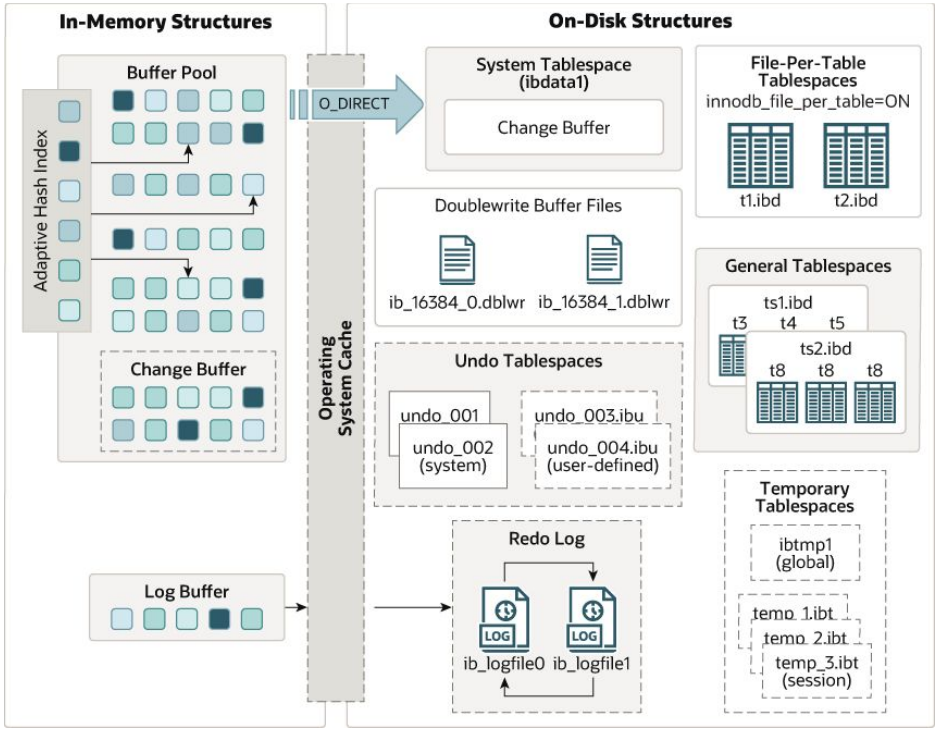
Architecture de MySQL

Table Metadata Cache
C’est une zone mémoire réservée pour stocker des informations sur les objets de la base de données (tables, indexes, …)

Key Cache
C’est une clé qui identifie de manière unique un objet du cache.

Storage Layer
Dans le serveur MySQL, différents types de moteurs de stockage sont utilisés en fonction des situations et des besoins : InnoDB, MyISAM, NDB, Memory, etc.
Les moteurs de stockage contiennent toutes les tables créées par l'utilisateur dans la base de données. Il permettent de récupérer et de stocker les données MySQL.

Architecture de InnoDB



Architecture de InnoDB

Buffer pool
C’est un cache de tables et d’indexes. Permet l’accès depuis la mémoire des données les plus fréquemment utilisées.

Adaptative Hash Index
Si une table tient presque entièrement en mémoire et que InnoDB voit que la construction d’un index de hachage serait bénéfique, alors InnoDB génère cet index.

The system tablespace
C’est l’espace disque qui stocke tous les changements de données. Il persiste les données du Change Buffer.

Change buffer
Un cache qui stocke tous les changements de données au niveau des indexes secondaires.
Les opérations INSERT, UPDATE et DELETE peuvent affecter un index qui n’est pas dans le cache.
Ces opérations sont appliquées ultérieurement en batches pour réduire les opérations I/O.

Architecture de InnoDB

File-Per-Table tablespace
Cet espace contient des fichiers. Chaque fichier contient des données et des indexes associés à une table.

General Tablespace
Espace partagé créé lors de l'utilisation de `CREATE TABLESPACE`. Cet espace est capable de stocker des données de plusieurs tables.

Undo tablespace
Contient les `undo logs`. Informations sur comment revenir sur tous les changements faits pendant une transaction.

Temporary tablespaces
Contient les données sur les tables temporaires créées par l'utilisateur ou par l'optimiseur d'exécution de requêtes.

Log buffer
Une zone mémoire dans laquelle les modifications de données associées à une transaction sont temporairement stockées avant d'être écrites sur le disque (data files). Son rôle principal est lié au processus de récupération après une panne (rollback) et à l'assurance que les transactions sont correctement consignées dans les fichiers journaux.

Architecture de InnoDB

Redo log
Egalement appelé journal de récupération, est un fichier sur le disque qui enregistre les modifications apportées aux données. Il est utilisé pour garantir la durabilité des transactions en cas de panne du système.
Le contenu du log buffer est périodiquement (ou lorsqu'il est suffisamment rempli) écrit dans le redo log sur le disque de manière asynchrone.

En cas de crash, le contenu du redo log est utilisé pour reconstituer (rollforward) les modifications non encore écrites sur le disque (data files). Lors du recovery, les données sont rejouées automatiquement pendant l'initialisation et avant que les connexions ne soient acceptées.

