Version expérimentale
En cours de validation

RÉSUMÉ THÉORIQUE – FILIÈRE DÉVELOPPEMENT DIGITAL OPTION WEB FULL STACK

M111 – GÉRER LES DONNÉES

Elaboré par :

Mohamed CHAKOUJ
Formateur à ISTA INEZGANE

90 heures

Equipe de rédaction et de lecture

Equipe de rédaction :

Mme Widad Jakjoud: Formatrice en développement digital

M. Mohamed Chakouj : Formateur en développement digital

Equipe de lecture :

Mme Laouija Soukaina : Formatrice animatrice au CDC Digital & IA

Mme Ghizlane El Khattabi : Formatrice animatrice au CDC Digital & IA

SOMMAIRE

1. Exploiter les fonctionnalités avancées d'un SGBD relationnel

Maitriser le langage de programmation procédurale sous MySQL
Optimiser une base de données MySQL
Protéger la base de données MySQL

2. Exploiter les fonctionnalités des bases de données NoSQL MongoDB

Découvrir les bases de données NoSQL
Mettre en place une base de données MongoDB
Modéliser les documents
Manipuler les données avec mongoDB
Effectuer des requêtes depuis des programmes Python
Sécuriser une base de données MongoDB

MODALITÉS PÉDAGOGIQUES

1

LE GUIDE DE SOUTIEN

Il contient le résumé théorique et le manuel des travaux pratiques

2

LA VERSION PDF

Une version PDF est mise en ligne sur l'espace apprenant et formateur de la plateforme WebForce Life

3

DES CONTENUS TÉLÉCHARGEABLES

Les fiches de résumés ou des exercices sont téléchargeables sur WebForce Life

4

DU CONTENU INTERACTIF

Vous disposez de contenus interactifs sous forme d'exercices et de cours à utiliser sur WebForce Life

5

DES RESSOURCES EN LIGNES

Les ressources sont consultables en synchrone et en asynchrone pour s'adapter au rythme de l'apprentissage

PARTIE 1

Exploiter les fonctionnalités avancées d'un SGBD relationnel

Dans ce module, vous allez :

- Maitriser le langage de programmation procédurale sous MySQL
- Optimiser une base de données MySQL
- Pouvoir protéger la base de données MySQL

50 heures

CHAPITRE 1

Maitriser le langage de programmation procédurale sous MySQL

Ce que vous allez apprendre dans ce chapitre :

- Rappeler le langage de manipulation des données (SQL) ;
- Présenter le langage de programmation procédural de MySQL ;
- Distinguer les différents types des programmes MySQL ;
- Maitriser les instructions de bases ;
- Maitriser les structures de contrôle ;
- Gérer les transactions ;
- Gérer les exceptions ;
- Manipuler les curseurs ;
- Créer les procédures stockées et les fonctions ;
- Mettre en place les déclencheurs.

?? heures

CHAPITRE 1

Maitriser le langage de programmation procédurale sous MySQL

1. **Rappeler le langage de manipulation des données (SQL) ;**
2. Présenter le langage de programmation procédural de MySQL ;
3. Distinguer les différents types des programmes MySQL ;
4. Maitriser les instructions de bases ;
5. Maitriser les structures de contrôle ;
6. Gérer les transactions ;
7. Gérer les exceptions ;
8. Manipuler les curseurs ;
9. Créer les procédures stockées et les fonctions ;
10. Mettre en place les déclencheurs.

PARTIE 1

Création d'une base de données :

CREATE DATABASE nom_de_la_base;

CREATE DATABASE *eshop_app_db*;

Suppression d'une base de données :

DROP DATABASE nom_de_la_base;

DROP DATABASE `*eshop_app_db*`;

Création d'une table dans la base de données :

```
USE nom_de_la_base;

CREATE TABLE nom_de_la_table (

nom_champ1 type_champ1(longueur) attributs,

nom_champ2 type_champ2(longueur) attributs, ...

type_clé1 nom_clé1,

type_clé2 nom_clé2, ... )

ENGINE = moteur_bd;
```

```
CREATE DATABASE clients (
id INT NOT NULL AUTO_INCREMENT,
nom CHAR(50) NOT NULL,
prenom CHAR(40),
date_naissance DATE,
adresse VARCHAR(50) DEFAULT ''Agadir'',
PRIMARY KEY (id)
)
ENGINE = InnoDB;
```

Remarque :

- Pour créer une table ayant la même structure qu'une autre:
  CREATE TABLE nouvelle_table **LIKE** ancienne_table ;
- Les données ne sont pas copiées.

PARTIE 1

Suppression d'une table :

L'instruction **DROP TABLE** est utilisée pour supprimer une table existante dans une base de données.

DROP TABLE nom_table;

```
DROP TABLE 'clients';
```

PARTIE 1

Modification d'une table :

- L'instruction **ALTER TABLE** est utilisée pour ajouter, supprimer ou modifier des colonnes dans une table existante.
- L'instruction **ALTER TABLE** est également utilisée pour ajouter et supprimer diverses contraintes sur une table existante.

- **NB**
  - DESCRIBE nom_table; permet d'afficher la structure de la table.

```
ALTER TABLE clients
ADD Email varchar(255);

ALTER TABLE clients
DROP COLUMN ;

ALTER TABLE clients
MODIFY COLUMN date_naissance int;

ALTER TABLE clients
ADD pays VARCHAR(50),
ADD sexe VARCHAR(20) AFTER prenom;

ALTER TABLE clients CHANGE COLUMN adresse ville
VARCHAR(100) NOT NULL;

ALTER TABLE clients
DROP COLUMN pays;

ALTER TABLE commandes
RENAME TO achats;

DESCRIBE clients;
```

PARTIE 1

Sélection des données :

- L'instruction **SELECT** est utilisée pour sélectionner des données dans une base de données.
- **SELECT** [{ **DISTINCT** | **DISTINCTROW** } | **ALL**] listeColonnes
  FROM nomTable1 [,nomTable2]...
  [**WHERE** condition]
  [clauseRegroupement]
  [**HAVING** condition]
  [clauseOrdonnancement]
  [**LIMIT** clause_limitation];

```
SELECT nom, ROUND(YEAR(CURDATE())-
date_naissance) AS age
FROM    clients
WHERE adresse='Agadir'
ORDER BY age DESC;
```

```
SELECT produit.id, produit.designation, produit.pu
,categorie.nom_cat
FROM produit
INNER JOIN categorie
ON produit.id_cat=categorie.id;
```

- **NB**
  - Mettre * au lieu des noms des champs pour sélectionner tous les colonnes.
  - Dans la syntaxe, [] signifier que le champ est optionnel

Les vues

- Les vues sont des objets de base de données qui vous permettent d'enregistrer une requête particulière sous forme de table. Cela vous permet d'enregistrer les résultats afin de pouvoir les utiliser ultérieurement.

```
CREATE VIEW  vw_clients_agadir AS
SELECT * FROM clients
WHERE  UPPER(adresse)=UPPER("agadir")
```

- **CREATE VIEW** `nom_view` AS requête_select

- **Remarque**

  Les vues dans MySQL sont interrogeables, ce qui signifie que vous pouvez les inclure dans une autre requête, un peu comme une table dans MySQL. Les vues peuvent également être mises à jour car vous pouvez INSÉRER, METTRE À JOUR et SUPPRIMER des lignes dans la table sous-jacente si la vue ne peut pas contenir les éléments suivants : MIN, MAX, SUM, AVG,et COUNT, DISTINCT , GROUP BY, HAVING , UNION , LEFT JOIN ou OUTER JOIN.

Table temporaire

- Une table temporaire est un type spécial de table qui vous permet de stocker un ensemble de résultats temporaires, que vous pouvez réutiliser plusieurs fois au cours d'une même session.
- Une table temporaire est très pratique lorsqu'il est impossible ou coûteux d'interroger des données nécessitant une seule instruction **SELECT** avec les clauses **JOIN**. Dans ce cas, vous pouvez utiliser une table temporaire pour stocker le résultat immédiat et utiliser une autre requête pour le traiter.
- Une table temporaire **MySQL** possède les fonctionnalités spécialisées suivantes :
  - **MySQL** supprime automatiquement la table temporaire lorsque la session se termine ou que la connexion est interrompue. Vous pouvez utiliser l'instruction **DROP TABLE** pour supprimer explicitement une table temporaire lorsque vous ne l'utilisez plus.
  - Une table temporaire n'est disponible et accessible qu'au client qui la crée.
  - Une table temporaire peut avoir le même nom qu'une table normale dans une base de données, dans ce cas cette dernière est masquée par la table temporaire dans les requêtes.

Création d'une table temporaire

- **CREATE TEMPORARY TABLE** nom_table(column_1_definition, column_2_definition, ..., table_constraints);

  Ou bien a partir du résultat d'un select:

- **CREATE TEMPORARY TABLE** temp_nom_table

  SELECT * **FROM** original_table

  LIMIT 0;

Suppression d'une table temporaire

- **DROP TEMPORARY TABLE** nom_table;

PARTIE 1

INSERT

- L'instruction **INSERT INTO** est utilisée pour insérer de nouveaux enregistrements dans une table.
- **INSERT INTO** nom_table (column1, column2, column3, ...)
  VALUES (value1, value2, value3, ...);

```
INSERT INTO clients( nom, prenom, date_naissance, adresse )
Values("alami","said","1980-5-22","casa")

INSERT INTO clients(nom, prenom, date_naissance )
Values("Zaki","Fatima","1986-5-22")
```

UPDATE

- L'instruction **UPDATE** est utilisée pour modifier les enregistrements existants dans une table.
- **UPDATE** nom_table
  SET colonne1 = valeur1, colonne2 = valeur2, ...
  WHERE condition;

```
UPDATE clients
Set nom='Fadani', adresse="Rabat"
WHERE id=2
```

DELETE

- L'instruction **DELETE** est utilisée pour supprimer des enregistrements existants dans une table.
- **DELETE FROM** nom_table **WHERE** condition**;**

```
DELETE FROM clients WHERE id=3;
```

CHAPITRE 1

Maitriser le langage de programmation procédurale sous MySQL

1. Rappeler le langage de manipulation des données (SQL) ;
2. **Présenter le langage de programmation procédural de MySQL ;**
3. Distinguer les différents types des programmes MySQL ;
4. Maitriser les instructions de bases ;
5. Maitriser les structures de contrôle ;
6. Gérer les transactions ;
7. Gérer les exceptions ;
8. Manipuler les curseurs ;
9. Créer les procédures stockées et les fonctions ;
10. Mettre en place les déclencheurs.

PARTIE 1

Introduction

- Les structures de contrôle habituelles d'un langage de programmtion (**IF**, **WHILE**…) ne font pas partie intégrante de la norme **SQL**. Elles apparaissent dans une sous-partie optionnelle de la norme (ISO/IEC 9075-5:1996. Flow-control statements).
- Cette extension de **SQL** permet de faire cohabiter les habituelles structures de contrôle (**IF**,**CASE**,**WHILE**,**REPEAT**,**LOOP**) avec des instructions **SQL** (principalement **SELECT**, **INSERT**, **UPDATE** et **DELETE**).
- Deux directives supplémentaires qui sont toutefois à utiliser avec modération :
  - **LEAVE** : qui sort d'une boucle (ou d'un bloc étiqueté)
  - **ITERATE** : qui force le programme à refaire un tour de boucle depuis le début

Définition

CHAPITRE 1

Maitriser le langage de programmation procédurale sous MySQL

1. Rappeler le langage de manipulation des données (SQL) ;
2. Présenter le langage de programmation procédural de MySQL ;
3. **Distinguer les différents types des programmes MySQL ;**
4. Maitriser les instructions de bases ;
5. Maitriser les structures de contrôle ;
6. Gérer les transactions ;
7. Gérer les exceptions ;
8. Manipuler les curseurs ;
9. Créer les procédures stockées et les fonctions ;
10. Mettre en place les déclencheurs.