Ensemble, le log buffer et le redo log assurent la sécurité transactionnelle en garantissant que les modifications de données sont enregistrées de manière durable sur disque.

Architecture de InnoDB

Undo Tablespaces

Contient des informations qui permettent de revenir sur les derniers changements effectués par une transaction.

Module 2 : Les vues en profondeur

Module 2 : Les vues en profondeur

- Création, utilisation et optimisation.
- Les vues matérialisées.

Les vues

Une vue représente une table virtuelle. Vous pouvez joindre plusieurs tables dans une vue et utiliser la vue pour présenter les données comme si elles provenaient d'une seule table.

Les vues (views) ont les caractéristiques suivantes

- N'accepte PAS les paramètres.
- Peut être utilisé comme élément de base dans une requête plus large.
- Ne peut contenir qu'une seule requête SELECT (sauf avec UNION et UNIONALL).
- Peut (dans certains cas) être utilisée comme cible d'une instruction INSERT, UPDATE ou DELETE.

Les vues

Voici un exemple de déclaration de vue

```
CREATE VIEW vw_user_profile
AS
  SELECT A.user_id, B.profile_description
  FROM tbl_user A
  LEFT JOIN tbl_profile B ON A.user_id = b.user_id;
```

La vue précédente utilise 2 tables dans sa requête (tbl_user et tbl_profile). L'utilisation de la vue `vw_user_profile` est équivalent à l'utilisation de la requête sous-jacente. Néanmoins la vue simplifie l'écriture de la requête.

```
SELECT * FROM vw_user_profile;
```

Instructions DML sur les vues

Les views (vues) peuvent être utilisées avec des instructions INSERT, UPDATE et DELETE lorsqu'elles

- Se composent que d'une seule table

```
CREATE VIEW ma_vue AS SELECT column1, column2 FROM ma_table;
```

- Pour les vues sur plusieurs tables, la clé unique ou primaire de la vue doit correspondre à la clé unique ou primaire d'au moins une des tables.

```
CREATE VIEW updatable_multitable_view AS
SELECT t1.id, t1.name, t2.description
FROM table1 t1 INNER JOIN table2 t2 ON t1.id = t2.id;
```

- La requête ne doit pas contenir de DISTINCT, GROUP BY, HAVING, UNION ou de sous-requêtes.
- La clause `SET` ne doit pas faire référence à des fonctions ou à des expressions.

Exemple d’instruction DML sur une vues

```sql
CREATE TABLE table1 (
    id INT PRIMARY KEY,
    name VARCHAR(255));

CREATE TABLE table2 (
    id INT PRIMARY KEY,
    description VARCHAR(255));

INSERT INTO table1 VALUES (1, 'John');
INSERT INTO table2 VALUES (1, 'Employee');

--Création de la vue
CREATE VIEW updatable_multi_table_view AS SELECT t1.id, t1.name, t2.description
FROM table1 t1 JOIN table2 t2 ON t1.id = t2.id;

--Mise à jour de la vue
UPDATE updatable_multi_table_view SET name = 'Jane' WHERE id = 1;
```

Les vues matérialisées

Une vue matérialisée est associées à une table physique stockée dans la base de données qui contient les résultats d'une requête SQL prédéfinie.

Elle est pré-calculée et stockée en tant que table réelle.

Les vues standards sont des requêtes dynamiques qui accèdent aux données en temps réel, tandis que les vues matérialisées sont des tables physiques précalculées qui permettent un accès plus rapide aux données.

Les vues matérialisées

Les vues matérialisées n'étant pas supportées dans MySQL, voici un exemple d'utilisation d'une vue matérialisée en PostgreSQL

```
CREATE TABLE notes (id int, note decimal(4, 2));

INSERT INTO notes VALUES (1, 10.3);
INSERT INTO notes VALUES (1, 9.7);

INSERT INTO notes VALUES (2, 11.1);
INSERT INTO notes VALUES (2, 12.9);

--création d'une vue matérialisée
CREATE MATERIALIZED VIEW mv_notes_stats AS
SELECT id, avg(note) AS moyenne, count(*) AS nb
FROM notes
GROUP BY id;
```

Les vues matérialisées

Les vues matérialisées ne sont pas mises à jour automatiquement. Lors de la création une table physique est implicitement associée à la vue.

Cette table physique sert à récupérer les données de la vue rapidement, mais ces données représentent un snapshot de la base de données au moment de la création de la vue matérialisée.

Il y a une commande spécifique pour rafraîchir la vue.
REFRESH MATERIALIZED VIEW mv_notes_stats;

Notez que lors du rafraîchissement de la vue, tout `select` sur cette vue matérialisée est locké le temps que cette vue se rafraichisse.

Vues matérialisées dans MySQL

MySQL ne supporte pas la création de vue matérialisée. Il existe néanmoins une approche qui peut émuler le comportement d'une vue matérialisée.

Créez une table classique

```
CREATE TABLE mv_notes_stats (id int, avg_val float, nb_val int);
```

Vues matérialisées dans MySQL

Puis créez une procédure stockée qui va rafraîchir les valeurs de la table précédemment créée.

```
DELIMITER $$
CREATE PROCEDURE refresh_notes_stats ()
BEGIN
  TRUNCATE TABLE mv_notes_stats;

  INSERT INTO mv_notes_stats
  SELECT id, avg(note), count(*)
  FROM notes GROUP BY id;
END;
$$
DELIMITER ;
```

Vues matérialisées dans MySQL

Pour rafraîchir les données de la table précédente, il ne manque plus qu'appeler la procédure stockée créée

```
CALL refresh_notes_stats;
```

Module 3 : Assurer l'intégrité des données

Module 3 : Assurer l'intégrité des données

- Comprendre l'intégrité des données à un niveau avancé.
- Mécanismes d’intégrité offerts par les bases de données.

Gestion des contraintes

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

1. Contrainte **PRIMARY KEY**. Clé primaire: spécifie les champs qui identifient la ligne.
2. Contrainte **FOREIGN KEY**. Clé étrangère: spécifie les champs qui référencent une ligne d'une autre table.
3. Contrainte **NOT NULL**. La valeur ne peut pas être nulle.
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 spécifier des contraintes impliquant plusieurs champs

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

Module 4 : Sécurité des données

Module 4 : Sécurité des données

- Confidentialité et mécanismes de sécurité d’accès aux données (DCL - Data Control Language).
- Gestion des droits et privilèges.

Confidentialité et mécanismes de sécurité d’accès

MySQL offre un système granulaire de contrôle des privilèges qui permet aux administrateurs de définir précisément les actions autorisées pour chaque utilisateur, comme SELECT, INSERT, UPDATE, DELETE, etc.

Les privilèges peuvent être définis au niveau de la base de données, de la table, de la colonne ou même au niveau des commandes SQL spécifiques.

Gestion des rôles

Un rôle correspond à une collection de privilèges. Nous pouvons ajouter ou supprimer des privilèges à un rôle.

Un compte utilisateur peut se voir attribuer des rôles, ce qui accorde au compte les privilèges associés à chaque rôle. Cela permet d'attribuer des ensembles de privilèges aux comptes et constitue une alternative pratique à l'octroi de privilèges individuels.

Pour créer un rôle utilisateur

```
CREATE ROLE 'developer';
```

Pour supprimer un rôle utilisateur

```
DROP ROLE 'developer';
```

Gestion des rôles

Vous pouvez octroyer des droits à vos rôles en utilisant la syntaxe suivante
GRANT <droits> ON <objets_base> TO <nom_role>;

Exemples

```
GRANT ALL ON app_db.* TO 'app_developer';
GRANT SELECT ON app_db.* TO 'app_read';
GRANT INSERT, UPDATE, DELETE ON app_db.* TO 'app_write';
```

Pour supprimer des droits d'un rôle particulier
REVOKE <droits> ON <objets_base> FROM <nom_role>;

Exemple

```
REVOKE INSERT, UPDATE, DELETE ON app_db.* FROM 'app_write';
```

Gestion des utilisateurs

Visualisez les utilisateurs créés

```
SELECT host, user FROM mysql.user;
```

Création d'un utilisateur

```
CREATE USER 'user1'@'localhost' IDENTIFIED BY 'pass123';
```

Création d'un utilisateur avec plusieurs rôles associés

```
CREATE USER 'user1'@'localhost' DEFAULT ROLE administrator, developer;
```

Modification d'un compte utilisateur pour lui assigner des rôles par défaut

```
ALTER USER 'joe'@'localhost' DEFAULT ROLE administrator, developer;
```

Vérouillage et dévérouillage d'un compte utilisateur

```
ALTER USER 'jeffrey'@'localhost' ACCOUNT LOCK;
ALTER USER 'jeffrey'@'localhost' ACCOUNT UNLOCK;
```

Suppression d'un utilisateur

```
DROP USER 'user1'@'localhost';
```

Gestion des droits

Vous pouvez assigner des droits à un utilisateur particulier

```
GRANT <droits> ON <objets_base> TO <nom_utilisateur>;
```

Exemple

```
GRANT ALL ON db1.* TO 'jeffrey'@'localhost';
```

Vous pouvez assigner des rôles à plusieurs utilisateurs

```
GRANT <rôles> ON <objets_base> TO <utilisateurs>;
```

Exemple

```
GRANT 'role1', 'role2' TO 'user1'@'localhost', 'user2'@'localhost';
```

Module 5 : Gestion avancée des données

Module 5 : Gestion avancée des données

- Répartition et réplication des données.
- *Les bases de données orientées objet (SGBDR Objet).*

Répartition et réplication des données

MySQL propose deux mécanismes distincts pour gérer la distribution des charges et le failover.

- La répartition (sharding).
- La réplication des données.