PARTIE 1

- Un sous proramme est un bloc d'instructions est composé de :
  - **BIGIN** :pour indique le debut du bloc
  - **DECLARE**(directive optionnelle):déclare une variable, un curseur, une exception, etc. ;
  - **END** : ferme le bloc.
- Un bloc peut être imbriqué dans un autre bloc

```
BEGIN
   SELECT 'Bloc d''instructions principal';
   BEGIN
      SELECT 'Bloc d''instructions 2, imbriqué dans le bloc principal';

      BEGIN
         SELECT 'Bloc d''instructions 3, imbriqué dans le bloc
d''instructions 2';
      END;
   END;

   BEGIN
      SELECT 'Bloc d''instructions 4, imbriqué dans le bloc principal';
   END;
END;
```

Définition

- Les fonctions peuvent être utilisées pour créer une logique de programmation personnalisée pour votre base de données. Ceci est utile dans les situations où vous avez du code répété dans plusieurs zones et que vous souhaitez éviter de copier le code plusieurs fois.
- MySQL possède de nombreuses fonctions intégrées que vous pouvez appeler pour renvoyer des valeurs ou effectuer des tâches sur des données, notamment **CURRENT_DATE**(), **AVG**(), **SUM**(), **ABS**() et **CONCAT**(). Ces fonctions peuvent être utilisées dans des instructions SQL, des vues et des procédures stockées.
- MySQL a également un autre type de fonction, connue sous le nom de Fonction Définie par l'Utilisateur (UDF).

Syntaxe

```
USE nom_bd;
DROP FUNCTION IF EXISTS nom_fonction;
CREATE FUNCTION nom_fonction ([parameter(s)])
        RETURNS  type_retour
        déclaration informative
        Instructions
```

```
USE eshop_app_db;
DROP FUNCTION IF EXISTS uf_nombre_clients_par_ville;
DELIMITER $$
CREATE FUNCTION uf_nombre_clients_par_ville ( ville VARCHAR(50))
        RETURNS INT
        READS SQL DATA
        BEGIN
                DECLARE nombre INT;
                SET nombre = (SELECT COUNT (*)
                        FROM clients
                        WHERE adresse = ville );
                RETURN nombre;
        END $$
DELIMITER ;
```

La déclaration informative peut prendre l'une des valeurs:

- **DETERMINISTIC** : La fonction renverra les mêmes valeurs si les mêmes arguments lui sont fournis, ce qui signifie qu'on connaisse toujours la sortie, compte tenu de l'entrée ;
- **READS SQL DATA** : Spécifie si la fonction lira les données de la base de données mais ne modifiera pas les données;
- **MODIFIES SQL DATA** : Spécifie si la fonction modifiera les données dans la base de données;
- **CONTAINS SQL** : Spécifie si la fonction aura des instructions SQL mais elles ne lisent ni ne modifient les données,

telles que : **SELECT CURRENT_DATE**();

PARTIE 1

Syntaxe

DLIMITER $$

- Le délimiteur par défaut est le point-virgule ;
- Cependant, lorsque nous définissons des fonctions, des procédures stockées et des déclencheurs, nous exécuterons souvent plusieurs instructions.
- Définir un délimiteur différent nous permet d'exécuter toutes les instructions comme une seule unité plutôt qu'individuellement.

Remarques :

- Nous ne pouvons pas renvoyer une table à partir d'une fonction MySQL.
- Une fonction peut renvoyer une chaîne, un entier, un caractère, etc.
- Pour renvoyer une table à partir de MySQL, on utilise une procédure stockée, et non pas une fonction.

Définition

- Les procédures stockées permettent de stocker un ensemble de requêtes SQL, à exécuter en cas de besoin.
- En règle générale, on doit utiliser des procédures stockées lorsqu'une requête ou un ensemble de requêtes doit être répété régulièrement.
- La procédure stockée peut contenir une instruction conditionnelle telle que **IF** ou **CASE** ou les boucles.
- La procédure stockée peut également exécuter une autre procédure stockée ou une fonction qui modularise le code.
- Les procédures stockées sont idéales pour déplacer des tâches de traitement lourdes vers le serveur MySQL.

Cas d'utilisation

- Imaginons que nous travaillons sur une application de vente connectée à une base de données MySQL et que nous devons enregistrer une vente. L'application enregistrerait une vente en procédant comme suit :
  - Détermination du montant total du paiement pour confirmer le paiement
    - Calcul de la valeur de vente (coût des ventes * valeur de l'article)
    - Calcul de la taxe de vente qui s'applique à la vente
  - Soustraction de l'article du tableau d'inventaire
  - Vérification de la valeur de stock minimum de l'article
  - Génération d'un reçu
- Toutes ces tâches peuvent être placées dans une seule **procédure stockée**.

Avantages d'une procédure stockée

Réduire le trafic réseau

- Plusieurs instructions SQL sont encapsulées dans une procédure stockée.
- Lorsque on l'exécute, au lieu d'envoyer plusieurs requêtes, on envoie uniquement le nom et les paramètres de la procédure stockée.

Facile à maintenir

- Les procédures stockées sont réutilisables. On peut implémenter la logique métier dans un SP, et elle peut être utilisée par des applications plusieurs fois, ou par différents modules d'une application.
- Une procédure stockée rend la base de données plus cohérente.
- Si une modification est nécessaire, on doit uniquement apporter une modification à la procédure stockée.

Sécurité

- Les procédures stockées sont plus sécurisées que les requêtes AdHoc.
- L'autorisation peut être accordée à l'utilisateur pour exécuter la procédure stockée sans donner l'autorisation aux tables utilisées dans la procédure stockée.
- La procédure stockée aide à protéger la base de données de SQL Injection.

Syntaxe

```
DELIMITER $$
CREATE PROCEDURE nom_procedure([paramètres])
        BEGIN
                le code SQL
        END $$
        Syntaxe pour exécuter une procédures stockée
call nom_procedure([paramètres]);
```

```
DELIMITER $$
CREATE PROCEDURE sp_client_par_ville(IN ville VARCHAR(50))
        BEGIN
                SELECT *
                FROM clients
                WHERE UPPER(adresse) = UPPER(ville);
        END $$
DELIMITER
- - Appel
call sp_client_par_ville (''agadir'');
```

Syntaxe

Les procédures stockées MySQL ont trois directions dans lesquelles un paramètre peut être défini:

IN

- La valeur est uniquement transmise à la procédure stockée. Elle est utilisée dans le cadre de la procédure. Cela revient à fournir une entrée à la procédure stockée.

OUT

- La valeur est uniquement transmise hors de la procédure stockée ; toutes les variables externes qui ont été affectées à cette position prendront la valeur transmise. Cela revient à renvoyer des valeurs à partir d'une procédure stockée.

INOUT

- Une variable et sa valeur (ExtVal) sont transmises à la procédure stockée (IntVal) et peuvent y être modifiées. Lorsque la procédure stockée est terminée, la valeur externe (ExtVal) sera égale à la valeur modifiée (IntVal)

Mode de passage de paramètres IN

IN est le mode par défaut. Lorsque vous définissez un paramètre IN dans une procédure stockée, le programme appelant doit passer un argument à la procédure stockée. Cette argument est passé par valeur.

```
DELIMITER $$
CREATE PROCEDURE sp_client_par_ville(IN ville VARCHAR(50))
    BEGIN
            SELECT *
            FROM clients
            WHERE UPPER(adresse) = UPPER(ville);
    END $$
DELIMITER ;
- - Appel
call sp_client_par_ville (''agadir'');
```

Mode de passage de paramètres OUT

- Il s'agit d'un passage de paramètre en sortie (par référence). On passe a la procédure stockée une variable de session dont la valeur peut être modifiée à l'intérieur de la procédure stockée.
- Notons que la procédure stockée ne peut pas accéder à la valeur initiale du paramètre OUT lorsqu'elle démarre.

```
DELIMITER $$
CREATE PROCEDURE sp_client_par_ville(IN ville VARCHAR(50),
                                     OUT nombre_clients INT)
    BEGIN
        SELECT COUNT(*) INTO nombre_clients
        FROM clients
        WHERE UPPER(adresse) = UPPER(ville);
    END $$
DELIMITER ;
- - Appel
CALL sp_client_par_ville (''agadir'', @total); - - @total est une variable de session
SELECT @total;
```

PARTIE 1

Mode de passage de paramètres INOUT

Un paramètre INOUT est une combinaison de paramètres IN et OUT. Cela signifie que le programme appelant peut transmettre l'argument et que la procédure stockée peut modifier le paramètre INOUT et retransmettre la nouvelle valeur au programme appelant.

```
DELIMITER $$
CREATE PROCEDURE SetCounter (INOUT counter INT,
                             IN inc INT
                             )
    BEGIN
            SET counter = counter + inc ;
    END $$
DELIMITER ;
- - Appel
SET @counter = 1;
CALL SetCounter (@counter', 1);  - - 2
CALL SetCounter (@counter', 1);  - - 3
CALL SetCounter (@counter', 5);  - - 8
SELECT @counter;  - - 8
```

Suppression d'une procédure stockée

- **DROP PROCEDURE** [**IF EXISTS**] nom_procedure_stockée ;
- Pour modifier une procédure stockée on doit supprimer et recréer la procédure stockée à l'aide des instructions **DROP PROCEDURE** et **CREATE PROCEDURE**.
- MySQL Workbench fournit un bon outil qui vous permet de modifier rapidement une procédure stockée.

```
DROP PROCEDURE IF EXISTS SetCounter ;
```

Liste des procédures stockées

SHOW PROCEDURE STATUS [LIKE 'pattern' | **WHERE** search_condition**]**

SHOW PROCEDURE STATUS ; - - affiche toutes les procédures stockées

SHOW PROCEDURE STATUS WHERE db = 'eshop_app_db' **; - -** affiche toutes les procédures stockées de la base de données eshop_app_db

SHOW PROCEDURE STATUS LIKE '%clients%' **; - -** affiche toutes les procédures stockées dont le nom respecte le pattern

Liste des procédures stockées en utilisant la table routine

```
SELECT
        routine_name
FROM
         information_schema.routines
WHERE
        routine_type = 'PROCEDURE'
        AND routine_schema = 'nom_base_donnees';
```

```
SELECT
        routine_name
FROM
         information_schema.routines
WHERE
        routine_type = 'PROCEDURE'
        AND routine_schema = 'eshop_app_db';
```

Result Grid | Filter Rows: | Export: | Wrap

ROUTINE_NAME
sp_client_par_ville
sp_client_par_ville_out

CHAPITRE 1

Maitriser le langage de programmation procédurale sous MySQL

1. Rappeler le langage de manipulation des données (SQL) ;
2. Présenter le langage de programmation procédural de MySQL ;
3. Distinguer les différents types des programmes MySQL ;
4. **Maitriser les instructions de bases ;**
5. Maitriser les structures de contrôle ;
6. Gérer les transactions ;
7. Gérer les exceptions ;
8. Manipuler les curseurs ;
9. Créer les procédures stockées et les fonctions ;
10. Mettre en place les déclencheurs.