Répartition (Sharding)

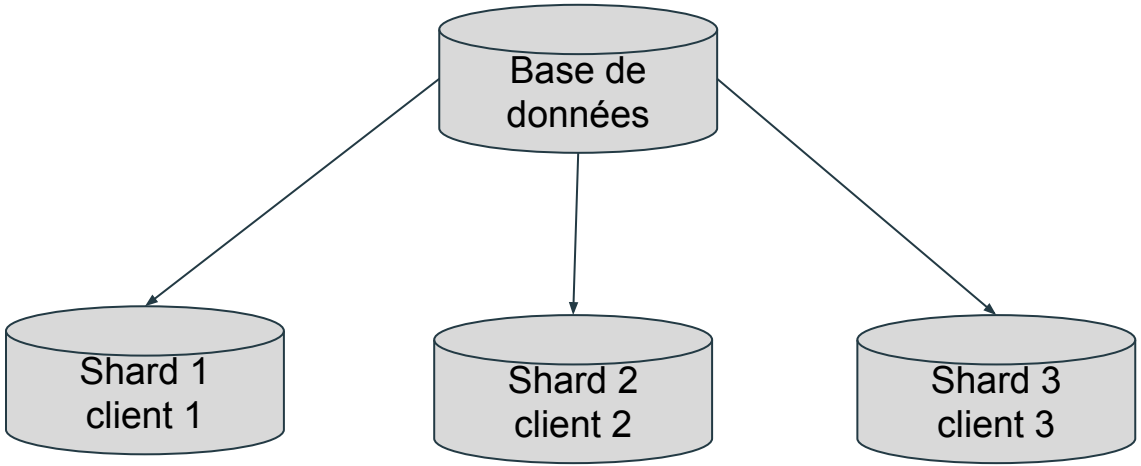
La répartition, également appelée sharding, consiste à diviser horizontalement une base de données en plusieurs fragments appelés shards.
Chaque shard contient une sous-partie des données totales.

Mécanisme
Les données sont réparties en fonction d'un critère spécifié, généralement une colonne de la table (sharding key).
Chaque shard est géré par un serveur distinct.
Les requêtes sont dirigées vers le shard approprié en fonction de la valeur du sharding key.

Avantages
Évite les goulots d'étranglement en répartissant la charge sur plusieurs serveurs.
Améliore les performances en parallélisant les opérations sur les shards.

Répartition (Sharding)

Le sharding est le partitionnement horizontal des données où chaque partition réside dans un nœud distinct ou une machine distincte.



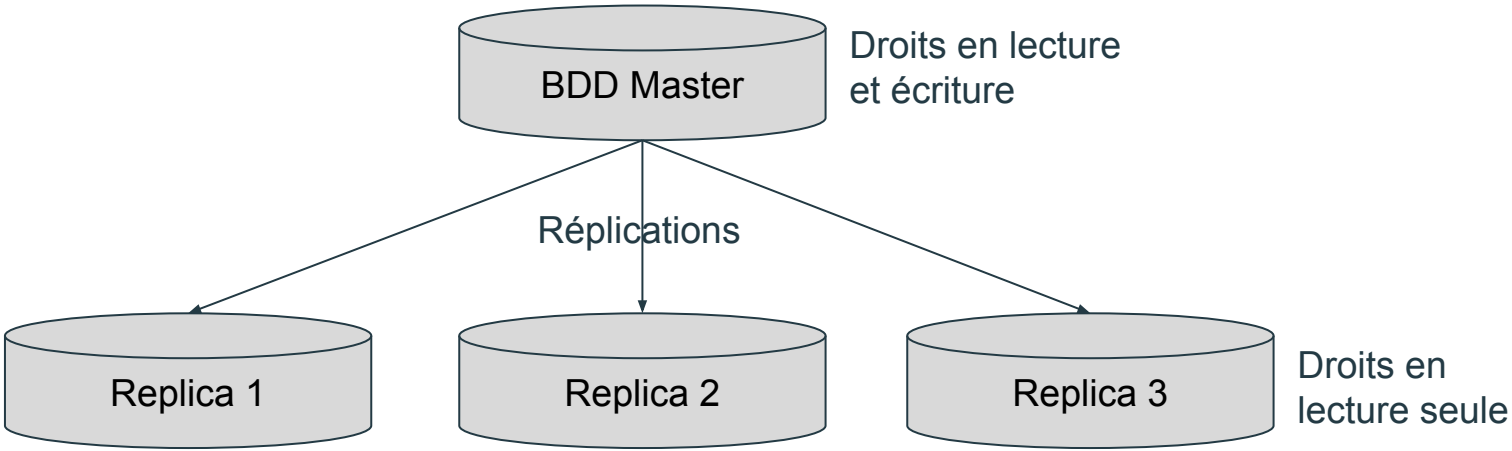
Réplication de données

La réplication consiste à créer et à maintenir une copie identique d'une base de données sur un ou plusieurs serveurs. Cette copie, appelée réplica, est mise à jour en temps réel avec les modifications apportées à la base de données principale.

Mécanisme
Une base de données principale (master) envoie les mises à jour (transactions) aux réplicas. Les réplicas appliquent ces mises à jour pour rester synchronisés avec la base de données principale.

Avantages
Améliore la disponibilité en fournissant des copies de secours des données.
Permet la répartition de la charge en dirigeant les requêtes de lecture vers les réplicas.