Déclaration des variables scalaires

- **DECLARE** nomVariable1[,nomVariable2...] typeMySQL [**DEFAULT** expression];
- Lors des conflits potentiels de noms (variables ou colonnes) dans des instructions **SQL** (principalement **INSERT**, **UPDATE**, **DELETE** et **SELECT**), le nom de la variable est prioritairement interprété au détriment de la colonne de la table (de même nom)

```
DECLARE v_dateNaissance DATE;
DECLARE v_trouve BOOLEAN DEFAULT TRUE;
DECLARE v_Dans2jours DATE  DEFAULT
ADDDATE(SYSDATE(),2);
DECLARE i, j, k INT;
```

Déclaration des variables de session (externes)

- Les variables dites de session (user-defined variables) sont déclarées à l'aide du symbole « @ ». C'est une variable définie par un client qui n'est pas visible par les autres clients. En d'autres termes, une variable définie par l'utilisateur qui est spécifique à la session.
- **SET** @nom_variable := valeur Ou **SELECT** @nom_variable := valeur

```
SELECT
        @max_prix := MAX (pu) -- initialiser la variable de session @max_prix par le maximum des prix
FROM
        Produits;

SELECT
        id, designation, qtstock, pu
FROM
        produit
WHERE
        pu = @max_prix; -- utiliser la variable @max_prix pour lister les produits don le prix est le max
```

PARTIE 1

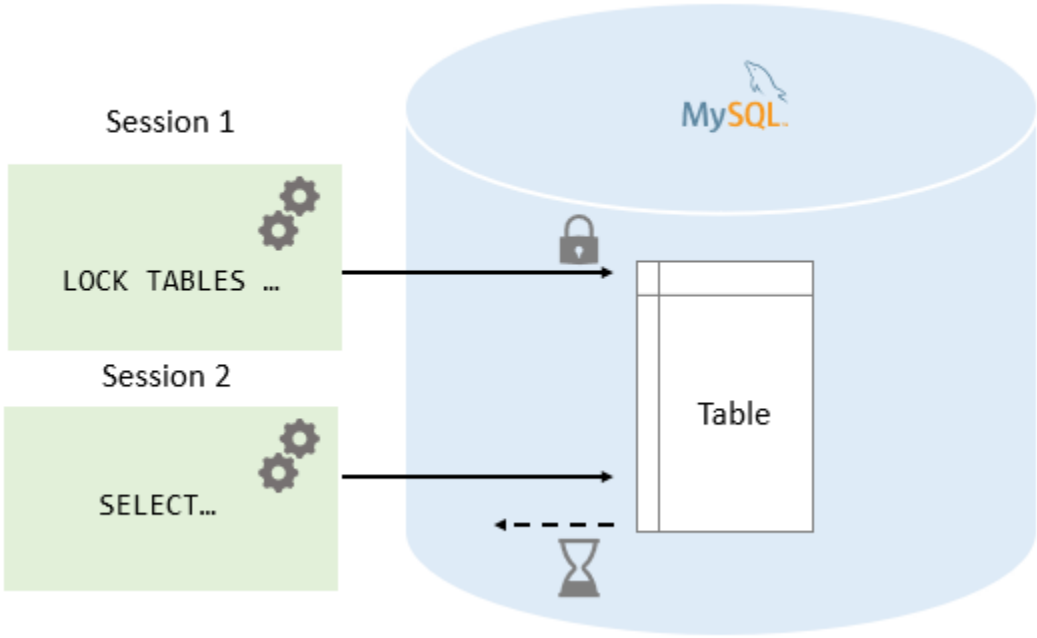
Portée des objets

- La portée d'un objet (variable, curseur ou exception) est limitée au bloc dans lequel il est déclaré. Un objet déclaré dans un bloc est accessible dans les sous-blocs. En revanche, un objet déclaré dans un sous-bloc n'est pas visible du bloc conteur.

PARTIE 1

Définition

- Un verrou est un drapeau associé à une table. MySQL perme à une session client d'acquérir explicitement un verrou de table pour empêcher d'autres sessions d'accéder à la même table pendant une période spécifique.
- Une session client peut acquérir ou libérer des verrous de table uniquement pour elle-même.
- Une session client ne peut pas acquérir ou libérer des verrou de table pour d'autres sessions client.



Syntaxe

- Pour verrouiller les tables :

  LOCK TABLES nom_table [**READ** | **WRITE**]

- Pour déverrouiller les tables :

  UNLOCK TABLES;

CHAPITRE 1

Maitriser le langage de programmation procédurale sous MySQL

1. Rappeler le langage de manipulation des données (SQL) ;
2. Présenter le langage de programmation procédural de MySQL ;
3. Distinguer les différents types des programmes MySQL ;
4. Maitriser les instructions de bases ;
5. **Maitriser les structures de contrôle ;**
6. Gérer les transactions ;
7. Gérer les exceptions ;
8. Manipuler les curseurs ;
9. Créer les procédures stockées et les fonctions ;
10. Mettre en place les déclencheurs.

IF

```
IF condition THEN
    instructions;
END IF;
```

IF-ELSE

```
IF condition THEN
    instructions;
ELSE
    instructions;
END IF;
```

```
IF SUBSTR (v_telephone,1,2) = '06' THEN
    SELECT ''C'est un portable' ;
ELSE
    SELECT ''C'est un fixe … '' ;
END IF ;
```

PARTIE 1

IF-ELSEIF-ELSE

```
IF condition1 THEN
        instructions;
ELSEIF condition2 THEN
        Instructions2;
ELSE
        instructions3;
END IF;
```

```
CREATE FUNCTION IncomeLevel (monthly_value INT)
RETURNS VARCHAR(20)
BEGIN
        DECLARE income_level VARCHAR(20);
        IF monthly_value <= 4000 THEN
                SET income_level = 'Low Income' ;
        ELSEIF monthly_value >  4000  AND monthly_value <= 7000 THEN
                SET income_level = 'Avg Income' ;
        ELSE
                SET income_level = 'High Income' ;
        END IF ;
        RETURN income_level ;
END ; //
DELIMITER ;
```

PARTIE 1

Case avec sélecteur

```
CASE variable_selecteur
        WHEN expr1 THEN instructions1;
        WHEN expr2 THEN instructions2;
        …
        WHEN exprN THEN instructionsN;
        [ELSE instructionsN+1;]
END CASE
```

```
CREATE PROCEDURE GetCustomerShipping (IN pCustomerNumber INT,
OUT pShipping VARCHAR(50))
BEGIN
        DECLARE customerCountry VARCHAR(100);
        SELECT country INTO customerCountry FROM customers
        WHERE customerNumber = pCustomerNumber ;
        CASE customerCountry
                WHEN 'USA' THEN
                        SET pShipping = '2-day Shipping' ;
                WHEN 'Canada' THEN
                        SET pShipping = '3-day Shipping' ;
                ELSE
                        SET pShipping = '5-day Shipping' ;
        END CASE ;
END$$
DELIMITER ;
```

Case sans sélecteur

```
CASE
        WHEN condition1 THEN instructions1;
        WHEN condition2 THEN instructions2;
        …
        WHEN condition¨N THEN instructionsN;
        [ELSE instructionsN+1;]
END CASE
```

```
SELECT OrderID, Quantity,
CASE
        WHEN Quantity > 30 THEN ''The quantity is greater than 30''
        WHEN Quantity=30 THEN ''The quantity is 30''
        ELSE ''The quantity is under 30''
END
FROM OrderDetails;
```

```
DECLARE v_mention CHAR(2) ;
DECLARE v_note DECIMAL(4,2) DEFAULT 9.8 ;
CASE
        WHEN v_note >= 16 THEN SET v_mention := 'TB';
        WHEN v_note >= 14 THEN SET v_mention := 'B';
        WHEN v_note >= 12 THEN SET v_mention := 'AB';
        WHEN v_note >= 10 THEN SET v_mention := 'P';
        ELSE  SET v_mention := 'R';
END CASE ;
```

PARTIE 1

LOOP

L'instruction LOOP vous permet d'exécuter une ou plusieurs instructions à plusieurs reprises.

[labele_debut:] **LOOP**

statement_list

END LOOP [label_fin]

```
CREATE TABLE test (VALUE INT NULL DEFAULT NULL) ;
DELIMITER $$
CREATE PROCEDURE ps_nombres()
BEGIN
    DECLARE a INT DEFAULT 1 ;
    simple_loop : LOOP
        INSERT INTO test VALUES (a) ;
        SET a = a + 1 ;
        IF a = 11 THEN
            LEAVE simple_loop ;
        END IF ;
    END LOOP simple_loop;
END $$
CALL ps_nombres();
SELECT value FROM test;
```

LOOP

- L'instruction **LEAVE** sort immédiatement de la boucle. Cela fonctionne comme l'instruction break dans d'autres langages de programmation comme PHP, C/C++ et Java.
- L'instruction **ITERATE** est utilisée pour ignorer l'itération de la boucle en cours et démarrer une nouvelle itération. **L'ITERATE** est similaire à l'instruction continue en PHP, C/C++ et Java.

```
DROP PROCEDURE LoopDemo ;
DELIMITER $$
CREATE PROCEDURE LoopDemo()
BEGIN
        DECLARE x INT ;
        DECLARE str VARCHAR (255) ;
        SET str = '' ;
        loop_label : LOOP
                        IF x > 10 THEN
                                        LEAVE loop_label;
                        END IF ;
                        SET x = x + 1 ;
                        IF (x mod 2) THEN
                                        ITERATE loop_label ;
                        ELSE
                                        SET str = CONCAT (str,x,',') ;
                        END IF ;
        END LOOP ;
        SELECT str ;
END$$
DELIMITER ;
```

WHILE

- La boucle **WHILE** est une instruction de boucle qui exécute un bloc de code de manière répétée tant qu'une condition est vraie.

- Pour l'utilisation de la boucle WHILE, on utilise la syntaxe :

[labele_debut:] **WHILE** condition **DO**
liste des instructions
END WHILE [labele_fin]

```
CREATE PROCEDURE dowhile () ;
BEGIN
    DECLARE v1 INT DEFAULT 5 ;
    WHILE v1 > 0 DO
        . . .
        SET  v1 = v1 – 1 ;
    END WHILE ;
END ;
```

REPEAT

- La liste d'instructions dans une instruction **REPEAT** est répétée jusqu'à (**UNTIL)** ce que l'expression condition soit vraie.
- Ainsi, un **REPEAT** entre toujours dans la boucle au moins une fois.
- **liste_instructions** se compose d'une ou plusieurs instructions, chacune terminée par un délimiteur d'instruction point-virgule (;).

```
DELIMITER //
CREATE PROCEDURE dorepeat (p1 INT) ;
BEGIN
        SET @x = 0 ;
        REPEAT
                SET @x = @x + 1 ;
        UNTIL @x > p1
END ;
```

CHAPITRE 1

Maitriser le langage de programmation procédurale sous MySQL

1. Rappeler le langage de manipulation des données (SQL) ;
2. Présenter le langage de programmation procédural de MySQL ;
3. Distinguer les différents types des programmes MySQL ;
4. Maitriser les instructions de bases ;
5. Maitriser les structures de contrôle ;
6. **Gérer les transactions ;**
7. Gérer les exceptions ;
8. Manipuler les curseurs ;
9. Créer les procédures stockées et les fonctions ;
10. Mettre en place les déclencheurs.