Réplication de données



Les bases de données orientées objet

Les bases de données orientées objet (OODB) sont un type de système de gestion de base de données (SGBD) qui étendent le modèle relationnel des bases de données pour prendre en charge la représentation et la manipulation d'objets complexes et de leurs relations.

Contrairement aux bases de données relationnelles traditionnelles, les bases de données orientées objet permettent de stocker des objets avec des attributs.

MySQL n’est pas une base de données orientée objet, mais nous pouvons regarder un exemple en PostgreSQL.

Les bases de données orientées objet

```
-- Création d'un type d'objet (objet composite)
CREATE TYPE address AS (
    street VARCHAR(255),
    city VARCHAR(255),
    zip_code VARCHAR(10));

-- Création d'une table utilisant le type d'objet
CREATE TABLE person (
    person_id INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    name VARCHAR(100),
    contact_info address);

-- Insertion de données
INSERT INTO person (name, contact_info) VALUES
    ('John Doe', ROW('123 Main St', 'Cityville', '12345')),
    ('Jane Smith', ROW('456 Oak St', 'Townburg', '56789'));
```

Module 6 : Requêtes avancées

Module 6 : Requêtes avancées

- Sous-requêtes et optimisation.
- Ordres ensemblistes : UNION, MINUS, INTERSECT.
- Les requêtes imbriquées et les sous-requêtes.

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 opérations ensemblistes

Les opérations ensemblistes en SQL, comprenant les opérateurs UNION, UNION ALL, EXCEPT et INTERSECT, permettent de combiner ou de comparer des résultats de requêtes.

- Les colonnes dans les requêtes liées par ces opérations doivent être du même type et dans le même ordre.
- Les opérations ensemblistes peuvent être utilisées dans des sous-requêtes, des vues ou des requêtes indépendantes.
- L'ordre d'évaluation des résultats peut varier en fonction du système de gestion de base de données.

Les opérations ensemblistes

La clause UNION

La clause UNION est utilisée pour combiner les résultats de deux ou plusieurs requêtes et éliminer 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';
```

Les opérations ensemblistes

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.

Les opérations ensemblistes

La clause EXCEPT

L'opérateur EXCEPT retourne les lignes présentes dans la première requête mais absentes de la deuxième requête.

Exemple

```
SELECT column1 FROM table1
EXCEPT
SELECT column1 FROM table2;
```

Les opérations ensemblistes

La clause INTERSECT

L'opérateur INTERSECT retourne les lignes qui sont communes aux résultats des deux requêtes.

Exemple

```
SELECT column1 FROM table1
INTERSECT
SELECT column1 FROM table2;
```

Optimisations

L'optimisation des requêtes vise à améliorer les performances en réduisant le temps d'exécution et les ressources nécessaires à l’exécution.

Voici une liste non exhaustive de techniques d’optimisation de requêtes

- Utilisez des indexes uniquement lorsque nécessaire.
- Évitez de récupérer des données inutiles.
- Utilisez des vues matérialisées.
- Partitionnez les grandes tables.
- Utilisez un cache applicatif.
- Analysez le plan d’exécution d’une requête.

Création d’indexes

Les indexes améliorent les performances en accélérant la recherche des données.
Cependant, créer trop d'indexes peut avoir un impact négatif sur les performances lors des opérations de modification des données (INSERT, UPDATE, DELETE).

Utilisez des indexes là où ils sont vraiment nécessaires.

Certains types de moteurs de stockage (storage engines) supportent en plus du B-Tree index, le hash index.

L’index de hachage (HASH INDEX) ne peut être utilisé que pour tester les conditions d’égalité. Il consomme moins de mémoire, il est également plus performant que le B-Tree.

L’index B-Tree (BTREE INDEX) est plus polyvalent. Il construit un arbre binaire de recherche qui peut être utilisé pour les comparaisons et l’égalité.
Peut être utile pour améliorer les performances des clauses WHERE, JOIN et ORDER BY.

Ne récupérez pas des données inutiles

Énumérez explicitement les colonnes nécessaires plutôt que d'utiliser SELECT *.
Cela réduit la quantité de données transférées et améliore les performances.

Exemple

```
-- Évitez cela
SELECT * FROM table1;

-- Utilisez plutôt
SELECT column1, column2 FROM table1;
```

Utilisez des Vues Matérialisées

Les vues matérialisées stockent physiquement les résultats d'une requête, offrant des performances améliorées pour les requêtes complexes, au détriment de la fraîcheur des données.

Malheureusement les vues matérialisées ne sont pas supportées dans MySQL.

Partitionnez les Grandes Tables

La partitioning divise une grande table en plusieurs parties (partitions) basées sur une condition spécifiée, améliorant ainsi les performances lors de la recherche de données.

Exemple