- Par défaut, le client MySQL est configuré pour utiliser la validation automatique.
- On n'a donc pas à le faire. Si on souhaite avoir la possibilité d'annuler l'instruction INSERT, on doit utiliser une transaction.
- Cela peut être fait avec une instruction BEGIN ou une instruction START TRANSACTION.
- Une fois qu'on a exécuté une ou plusieurs instructions pour modifier les données, on doit utiliser COMMIT ou ROLLBACK.

```
BEGIN ; - - Ceci indique le début de la transaction
        INSERT INTO mytable VALUES (1, 'foo','bar','bas') ;
        SELECT * FROM mytable ;
COMMIT; - - Utiliser ROLLBACK au lieu de COMMIT si on veut annuler l'operation
```

Remarque : Pour désactiver le commit automatique utiliser :

```
SET autocommit = 0;

Ou

SET autocommit = OFF
```

PARTIE 1

Contrôle des transactions

- Les commandes suivantes sont utilisées pour contrôler les transactions:

COMMIT

- Pour enregistrer les modifications

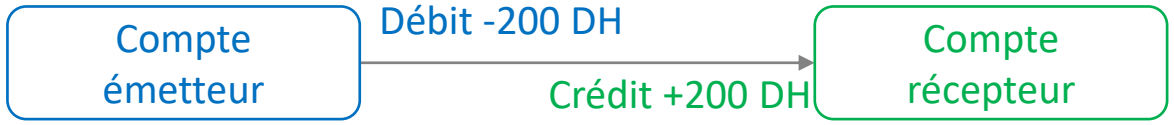
ROLLBACK

- Pour annuler les modifications

- Ces commandes ne sont utilisées qu'avec les commandes DML telles que - **INSERT**, **UPDATE** et **DELETE** uniquement.

Exemple

- Le meilleur exemple pour comprendre une TRANSACTION MySQL est un transfert d'argent entre 2 comptes d'une même banque.



- Supposons que le débit ait été un succès, mais que le crédit n'ait pas eu lieu, peut-être à cause de problèmes de base de données. Dans ce cas, la base de données serait dans un état incohérent.
- Idéalement, on veut que cette transaction (à la fois de crédit et de débit) se produise, sinon aucune d'entre elles ne se produira. c'est-à-dire que dans ce cas, s'il s'agissait d'une transaction, un échec de crédit aurait entraîné une annulation de l'opération de débit et il n'y aurait eu aucun changement dans l'état de la base de données.

```
START TRANSACTION ;
    UPDATE compte SET solde = solde – 200 WHERE accountno = 'ACC1' ;  - - Débiter le compte ACC1
    UPDATE compte SET solde = solde + 200 WHERE accountno = 'ACC2' ;  - - Crediter le compte ACC2
COMMIT ; - - Valider
```

PARTIE 1

PARTIE 1

Définition

- Les événements MySQL sont des tâches qui s'exécutent selon un calendrier spécifié. Par conséquent, les événements MySQL sont parfois appelés événements planifiés.
- MySQL utilise un thread spécial **appelé thread du planificateur d'événements** pour exécuter tous les événements planifiés.
- On peut afficher l'état du thread du planificateur d'événements en exécutant la commande **SHOW PROCESSLIST** :

```
CREATE EVENT test_event_03
ON SCHEDULE EVERY 1 MINUTE
STARTS CURRENT_TIMESTAMP
ENDS CURRENT_TIMESTAMP + INTERVAL 1 HOUR
DO
        INSERT INTO messages (message, created_at
        VALUES ('Test MySQL recurring Event', NOW());
```

CHAPITRE 1

Maitriser le langage de programmation procédurale sous MySQL

1. Rappeler le langage de manipulation des données (SQL) ;
2. Présenter le langage de programmation procédural de MySQL ;
3. Distinguer les différents types des programmes MySQL ;
4. Maitriser les instructions de bases ;
5. Maitriser les structures de contrôle ;
6. Gérer les transactions ;
7. **Gérer les exceptions ;**
8. Manipuler les curseurs ;
9. Créer les procédures stockées et les fonctions ;
10. Mettre en place les déclencheurs.

Introduction

- Afin d'éviter qu'un programme ne s'arrête dès la première erreur suite à une instruction SQL, il est indispensable de prévoir les cas potentiels d'erreurs et d'associer à chacun de ces cas la programmation d'une exception (handler dans le vocabulaire de MySQL).
- Les exceptions peuvent être gérées dans un sous-programme (fonction ou procédure cataloguée) ou un déclencheur.
- Une exception MySQL correspond à une condition d'erreur et peut être associée à un identificateur (exception nommée).
- Une exception est détectée (aussi dite « levée ») si elle est prévue dans un handler au cours de l'exécution d'un bloc (entre **BEGIN** et **END**).
- Une fois levée, elle fait continuer (ou sortir du bloc) le programme après avoir réalisé une ou plusieurs instructions que le programmeur aura explicitement spécifiées.
- Deux mécanismes qui peuvent être mis en œuvre pour gérer une exception en Mysql : **CONTINUE** et **EXIT**.

PARTIE 1

Syntaxe

```
DECLARE handler_action HANDLER
        FOR condition_value [ , condition_value] …
        statement
Handler_action : {
        CONTINUE
        | EXIT
        | UNDO
}
Condition_value : {
        mysql_error_code
        | SQLSTATE [VALUE] sqlstate_value
        | condition_name
        | SQLWARNING
        | NOT FOUND
        | SQL EXCEPTION
}
```

- **CONTINUE** : (appelée handler) force à poursuivre l'exécution de programme lorsqu'il se passe un événement prévu dans la clause **FOR**
- **EXIT** fait sortir l'exécution du bloc courant entre **BEGIN** et **END**
- **SQLSTATE** : code d'erreur qui permet de couvrir toutes les erreurs d'un état donné. (https://dev.mysql.com/doc/connector-j/5.1/en/connector-j-reference-error-sqlstates.html)
- **nomException** : s'applique à la gestion des exceptions nommées (étudiées plus loin).
- **SQLWARNING** :permet de couvrir toutes les erreurs d'état **SQLSTATE** débutant par 01.
- **NOT FOUND** : permet de couvrir toutes les erreurs d'état **SQLSTATE** débutant par 02 (relatives à la gestion des curseurs).
- **SQLEXCEPTION** : gère toutes les erreurs qui ne sont ni gérées par **SQLWARNING** ni par **NOT FOUND**
- **statement MySQL** : une ou plusieurs instructions du langage de MySQL (bloc, appel possibles par **CALL** d'une fonction ou d'une procédure cataloguée).

Exceptions avec EXIT

- Essayons d'insérer un client sans spécifier le champs obligatoire (NOT NULL) nom. Une exception et déclenchée ayant le code 1040.
- Gérons nous cette exception avec le handler Exit

```
DELEMITER//;
CREATE PROCEDURE ps_ajouter_client_Exception1(IN p_nom VARCHAR (200),
                                              IN p_prenom VARCHAR(200),
                                              IN p_adresse VARCHAR(200))
BEGIN
    DECLARE flagNOTNULL BOOLEAN DEFAULT 0 ;
    BEGIN - - début bloc de déclaration des deux exceptions
        DECLARE EXIT HANDLER FOR 1048
        SET flagNOTNULL := -1;
        INSERT INTO clients (nom, prenom, adresse) VALUES (p_nom, p_prenom, p_adresse);
        SELECT 'le client est ajouté avec succès';
    END; - - fin bloc de déclaration des deux exceptions
IF flagNOTNULL THEN
    SELECT CONCAT ('Le chmp nom doit être non null') AS 'Resultat ps_ajouter_client_Exception1';
END IF;
END;//
CALL ps_ajouter_client_Exception1(null, 'Saloua', 'Safi');
```

```
insert into clients(nom,prenom,adresse) values(null,'saad','Safi')
```

Output

Action Output

#	Time	Action	Message
1	11:46:15	insert into clients(nom,prenom,adresse) values(...	Error Code: 1048. Column 'nom' cannot be null

```
mysql> CALL ps_ajouter_client_Exception1('Saadi','Saloua','Safi');
+---------------------------------+
| Le client est ajoute avec succes |
+---------------------------------+
| Le client est ajoute avec succes |
+---------------------------------+
```

```
mysql> CALL ps_ajouter_client_Exception1(null,'Saloua','Safi');
+--------------------------------------+
| Resultat ps_ajouter_client_Exception1 |
+--------------------------------------+
| Le champ nom doit etre non null       |
+--------------------------------------+
```

PARTIE 1

Exceptions avec EXIT

- L' exemple suivant décrit une procédure qui gère une erreur : '*aucun client n'est associé à ID passé en paramètre* (*NOT FOUND)*'.
- La procédure ne se termine pas correctement si plusieurs lignes sont retournées (*erreur Result consisted of more than one row*).

```
CREATE PROCEDURE ps_exc_not_found_Exemple (IN p_nom VARCHAR (200))
BEGIN
        DECLARE flagNOTFOUND BOOLEAN DEFAULT 0 ;
        DECLARE var1 VARCHAR(20) ;
        BEGIN
                DECLARE EXIT HANDLER FOR NOT FOUND
                SET flagNOTFOUND := -1;
                SELECT nom INTO var1 FROM clients
                WHERE nom = p_nom ;
                SELECT CONCAT ('Le seul client avec le nom ', p_nom, ' est ', var1)
                        AS 'Resultat ps_exc_not_found_Exemple1';
        END
        IF flagNOTFOUND THEN
                SELECT CONCAT ('Il n`'y a pas de client aven le nom ', p_nom)
                        AS 'Resultat ps_exc_not_found_Exemple1' ;
        END IF;
END;
CALL ps_exc_not_found_Exemple1('Dalaj');
```

PARTIE 1

Exception FOR SQLEXCEPTION

- L'exemple suivant décrit une procédure qui gère une erreur non spécifique .
- **Remarque**: MySQL ne permet pas, pour l'instant, de récupérer dynamiquement, au sein d'un sous-programme, le code et le message de l'erreur associée à une exception levée suite à une instruction SQL, et qui n'a pas été prévue dans un handler

```
DELIMITER // ;
CREATE PROCEDURE autreErreur ()
BEGIN
        SELECT 'Une autre erreur est survenue' ;
END ; //
DELIMITER // ;

CREATE PROCEDURE handlerdemoSQLEXCEPTION ()
BEGIN
        DECLARE EXIT HANDLER FOR SQLEXCEPTION CALL autreErreur();
        INSERT INTO table_inexistante VALUES (1);
        SET @x = 99;
END;//
CALL handlerdemoSQLEXCEPTION();
SELECT @x;
```

Exceptions avec Continue

- L'exemple suivant décrit une procédure qui gère une erreur : Sqlstate : ER_DUP_KEY '23000'
- Malgré qu'une exception de duplication de clé primaire ER_DUP_KEY est causée par l'instruction insert, l'exécution des instructions après insert continue et la variable de session @x est initialisée.

```
CREATE TABLE test (code INT, PRIMARY KEY (code));
DELIMITER // ;
CREATE PROCEDURE handlerdemo ()
        BEGIN
        DECLARE CONTINUE HANDLER FOR SQLSTATE '23000'
        SET @x2 = 1;
        SET @x = 1;
        INSERT INTO test VALUES (1);
        SET @x = 2;
        INSERT INTO test VALUES (1);
        SET @x = 3;
END;//
CALL handlerdemo();
SELECT @x;
```

Exceptions nommées (condition)

- Pour intercepter une erreur MySQL et lui attribuer au passage un identificateur, il faut utiliser la clause DECLARE CONDITION.
- Pour la déclaration, on utilise la syntaxe suivante :

DECLARE nomException **CONDITION FOR**
{**SQLSTATE** [**VALUE**] 'valeur_sqlstate' | code_erreur_mysql}

PARTIE 1

L'instruction SIGNAL

- On utilise l'instruction **SIGNAL** pour renvoyer une condition d'erreur ou d'avertissement à l'appelant à partir d'un sous programme, par exemple une procédure stockée, une fonction stockée, un déclencheur ou un événement. L'instruction **SIGNAL** permet de contrôler les informations à renvoyer, telles que la valeur et le message **SQLSTATE**.
- Pour l'utilisation de l'instruction, on utilise la syntaxe suivante :

  SIGNAL SQLSTATE | nom_exception_nomee;
  SET info_1 = valeur_1,
  info_2= valeur_2, etc;
- Info pour prendre :
  MESSAGE_TEXT, **MYSQL_ERRNO** , **SCHEMA_NAME**, nom_table, **COLUMN_NAME**,

```
CREATE PROCEDURE ajouter_commande (IN commandeNo,
                                   IN produitCode varchar(45),
                                   IN qte int,
                                   IN prix double,
                                   IN ligneNO int )
BEGIN
    DECLARE C INT ;
    SELECT COUNT (commandeNumber) INTO C
    FROM commandes
    WHERE commandeNumber = commandeNo;
    - - teste si le numéro de la commande existe
    IF (C != 1) THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Commande
            introuvable';
    END IF ;
    - - suite de code a exécuter si pas d'erreur
    - - …
END
```

L'instruction RESIGNAL

- l'instruction **RESIGNAL** est similaire à l'instruction **SIGNAL** en termes de fonctionnalité et de syntaxe, sauf que :
  - On doit utiliser l'instruction **RESIGNAL** dans un gestionnaire d'erreurs ou d'avertissements, sinon on obtient un message d'erreur « *RESIGNAL lorsque le gestionnaire n'est pas actif* ». Notons qu'on peut utiliser l'instruction **SIGNAL** n'importe où dans une procédure stockée.
  - On peut omettre tous les attributs de l'instruction **RESIGNAL**, même la valeur **SQLSTATE**.

```
DELIMITER $$
CREATE PROCEDURE Divide (IN numerator INT,
                         IN denominator INT,
                         OUT result double)
BEGIN
        DECLARE division_by_zero CONDISTION FOR SQLSTATE '22012' ;
        DECLARE CONTINUE HANDLER FOR division_by_zero
        RESIGNAL SET MESSAGE_TEXT = 'Division by zero / Denominator cannot be
zero';
        - -
        IF denominator = 0 THEN
                SIGNAL division_by_zero;
        ELSE
                SET result := numerator / denominator;
        END IF ;
END
```

PARTIE 1

CHAPITRE 1

Maitriser le langage de programmation procédurale sous MySQL

1. Rappeler le langage de manipulation des données (SQL) ;
2. Présenter le langage de programmation procédural de MySQL ;
3. Distinguer les différents types des programmes MySQL ;
4. Maitriser les instructions de bases ;
5. Maitriser les structures de contrôle ;
6. Gérer les transactions ;
7. Gérer les exceptions ;
8. **Manipuler les curseurs ;**
9. Créer les procédures stockées et les fonctions ;
10. Mettre en place les déclencheurs.

Définition

- Un curseur est une zone mémoire qui est générée côté serveur (mise en cache) et qui permet de traiter individuellement chaque ligne renvoyée par un SELECT.
- Un sous-programme peut travailler avec plusieurs curseurs en même temps. Un curseur, durant son existence (de l'ouverture à la fermeture), contient en permanence l'adresse de la ligne courante.
- Tout curseur MySQL dispose des propriétés suivantes :

Read-only (lecture seule)

- Aucune modification dans la base n'est possible à travers ce dernier (sauf si on ajoute la clause FOR UPDATE) ;

Non-scrollable (non navigable)

- Une fois ouvert, le curseur est parcouru du début à la fin sans pouvoir revenir à l'enregistrement précédent ;

Asensitive (insensible)

- Toute mise à jour opérée dans la base de données n'est pas répercutée dans le curseur une fois ouvert (utilise une copie temporaire des données et ne pointe pas sur les données réelles.

Syntaxe

Pour l'utilisation des curseurs on suit les étapes suivantes :

1. Déclaration du curseur avant la déclaration des variables
2. Utilisation de l'instruction OPEN pour initialiser le jeu de résultats pour le curseur
3. Utilisation de l'instruction FETCH pour récupérer la ligne suivante pointée par le curseur et déplacer le curseur vers la ligne suivante dans le jeu de résultats.
4. 4-Fermeture du curseur.

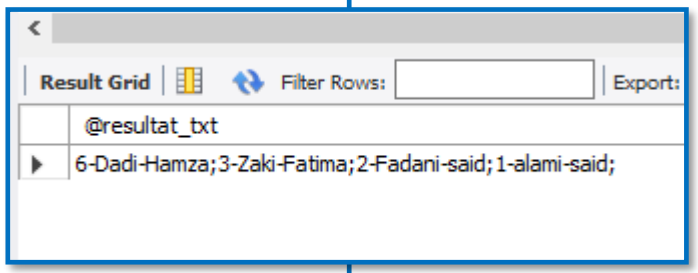
```
DECLARE nom_curseur CURSOR FOR  instruction_SELECT
OPEN nom_curseur;
FETCH nom_curseur INTO  liste_variables;
CLOSE nom_curseur;
```

Exemple

- Ce curseur permet de construire une chaine contenant les emails des employés.
- **Remarques** :
  - Un curseur doit toujours être associé à une instruction SELECT.
  - Lorsque on travaille avec le curseur MySQL, on doit également déclarer un gestionnaire NOT FOUND pour gérer la situation où le curseur ne trouve aucune ligne.

```
DELIMITER $$
CREATE PROCEDURE lister_clients (INOUT resultat_txt VARCHAR(4000))
BEGIN
        DECLARE finished INTEGER DEFAULT 0 ;
        DECLARE v_id INT ;
        DECLARE v_nom VARCHAR (100) ;
        DECLARE v_prenom VARCHAR (100) ;
        DECLARE info VARVHAR (400) DEFAULT ‘’’’ ;
        DECLARE cur_info_client CURSOR FOR SELECT id,nom,prenom FROM clients;
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;
        OPEN cur_info_client ;
        boucle_parcours_clients : LOOP
                FETCH cur_info_client INTO v_id, v_nomn v_prenom ;
                IF finished = 1 THEN
                        LEAVE boucle_parcours_clients ;
                END IF ;
                SET info = CONCAT (v_id, ’’-’’,v_nom,’’-’,v_prenom);
                SET resultat_txt = CONCAT (info, ’’;’’,resultat_txt) ;
        END LOOP boucle_parcours_clients ;
        CLOSE cur_info_client;
END $$
```

```
- - Appel de la ps
SET @resultat_txt = ‘’’’;
CALL lister_clients(@resltat_txt) ;
SELECT @resultat_txt ;
```



PARTIE 1

Curseur pour modification

- Si on veut verrouiller les lignes d'une table interrogée par un curseur dans le but de mettre à jour la table, sans qu'un autre utilisateur ne la modifie en même temps, il faut utiliser la clause **FOR UPDATE**. Elle s'emploie lors de la déclaration du curseur et verrouille les lignes concernées dès l'ouverture du curseur.
- Les verrous sont libérés à la fin de la transaction.

```
DELIMITER $$
CREATE PROCEDURE lister_clients_avec_MAJ_adresse ( INOUT resultat_txt VARCHAR (4000))
BEGIN
        DECLARE finished INTEGER DEFAULT 0 ;
        DECLARE v_id INT ;
        DECLARE v_nom VARCHAR(100) ;
        DECLARE v_prenom VARCHAR(100) ;
        DECLARE v_adresse VARCHAR(100) ;
        DECLARE info VARCHAR(400) DEFAULT '''' ;
        DECLARE cur_info_client CURSOR FOR SELECT id,nom,prenom,adresse
        FROM clients FOR UPDATE ;
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET set finished = 1 ;
        OPEN cur_info_client ;
```

```
        boucle _parcours_clients : LOOP
                FETCH cur_info_client INTO v_id,v_nom,v_prenom,v_adresse;
                IF finished = 1 THEN
                        LEAVE boucle _parcours_clients ;
                END IF;
                IF UPPER(v_adresse)='AGADIR' THEN
                        UPDATE clients
                        SET adresse = 'GRANDE AGADAIR'
                        WHERE id = v_id;
                END IF;
                SET info = CONCAT (v_id, ''-'',v_nom,''-',v_prenom);
                SET resultat_txt = CONCAT (info, '';'',resultat_txt) ;
        END LOOP boucle_parcours_clients ;
        CLOSE cur_info_client;
END $$
DELIMITER ;
```

PARTIE 1

Restrictions d'utilisation

Il n'est pas possible de déclarer un curseur FOR UPDATE en utilisant dans la requête les directives :

- DISTINCT ;
- GROUP BY;
- Un opérateur ensembliste ;
- Une fonction d'agrégat.

CHAPITRE 1

Maitriser le langage de programmation procédurale sous MySQL

1. Rappeler le langage de manipulation des données (SQL) ;
2. Présenter le langage de programmation procédural de MySQL ;
3. Distinguer les différents types des programmes MySQL ;
4. Maitriser les instructions de bases ;
5. Maitriser les structures de contrôle ;
6. Gérer les transactions ;
7. Gérer les exceptions ;
8. Manipuler les curseurs ;
9. **Créer les procédures stockées et les fonctions ;**
10. Mettre en place les déclencheurs.

CHAPITRE 1

Maitriser le langage de programmation procédurale sous MySQL

1. Rappeler le langage de manipulation des données (SQL) ;
2. Présenter le langage de programmation procédural de MySQL ;
3. Distinguer les différents types des programmes MySQL ;
4. Maitriser les instructions de bases ;
5. Maitriser les structures de contrôle ;
6. Gérer les transactions ;
7. Gérer les exceptions ;
8. Manipuler les curseurs ;
9. Créer les procédures stockées et les fonctions ;
10. **Mettre en place les déclencheurs.**

Définition

- Un déclencheur est une procédure stockée, dans la base de données, qui s'appelle automatiquement à chaque fois qu'un événement spécial se produit sur la table à laquelle le déclencheur est attaché.
- Un déclencheur peut être utilisé pour valider les données, enregistrer les anciennes et les nouvelles valeurs dans une table d'audit (log) ou s'assurer que les règles métier sont respectées.
- Par exemple, un déclencheur peut être invoqué lorsqu'une ligne est insérée dans une table spécifique ou lorsque certaines colonnes de la table sont mises à jour.

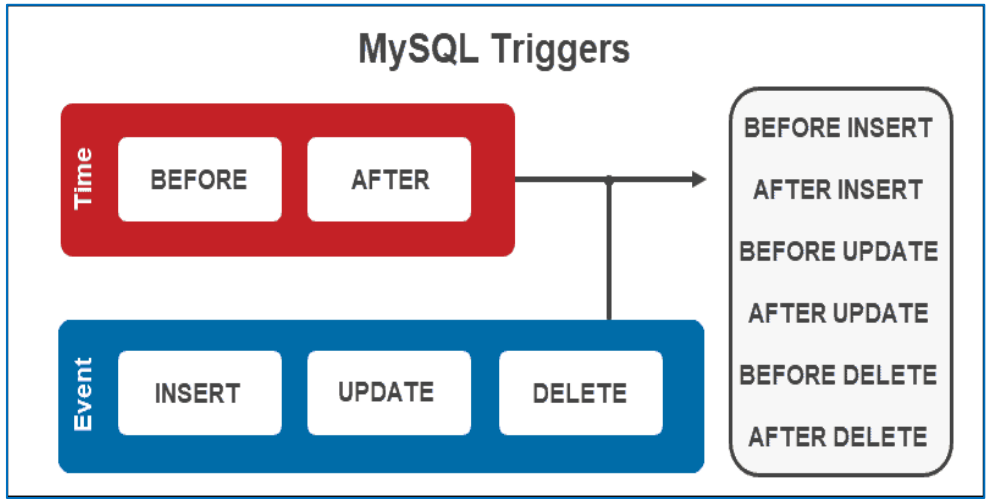
Syntaxe

```
CREATE TRIGGER [nom_declencheur]
[before | after]
{insert | update | delete}
ON [nom_table]
[for each row]
[corps_declencheur]
```

- [before | after] ([avant | apres]) : Ceci spécifie quand le déclencheur sera exécuté.
- {insert | update | delete} ({insérer | mise à jour | supprimer}) : ceci spécifie l'opération DML.
- ON [nom_table] : Ceci spécifie le nom de la table associée au déclencheur.
- [for each row] : Ceci spécifie un déclencheur au niveau de la ligne, c'est-à-dire que le déclencheur sera exécuté pour chaque ligne affectée.
- [corps_declencheur] : Ceci fournit les opérations à effectuer lorsque le déclencheur est déclenché.