```
-- Création de partitionnement par RANGE
CREATE TABLE sales (
    sale_date DATE,
    amount DECIMAL(10,2)
)
PARTITION BY RANGE (YEAR(sale_date)) (
    PARTITION p0 VALUES LESS THAN (1990),
    PARTITION p1 VALUES LESS THAN (2000),
    PARTITION p2 VALUES LESS THAN (2010),
    PARTITION p3 VALUES LESS THAN (2020),
    PARTITION p4 VALUES LESS THAN MAXVALUE
);
```

Partitionnez les Grandes Tables

Il existe plusieurs types de partitioning

- **Range partitioning**. Partitionne la table en fonction des valeurs de colonne comprises dans une plage donnée.
- **List partitioning**. Semblable au partitionnement par RANGE, sauf que la partition est sélectionnée en fonction des colonnes correspondant à l'une d'un ensemble de valeurs discrètes.
- **Hash partitioning**. Partitionne en fonction des valeurs retournées par la fonction de hachage appliquée à une ou plusieurs colonnes.
- **Key partitioning**. Similaire au Hash partitioning. Peut contenir des colonnes de type non entier. MySQL fournit la fonction de hachage.

Utilisez un Cache Applicatif

Un cache applicatif peut stocker en mémoire les résultats fréquemment utilisés, réduisant ainsi la charge sur la base de données.

Exemple

Utilisation de Redis, Couchbase ou Memcached pour stocker des résultats de requêtes fréquemment utilisés.

Analysez le plan d’exécution

La commande EXPLAIN fournit des informations sur la manière dont MySQL exécute une requête, aidant à identifier les goulots d'étranglement et à optimiser les requêtes.

Exemple

```
EXPLAIN SELECT column1 FROM table1 WHERE condition;
```

Par défaut le format d’affichage de la commande `EXPLAIN` est le format traditionnel `TRADITIONAL`.

Utilisez le format `TREE` pour une meilleur représentation.

Analysez le plan d’exécution

Pour voir le format d’affichage de la commande `EXPLAIN`

```
SELECT @@explain_format;
```

Pour sélectionner le mode d’affichage `TREE`

```
SET @@explain_format=TREE;
```

Exemple

```
EXPLAIN SELECT * FROM test;
```

Affiche

```
+----------------------------------------+
| EXPLAIN                                |
+----------------------------------------+
| -> Table scan on test  (cost=0.65 rows=4)
|
+----------------------------------------+
```

Module 7 : Maîtrise des jointures

Module 7 : Maîtrise des jointures

- Alias de tables et de champs.
- Jointures externes : concepts avancés et optimisations.
- Auto-jointures et Tetha-jointures.

Alias de tables

Les alias de tables sont des noms temporaires que vous attribuez à une table pour simplifier la syntaxe des requêtes.

Ils sont souvent utilisés pour rendre les requêtes plus lisibles, en particulier lorsque vous avez des tables avec des noms longs ou complexes.

Exemple

```
SELECT e.employee_id, e.employee_name, d.department_name
FROM employees e
INNER JOIN departments d ON e.department_id = d.department_id;
```

Alias de champs

Les alias de champs permettent de renommer temporairement une colonne dans le résultat d'une requête. Cela est utile pour clarifier les noms de colonnes, effectuer des calculs ou rendre les résultats plus compréhensibles.

Exemple

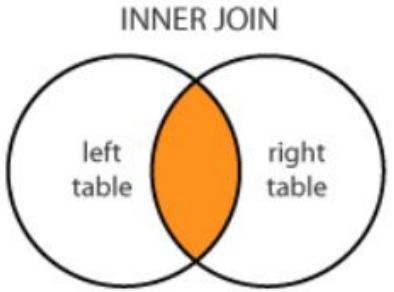
```
SELECT employee_id, employee_name, salary * 1.1 AS increased_salary
FROM employees;
```

INNER JOIN

L'INNER JOIN renvoie uniquement les lignes qui ont des correspondances dans les deux tables.

Exemple

```
SELECT employees.employee_id, employees.employee_name, departments.department_name
FROM employees
INNER JOIN departments
ON employees.department_id = departments.department_id;
```

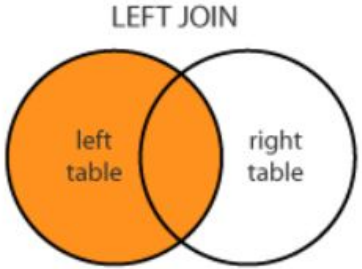


LEFT JOIN

Le LEFT JOIN renvoie toutes les lignes de la table de gauche (table principale), et les lignes correspondantes de la table de droite. Si aucune correspondance n'est trouvée, les colonnes de la table de droite auront des valeurs NULL.

Exemple