Types

- Les déclencheurs **BEFORE** exécutent l'action du déclencheur avant l'exécution de l'instruction de déclenchement(ÉVÉNEMENT).
- Les déclencheurs **AFTER** exécutent l'action du déclencheur après l'exécution de l'instruction de déclenchement
- Il y a trois ÉVÉNEMENTS possibles auxquels un déclencheur peut être assigné : **INSÉRER**, **METTRE À JOUR** et **SUPPRIMER**.



PARTIE 1

Exemple

```
17      delimiter //
18      CREATE TRIGGER tr_containte_age BEFORE INSERT
19      ON clients
20      FOR EACH ROW
21      BEGIN
22      DECLARE MinAge INT;
23      SET MinAge:=18;
24      IF NEW.date_naissance >(SELECT DATE_SUB(curdate(), interval MinAge year)) THEN
25      SIGNAL SQLSTATE '50001' SET MESSAGE_TEXT = 'La personne doit etre age de plus de  18.';
26      END IF;
27      END//
28      delimiter ;
29      -- tester le declencheur par une operation insert sur clients
30      INSERT INTO clients(nom,prenom,date_naissance,adresse)
31      Values("Slami","saida","2010-5-22","casa")
```

Output

Action Output

#	Time	Action	Message
2	11:11:06	INSERT INTO clients(nom,prenom,date_naissance,adresse) Values("Slami","saida","2010-5...	Error Code: 1644. La personne doit etre age de plus de 18.

Avantages

Les déclencheurs contribuent :

- À renforcer l'intégrité des données
- À la détection des erreurs
- Aux tâches qui peuvent être exécutées automatiquement lorsque le déclencheur se déclenche plutôt que d'être planifiées.
- À auditer les changements de données, enregistrer les événements et aider à prévenir les transactions invalides.

Restrictions avec les déclencheurs

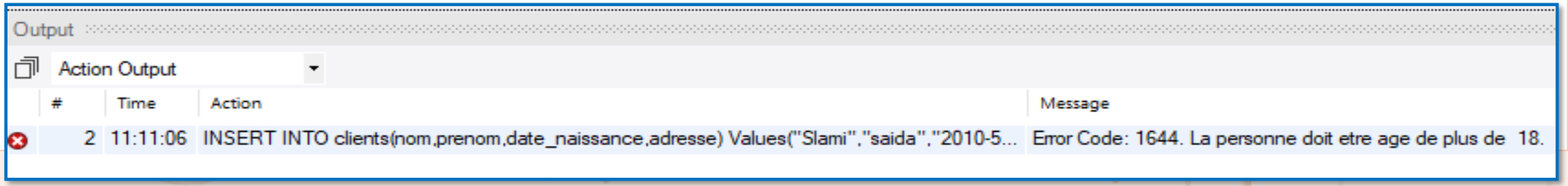
- On ne peut avoir qu'un seul déclencheur par événement ; On ne peut avoir qu'un seul déclencheur BEFORE UPDATE sur une table donnée, mais on peut y exécuter plusieurs instructions ;
- Les déclencheurs ne renvoient pas de valeurs ;
- Ils ne peuvent pas utiliser l'instruction CALL et ils ne peuvent pas créer de tables ou de vues temporaires ;
- Les déclencheurs peuvent entraîner des résultats incohérents en fonction de plusieurs facteurs, notamment le type de moteur de base de données (InnoDB, MyISAM,..) utilisé avec la table à laquelle le déclencheur a été attribué.

OLD/NEW

- le mot-clé OLD est utilisé pour accéder aux données de ligne qui sont remplacées par la mise à jour.
- Le mot-clé NEW permet d'accéder aux données de ligne entrantes qui remplaceront l'ancienne ligne, en cas de succès.
- Selon le type de déclencheur créé, les lignes OLD et NEW peuvent ne pas être disponibles :
  - INSERT TRIGGER : Accès possible a New uniquement.
  - UPDATE TRIGGER : Accès aux deux pseudo-lignes NEW et OLD
  - DELETE TRIGGER : Accès uniquement a OLD pseudo-lignes, c'est-à-dire qu'il n'y a pas de ligne OLD sur un déclencheur INSERT et pas de ligne NEW sur un déclencheur DELETE.

Exemples : BEFORE INSERT

```
DELIMITER //
CREATE TRIGGER tr_contrainte_age BEFORE INSERT
FOR clients
BEGIN
        DECLARE MinAge INT ;
        SET MinAge := 18;
        IF NEW.date_naissance >(SELECT DATE_SUB (curdate(), interval MinAge year)) THEN
                SIGNAL SQLSTATE '50001' SET MESSAGE_TEXT = 'La personne doit etre agee de plus de 18 ans. ' ;
        END IF ;
END //
DELIMITER ;
- - tester le déclencheur par une opération insert sur la table clients
INSERT INTO clients (nom, prenom, date_naissance, adresse)
VALUES (''Slami'', ''Saida'',''2010-5-22'', ''casa'')
```



Exemples : AFTER INSERT

```
- - Initialisation de l age moyenne age
CREATE TABLE moyenne_age (moyenne double);
INSERT INTO moyenne_age SELECT (AVG (TIMESTAMPDIFF(YEAR, date_naissance, CURDATE())) FROM clients ;
SET SQL_SAFE_UPDATES = 0;


- - Creation du trigger
DROP TRIGGER IF EXISTS mise_ajour_moyenne_age ;
DELIMITER //
CREATE TRIGGER mise_ajour_moyenne_age AFTER INSERT
ON clients
FOR EACH ROW
UPDATE moyenne_age SET moyenne = age SELECT (AVG (TIMESTAMPDIFF(YEAR, date_naissance, CURDATE())) FROM clients ;
DELIMITER ;
- - déclencher le triger
INSERT INTO clients (nom, prenom, date_naissance, adresse)
VALUES (''Dadi'','Hamza'','2000-5-22'','casa'');
- - afficher la nouvelle valeur de l'age moyen
SELECT * FROM moyenne_age;
```

Exemples : BEFORE UPDATE

```
DELIMITER //
CREATE TRIGGER tr_modification_age_controle BEFORE UPDATE
ON clients
FOR EACH ROW

BEGIN
        DECLARE MinAge INT ;
        SET MinAge := 18 ;
        IF NEW.date_naissace > (SELECT DATE_SUB(curdate(), interval MinAge year )) THEN
                SIGNAL SGLSTATE '50002' SET MESSAGE_TEXT = 'La personne doit etre agee de plus de 18 ans.';
        END IF;
END;
DELIMITER ;


- - tester le trigger
UPDATE Clients SET date_naissance = '2010-9-23'
WHERE id = 6;
UPDATE Clients SET date_naissance = '1999-9-23'
WHERE id = 7;
```

Exemples : AFTER UPDATE

- Ce déclencheur after_sales_update est automatiquement déclenché avant qu'un événement de mise à jour ne se produise pour chaque ligne de la table des ventes(sales).
- Si on met à jour la valeur dans la colonne de quantité à une nouvelle valeur, le déclencheur insère une nouvelle ligne pour consigner les modifications dans la table SalesChanges.

```
DELIMITER //

CREATE TRIGGER after_sales_update AFTER UPDATE
ON sales
FOR EACH ROW
BEGIN
        IF OLD.quantity <> NEW.quantity THEN
                INSERT INTO SalesChanges (salesId, beforeQuantity, afterQuantity)
                VALUES (OLD.id, OLD.quantity, NEW.quantity) ;
        END IF ;
END $$

DELIMITER ;
```

Exemples : BEFORE DELETE

Ce déclencheur permet avant de supprimer un salaire de le sauvegarder dans une table d'archivage SalaryArchives.

```
DELIMITER //

CREATE TRIGGER before_salaries_delete BEFORE DELETE
ON salaries
FOR EACH ROW
BEGIN
        INSERT INTO  SalaryArchives (employeeNumber, validFrom, amount)
        VALUES (OLD.employeeNumber, OLD.validFrom, OLD.amount) ;
END $$

DELIMITER ;
```

Exemples : AFTER DELETE

Ce déclencheur permet après la suppression d'un salaire de mettre à jour le budget des salaires stockée dans la table SalaryBudgets.

```
CREATE TRIGGER after_salaries_delete AFTER DELETE
ON salaries
FOR EACH ROW
UPDATE SalaryBudgets
SET total = total - OLD.salary;
```

LISTE des déclencheurs

Pour lister les déclencheur de la base de données, on utilise :

SHOW TRIGGERS;

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Trigger	Event	Table	Statement	Timing	Created	sql_mode
tr_containte_age	INSERT	clients	BEGIN DECLARE MinAge INT; SET MinAge:=18; ...	BEFORE	2022-07-23 11:09:50.16	STRICT_TRA
mise_ajour_moyenne_age	INSERT	clients	UPDATE moyenne_age SET moyenne = (SELEC...	AFTER	2022-07-23 12:05:23.40	STRICT_TRA
tr_modification_age_controle	UPDATE	clients	BEGIN DECLARE MinAge INT; SET MinAge:=18; ...	BEFORE	2022-07-23 12:23:51.08	STRICT_TRA

Suppression d'un déclencheur

- Pour supprimer un trigger, on utilise :

  DROP TRIGGER [**IF EXISTS**] [nom_bd.nom_declencheur];

- **Remarque** : Si on supprime une table, MySQL supprimera automatiquement tous les déclencheurs associés à la table.

Modification d'un déclencheur

- Il n'existe pas une instruction pour modifier un trigger, on doit le supprimer et le recréer.
- **LOCK** permet aux sessions client d'acquérir explicitement des verrous de table dans le but de coopérer avec d'autres sessions pour accéder aux tables, ou d'empêcher d'autres sessions de modifier les tables pendant les périodes où une session nécessite un accès exclusif à celles-ci.
- **NB**: **MariaDB**, dans la version 10.1.4, a ajouté la prise en charge de **CREATE** OR **REPLACE TRIGGER** .

```
LOCK TABLES t1 WRITE ;
DROP TRIGGER t1_bi ;
DELIMITER $$
CREATE TRIGGER t1_bi BEFORE INSERT
ON t1
FOR EACH ROW
BEGIN
        …
END $$
DELIMITER ;
UNLOCK TABLES ;
```

CHAPITRE 2

Optimiser une base de données MySQL

Ce que vous allez apprendre dans ce chapitre :

- Optimiser les requêtes SQL ;
- Optimiser la structure de la base de données ;
- Optimiser la configuration de serveur MySQL.

CHAPITRE 2

Optimiser une base de données MySQL

1. **Optimiser les requêtes SQL ;**
2. Optimiser la structure de la base de données ;
3. Optimiser la configuration de serveur MySQL.

Conseils

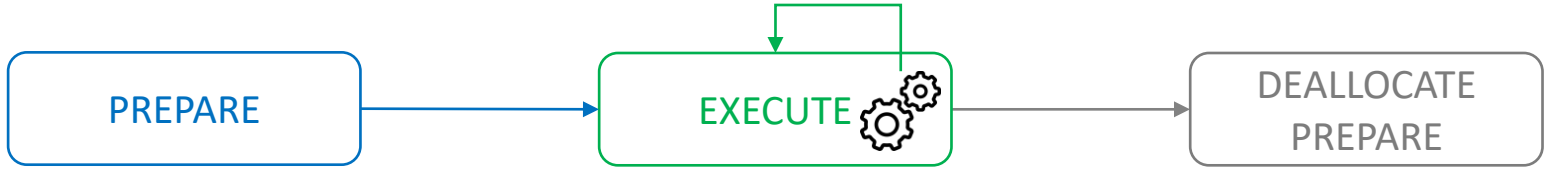
Quelque conseils pour optimiser les requêtes SQL:

- Éviter si possible les **SELECT** * et réduire le nombre de champs, afin de réduire les données chargées en mémoire ;
- Remplacer lees clauses **WHERE** … **IN** par **WHERE EXISTS ;**
- Éviter d'utiliser des fonctions dans les prédicats : exemple **SELECT** * **FROM TABLE1 WHERE UPPER(COL1)='ABC'** ;
- Éviter d'utiliser un caractère générique (%) au début d'un prédicat like ;
- Utiliser la jointure interne (**inner join**), au lieu de la jointure externe (**outer join**) si possible ;
- N'utiliser **DISTINCT** et **UNION** que si nécessaire ;
- N'utiliser **ORDER BY** que si nécessaire ;
- Compter les requêtes sur chaque page, un grand nombre de requêtes peut signifier un « problème N+1 », c'est à dire une requête **SELECT** placée dans une boucle ;
- Utiliser les requêtes préparées ou les procédures stockées facilite la mise en cache des requêtes en interne par **MySQL** et assure un bon niveau de sécurité ;
- Utiliser la clause **EXPLAIN** pour comprendre le fonctionnement d'une requête et quelles sont les clauses qui impactent ses performances.

Instruction préparée

- Avant, MySQL utilise le protocole textuel pour communiquer avec le client , ce protocole a de sérieuses implications sur les performances.
- Pour résoudre ce problème, MySQL a ajouté une nouvelle fonctionnalité appelée instruction préparée depuis la version 4.1.

```
PREPARE stmt1 FROM
        'SELECT id, designation FROM produit
        WHERE id = ? ' ;
- - execution
SET @id = '999' ;
EXECUTE stm1 USING @id ;
- - libération
DEALLOCATE PREPARE stmt1 ;
```



Les Index

- Comme l'index d'un ouvrage vous aide à atteindre les pages concernées par un mot recherché, un index MySQL permet d'accélérer l'accès aux données d'une table.
- La plupart des index de MySQL (**PRIMARY KEY**, **UNIQUE**, **INDEX**, et **FULLTEXT**) sont stockés dans des arbres équilibrés (balanced trees : B-trees).
- D'autres types d'index existent, citons ceux qui portent sur des colonnes **SPATIAL** (reverse key : R-trees), et ceux appliqués aux tables **MEMORY** (tables de hachage : hash).
- L'optimiseur de requête peut utiliser des index pour localiser rapidement les données sans avoir à analyser chaque ligne d'une table pour une requête donnée.
- Lorsqu'on crée une table avec une clé primaire ou une clé unique, MySQL crée automatiquement un index spécial nommé **PRIMARY**. Cet index est appelé index clusterisé.
- L'index **PRIMARY** est spécial car l'index lui-même est stocké avec les données dans la même table. L'index clusterisé applique l'ordre des lignes dans la table.
- Les autres index autres que l'index **PRIMARY** sont appelés index secondaires ou index non clusterisés.

Création d'index

En utilisant ALTER TABLE

- ALTER TABLE nom_table
  ADD INDEX nom_index(liste_colonne)

Lors de la création de la table:

- CREATE TABLE nom_table(
  ……
  INDEX (liste_colonne)
  );

3-Apres la création de la table:

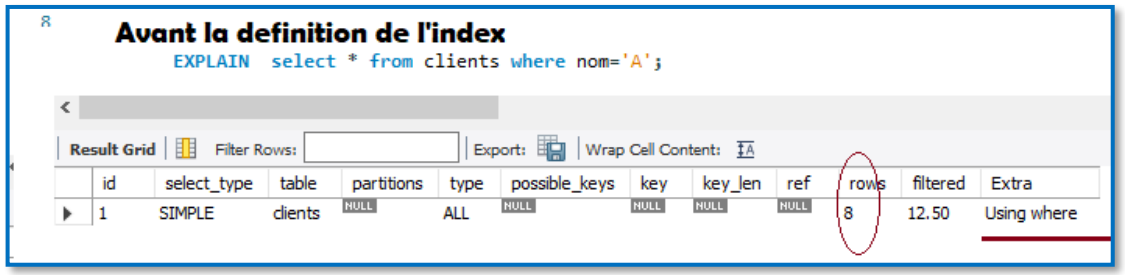
- CREATE [UNIQUE | FULLTEXT | SPATIAL]
  INDEX nomIndex [USING BTREE | HASH]
  ON [nomBase.]nomTable (colonne1 [(taille1)] [ASC | DESC],...) ;

- **UNIQUE** : permet de créer un index qui n'accepte pas les doublons.
- **FULLTEXT** permet de bénéficier de fonctions de recherche dans des textes (flot de caractères).
- **SPATIAL** permet de profiter de fonctions pour les données géographiques
- **ASC** et **DESC** précisent l'ordre (croissant ou décroissant).

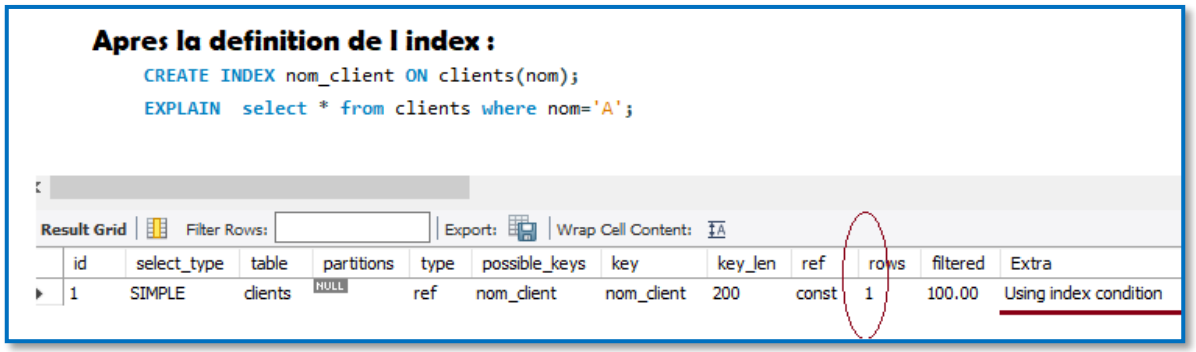
```
USE eshop_app_db;
CREATE UNIQUE INDEX idx_clients_adresse USING BTREE
ON clients (adresse DESC);
```

Avantages

Avant la définition de l'index



Après la définition de l'index



PARTIE 1

Suppression d'un index

DROP INDEX nom_index **ON** nom_table;

DROP INDEX idx_clients **ON** clients;

Avantage

Comme nous pouvons le voir, MySQL n'a eu qu'à localiser 1 ligne à partir de l'index nom_index comme indiqué dans la colonne clé sans parcourir toute la table.

Stratégie d'indexation

Les points importants qu'on doit prendre en compte dans votre stratégie d'indexation sont les suivants :

- Créer une clé primaire (généralement, la colonne se terminera par id) ;
- Prédir les colonnes qui seront souvent interrogées dans votre application avec **WHERE**, **GROUP BY**, **HAVING** et Clauses **ORDER BY** ;
- Avoir un index sur les colonnes qui seront utilisées avec des fonctions, telles que **SUM** (), **COUNT** (), **MIN** (), **MAX** () et **AVG** (); pour en bénéficier en termes de performances;
- Ne surcharger pas la base de données avec trop d'index, car cela aura un impact sur les performances des clés secondaires **MySQL**;
- Prédire avec des index uniques pour accélérer les requêtes de jointure (généralement, les colonnes qui se terminent par _id) ;
- En règle générale, les noms d'utilisateur ou les e-mails les colonnes sont de bons candidats pour l'indexation ; certaines colonnes comme URL ; la plupart des applications ont généralement un UUID.

CHAPITRE 2

Optimiser une base de données MySQL

1. Optimiser les requêtes SQL ;
2. **Optimiser la structure de la base de données** ;
3. Optimiser la configuration de serveur MySQL.

Optimisation de la structure de la base de données

CHAPITRE 2

Optimiser une base de données MySQL

1. Optimiser les requêtes SQL ;
2. Optimiser la structure de la base de données ;
3. **Optimiser la configuration de serveur MySQL.**

Quelque conseils pour l'optimisation des performances d'un serveur MySQL

- Si on utilise des disques durs traditionnels (HDD), on peut effectuer une mise à niveau vers des disques à semi-conducteurs (SSD) pour une amélioration des performances.
- On peut ajuster la cache mémoire pour améliorer les performances. Si on n'a pas assez de mémoire ou si la mémoire existante n'est pas optimisée, on peut finir par nuire à nos performances au lieu de les améliorer.
- MyISAM est un ancien style de base de données utilisé pour certaines bases de données MySQL. C'est une conception de base de données moins efficace. Le plus récent InnoDB prend en charge des fonctionnalités plus avancées et dispose de mécanismes d'optimisation intégrés.
- Essayer d'utiliser la dernière version de MySQL
- Envisager d'utiliser un outil d'amélioration automatique des performances(tuning-primer, MySQLTuner,..)

CHAPITRE 3

Protéger la base de données MySQL

Ce que vous allez apprendre dans ce chapitre :

- Définir la sécurité générale du serveur,
- Implémenter les règles de sécurité et droits d'accès au serveur MySQL
- Gérer les comptes utilisateurs.

CHAPITRE 3

Protéger la base de données MySQL

1. **Définir la sécurité générale du serveur ;**
2. Implémenter les règles de sécurité et droits d'accès au serveur MySQL ;
3. Gérer les comptes utilisateurs.

Définition

- La sécurité des bases de données fait référence à la gamme d'outils, de contrôles et de mesures conçus pour établir et préserver la confidentialité, l'intégrité et la disponibilité des bases de données;
- La sécurité de la base de données doit traiter et protéger les éléments suivants :

Les données dans la base de données

Le système de gestion de base de données (SGBD)

Toutes les applications associées

Le serveur de base de données physique et/ou le serveur de base de données virtuel et le matériel sous-jacent

L'infrastructure informatique et/ou le réseau utilisés pour accéder à la base de données

CHAPITRE 3

Protéger la base de données MySQL

1. Définir la sécurité générale du serveur;
2. **Implémenter les règles de sécurité et droits d'accès au serveur MySQL ;**
3. Gérer les comptes utilisateurs.