```
SELECT employees.employee_id, employees.employee_name, departments.department_name
FROM employees
LEFT JOIN departments
ON employees.department_id = departments.department_id;
```

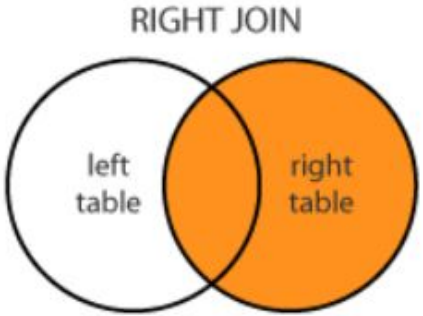


RIGHT JOIN

Le RIGHT JOIN est similaire au LEFT JOIN, mais renvoie toutes les lignes de la table de droite, et les lignes correspondantes de la table de gauche. Les valeurs NULL seront présentes du côté gauche si aucune correspondance n'est trouvée.

Exemple

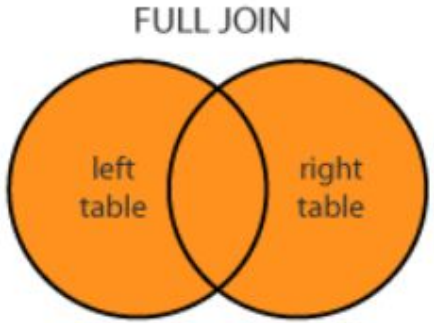
```
SELECT employees.employee_id, employees.employee_name, departments.department_name
FROM employees
RIGHT JOIN departments
ON employees.department_id = departments.department_id;
```



FULL OUTER JOIN

Le FULL OUTER JOIN renvoie toutes les lignes lorsqu'il y a une correspondance dans l'une des tables. Si aucune correspondance n'est trouvée, les colonnes de la table sans correspondance auront des valeurs NULL.

```
SELECT employees.employee_id, employees.employee_name, departments.department_name
FROM employees
FULL OUTER JOIN departments
ON employees.department_id = departments.department_id;
```



Les auto jointures

Une auto-jointure se produit lorsqu'une table est jointe à elle-même, souvent en utilisant un alias pour distinguer les références à la même table.

Exemple

```
SELECT e1.employee_name, e2.employee_name AS manager_name
FROM employees e1
INNER JOIN employees e2 ON e1.manager_id = e2.employee_id;
```

Les theta jointures

Les theta jointures sont des jointures basées sur des conditions autres que l'égalité (>, <, >=, <=, etc.).

Exemple

```
SELECT DISTINCT h.name
FROM hospitals h, schools s
WHERE distance(h.location,s.location) < 5;
```

Module 8 : SQL dynamique et procédural

Module 8 : SQL dynamique et procédural

- Fonctions : création, utilisation et optimisation.
- Procédures stockées : meilleures pratiques.
- Triggers : conception, utilisation et pièges courants.

Procédure stockée

Qu'est-ce qu'une Procédure Stockée ?

Une procédure stockée est une série d'instructions SQL regroupées sous un nom unique et stockées dans la base de données.
Elle peut être invoquée par d'autres programmes ou procédures.

Exemple

```
DELIMITER //
CREATE PROCEDURE stored_proc_tutorial.spInsererEtudiant(IN nom VARCHAR(20), IN prenom VARCHAR(20))
BEGIN
    INSERT INTO etudiant (nom, prenom) VALUES (nom, prenom);
END //
DELIMITER ;
```

Fonction stockée

Qu'est-ce qu'une Fonction Stockée ?

Une fonction stockée est similaire à une procédure stockée, mais elle retourne une valeur. Elle est utilisée dans des expressions SQL pour effectuer des calculs ou des opérations.

Exemple

```
DELIMITER $$
CREATE PROCEDURE stored_proc_tutorial.fpGetStudentName(IN id INT)
BEGIN
    DECLARE nomEtudiant VARCHAR(20);
    SET nomEtudiant = (SELECT nom FROM etudiant where id_etudiant = id);
    RETURN (nomEtudiant);
END $$
DELIMITER ;
```

Utilisation des Procédures et Fonctions

Les procédures et fonctions stockées peuvent être appelées depuis d'autres requêtes SQL ou programmes.

Appel d’une procédure stockée

```
CALL nom_de_la_procedure(arg1, arg2);
```

Appel d’une fonction stockée

```
SELECT nom, nom_de_la_fonction(prod_cost,prod_price) AS profit FROM products;
```

Voici un appel de fonction stockée à partir d’une procédure ou fonction stockée

```
SET customerLevel = nom_de_la_fonction(credit);
```

Procédures et fonctions stockées

Les procédures et les fonctions stockées peuvent avoir des paramètres en entrée et des paramètres en sortie.

Par défaut les paramètres sont des paramètres d'entrée, mais nous pouvons stipuler s'il s'agit de paramètres d'entrée ou de paramètres de sortie à l'aide des mots clés.

- `IN` spécifie un paramètre d'entrée.
- `OUT` spécifie un paramètre de sortie.

Procédures et fonctions stockées

Exemple de procédure stockée avec un paramètre en entrée et un paramètre en sortie

```
DELIMITER $$
CREATE PROCEDURE GetCustomerLevel(IN  customerNo INT, OUT customerLevel VARCHAR(20))
BEGIN
  --code de la procédure stockée
END$$
DELIMITER ;
```

L'appel de la fonction précédente peut se fait de cette manière

```
CALL GetCustomerLevel(-131,@customerLevel);
SELECT @customerLevel;
```

Procédures et fonctions meilleurs pratiques

Paramètres
Utilisez des paramètres pour rendre vos procédures / fonctions flexibles.
Vérifiez les paramètres d'entrée et retournez une erreur en cas de mauvais paramètre.

Transactions
Si nécessaire, incluez des transactions pour assurer la cohérence des données.

Gestion des Erreurs
Ajoutez une gestion appropriée des erreurs pour une meilleure robustesse.

Commentaires
Documentez votre code en ajoutant des commentaires pour faciliter la compréhension.

Optimisation
Optimisez vos procédures/fonctions pour des performances optimales.

Avantages des Procédures et Fonctions Stockées

Réutilisation du Code
Les procédures et fonctions stockées permettent la réutilisation du code SQL.

Sécurité
Elles offrent un meilleur contrôle d'accès aux données en limitant l'accès direct aux tables.

Maintenance Facile
Les modifications ne nécessitent qu'une mise à jour de la procédure ou de la fonction.

Les triggers

Qu'est-ce qu'un Trigger ?

Un trigger (déclencheur) est un ensemble d'instructions SQL associées à une table et activé automatiquement en réponse à certains événements liés à cette table, tels que l'insertion, la mise à jour ou la suppression de données.

Les triggers peuvent s’exécuter avant ou après l'événement.

Les triggers facilitent et automatisent des actions au sein d’un Système de Gestion de Base de Données (SGBD).

Les triggers

Création d'un Trigger

La création d'un trigger se fait en spécifiant l'événement déclencheur (INSERT, UPDATE, DELETE), le moment d'activation (BEFORE ou AFTER), la table concernée, et le corps du trigger (les instructions SQL).

Syntaxe

```
CREATE TRIGGER <nom_du_trigger>
{BEFORE | AFTER} {INSERT | UPDATE| DELETE }
ON <nom_table> FOR EACH ROW
<action_du_trigger>;
```

`Action_du_trigger` correspond au code du trigger à exécuter en cas de déclenchement de l'événement.

Les triggers

Exemple

```
DELIMITER //
CREATE TRIGGER after_insert_example
AFTER INSERT ON ma_table
FOR EACH ROW
BEGIN
    -- Instructions SQL
END //
DELIMITER ;
```

Utilisation des Triggers

Les triggers sont automatiquement déclenchés lorsqu'un événement spécifié survient sur la table associée. Ils peuvent être utilisés pour effectuer des actions telles que la validation des données, la mise à jour d'autres tables, ou la journalisation des changements.

Exemple

```
-- Trigger qui met à jour une colonne timestamp lors d'une modification
CREATE TRIGGER before_update_timestamp
BEFORE UPDATE ON ma_table
FOR EACH ROW
SET NEW.updated_at = NOW();
```

Meilleures Pratiques pour les Triggers

Simplicité
Les triggers doivent être aussi simples que possible pour faciliter la compréhension et la maintenance.

Évitez les Actions Complexes
Limitez les actions des triggers pour éviter les problèmes de performances ou de complexité.
Les triggers peuvent avoir un impact très négatif sur les performances en écriture
Soyez prudent pour éviter la création de boucles infinies, où un trigger déclenche un autre trigger.

Documentation
Documentez soigneusement chaque trigger pour expliquer son but, son fonctionnement, et les conséquences éventuelles.

Test Rigoureux
Avant de déployer des triggers en production, effectuez des tests rigoureux pour vous assurer qu'ils fonctionnent comme prévu.

Triggers: autres opérations

Afficher tous les triggers

```
SHOW TRIGGERS
    [{FROM | IN} db_name]
    [LIKE 'pattern' | WHERE expr];
```

Exemple

```
SHOW TRIGGERS LIKE 'acc%';
```

Supprimer un trigger

```
DROP TRIGGER IF EXISTS nom_du_trigger;
```

Les evénements

Les événements sont utilisés pour automatiser des tâches récurrentes ou pour effectuer des opérations à des moments spécifiques, sans nécessiter une intervention manuelle.

Un événement en MySQL est un objet planifié qui exécute automatiquement des tâches à des moments spécifiés ou selon une périodicité définie.

Ces tâches peuvent inclure l'exécution de requêtes SQL, l'appel de procédures stockées, ou d'autres opérations.

MySQL Event Scheduler est chargé de gérer la planification et l'exécution des événements. Ils sont analogues aux cronjobs sous Linux ou aux planificateurs de tâches sous Windows.

Par exemple, vous pouvez planifier un événement qui optimise toutes les tables de la base de données pour qu'il s'exécute tous les dimanches à 1h00 du matin.

Les evénements

Création d'un Événement s’exécutant une fois

```
CREATE EVENT myevent
    ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 1 HOUR
    DO
      UPDATE myschema.mytable SET mycol = mycol + 1;
```

Création d'un Événement s’exécutant périodiquement

```
CREATE EVENT e_hourly
    ON SCHEDULE
      EVERY 1 HOUR
    COMMENT 'Suppression des données de la table chaque heure.'
    DO
      DELETE FROM site_activity.sessions;
```

Les evénements

Afficher tous les triggers d'une base

```
SHOW EVENTS FROM ma_base;
```

Suppression d'un trigger

```
DROP EVENT mon_evenement;
```