- Voici les bonnes pratiques de sécurité des bases de données :

1- S'assurer que les serveurs de bases de données physiques sont sécurisées

2- Utiliser des serveurs de base de données séparés

3- Configurer un serveur proxy HTTPS

4- Implémenter un protocole de cryptage

5- Créer des sauvegardes régulières de la base de données

6- Mettre à jour les applications régulièrement

7- Utiliser une authentification utilisateur forte

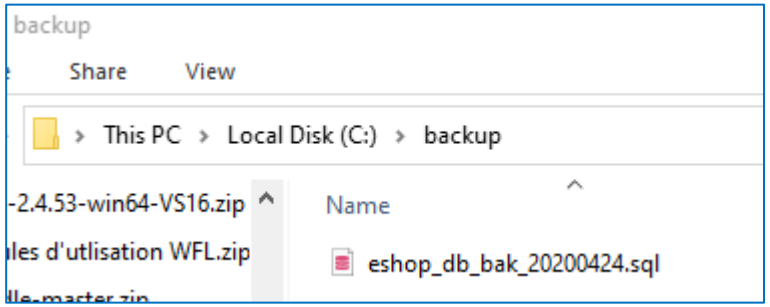
8- Attribuer des rôles de sécurité à tous les utilisateurs

9- Éviter d'utiliser les ports réseau par défaut

Sauvegarde et restauration de la base de données

- Il est essentiel d'effectuer des sauvegardes régulières de la base de données pour pouvoir la récupérer en cas de perte.
- L'outil **mysqldump** permet de faire une sauvegarde d'une ou plusieurs bases de données en générant un fichier texte contenant des instructions SQL qui peuvent recréer les bases de données à partir de zéro.

```
C:\Users\Med>mysqldump -u root -p eshop_app_db > C:\Backup\eshop_db_bak_20200424.sql
Enter password: ****
```



Sauvegarde et restauration de la base de données

- Pour restaurer la base de données **MySQL** avec **mysqldump**, on suit 2 étapes:

Étape 1

- Créer une nouvelle base de données nom_bd

Étape 2

- Restaurer avec la commande:
  mysql -u [utilisateur] -p [nom_bd] < [nom_fichier].sql

```
C:\Users\Med>mysql -u root -p test_bk < C:\Backup\eshop_db_bak_20200424.sql
Enter password: ****
```

Règles de sécurité et droits d'accès au serveur MySQL

CHAPITRE 3

Protéger la base de données MySQL

1. Définir la sécurité générale du serveur;
2. Implémenter les règles de sécurité et droits d'accès au serveur MySQL ;
3. **Gérer les comptes utilisateurs.**

1-création utilisateur:

- Un utilisateur est défini par deux éléments :
  - son login ;
  - l'hôte à partir duquel il se connecte.

- Pour la création d'un nouvel utilisateur, on la syntaxe :
  CREATE USER ' login '@' hote ' [**IDENTIFIED BY** 'mot_de_passe '];

CREATE USER 'saadi '@' localhost ' **IDENTIFIED BY** 'saadi_anass' ;

CREATE USER 'bensaid '@' 194.28.12.4 ' **IDENTIFIED BY** 'bensaid99' ;

1-création utilisateur:

- Il est également possible de permettre à un utilisateur de se connecter à partir de plusieurs hôtes différents (sans devoir créer un utilisateur par hôte) : en utilisant le joker %

- - Said peut se connecter à partir de n'importe quel hôte dont l'adresse IP commence par 194.28.12

CREATE USER 'Said '@' 194.28.12.% ' **IDENTIFIED BY** 'baskeball18' ;

- - Hanane peut se connecter à partir de n'importe quel hôte

CREATE USER 'Hanane '@' % ' **IDENTIFIED BY** 'looking4sun'

1-création utilisateur: Connexion

- Pour se connecter à MySQL avec les informations d'identification d'un utilisateur, on utilise cette commande en invite de commande:

mysql -h <host> -u <user> -p <db>

```
C:\Users\Med>mysql -h localhost -u root -p eshop_app_db
Enter password: ****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 53
Server version: 8.0.28 MySQL Community Server - GPL

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

2 – Renommer un utilisateur

RENAME USER ‘ancien_nom '@’ ancien_hote ' **TO** ‘nouveau_nom'@' noveau_hote';

3 – Modifier le mot de passe d’un utilisateur

SET PASSWORD FOR ‘utilisateur '@‘hote' = **PASSWORD** (‘nouveau_pw')

3 – Supprimer un utilisateur

DROP USER ' login '@ 'hote ';

Les privilèges

- Lorsque l'on crée un utilisateur avec **CREATE USER**, celui-ci n'a au départ aucun privilège, aucun droit ;
- En SQL, avoir un privilège, c'est avoir l'autorisation d'effectuer une action sur un objet. Il existe de nombreux privilèges ;
- Les privilèges **SELECT**, **INSERT**, **UPDATE** ,**DELETE**, **CREATE TABLE**, **CREATE TEMPORARY TABLE** , **CREATE VIEW**, **ALTER TABLE**, **DROP**, **CREATE ROUTINE**, **LOCK TABLES**,…etc, permettent aux utilisateurs d'exécuter ces mêmes commandes
- Il y a différents niveaux d'application des privilèges :

Globale	• *.*
Base de données	• [nom_bd].*
Table	• [nom_bd].[nom_table]
Routine stockée	• [nom_bd].[nom_routine]

Ajout d'un privilège à un utilisateur

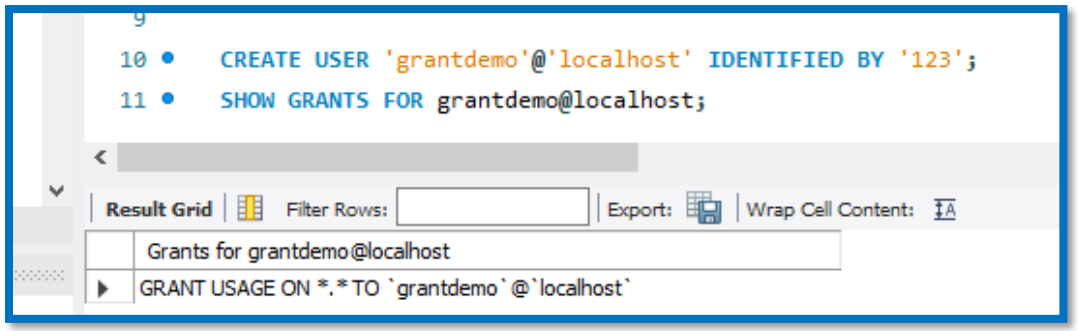
- Pour pouvoir ajouter un privilège à un utilisateur, il faut posséder le privilège **GRANT OPTION**.
- Au début , seul l'utilisateur « root » le possède.
- Pour attribuer des privilèges à un utilisateur, on utilise :

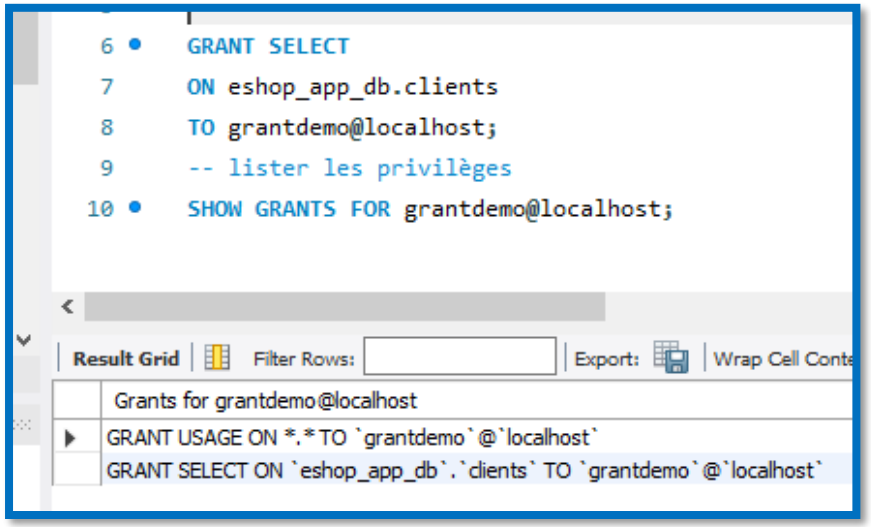
GRANT privilege1 [(liste_colonnes)] [, privilege2 [(liste_colonnes)], ...] **ON** [type_objet] niveau_privilege **TO** utilisateur [**IDENTIFIED BY** mot_de_passe];

Ajout d'un privilège à un utilisateur : exemples

```
GRANT SELECT ,
UPDATE (nom, prenom, adresse ) ,
DELETE ,
INSERT
ON eshop_app_db.clients
TO saadi@localhost ;
```

```
-- accorder le privilège SELECT sur toutes les tables de toutes
-- les base de données à l'utilisateur saadi
GRANT SELECT ,
ON *.*
TO saadi@localhost ;
```





PARTIE 1

Révocation des privilèges

```
REVOKE privilege [, privilege , ...]
ON niveau_privilege
FROM utilisateur ;
```

```
REVOKE INSERT
ON eshop_app_db.clients
FROM saadi@localhost ;
```

Gestion des rôles

- En plus d'accorder des autorisations à des utilisateurs individuels, dans MySQL, il est également possible de créer des rôles et d'accorder des autorisations aux rôles, puis d'attribuer des rôles aux utilisateurs.
- Cela facilite grandement la gestion des groupes d'utilisateurs avec des autorisations similaires. Pour créer un rôle de développeur Web, nous pouvons fournir la requête suivante :

CREATE ROLE 'nom_role';

```
CREATE ROLE 'webdeveloper';
-- affecter des privileges au rôle
GRANT SELECT ON mysql.user TO 'webdeveloper';
-- ajouter un utilisateur au rôle
GRANT 'webdeveloper' TO 'rh_salmi'@'entrbc.com.net';
```

- Un compte peut n'avoir aucun rôle, un seul rôle ou plusieurs rôles.
- Si un rôle est accordé à un utilisateur, on doit peut-être indiquer à MySQL quels rôles vous souhaitez utiliser à l'aide de la requête suivante: **SET ROLE** 'webdeveloper'.

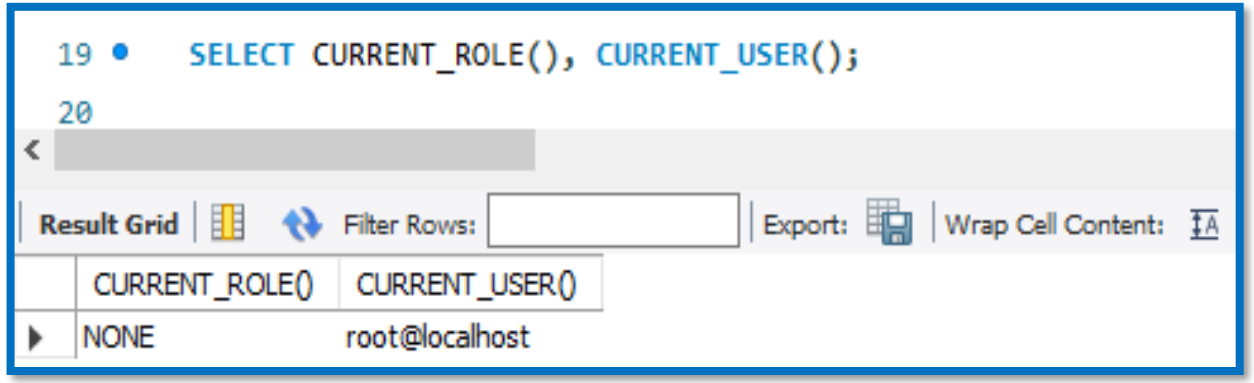
Gestion des rôles

- La création d'un utilisateur, l'attribution d'un rôle et la définition de ce rôle par défaut peuvent être effectuées dans une seule instruction telle que :

```
CREATE USER 'u2'@'%'
INDENTIFIED BY 'foobar'
DEFAULT ROLE 'webdeveloppper';
```

- Pour voir quel utilisateur et quel rôle on utilise, on peut exécuter la commande :

```
SELECT CURRENT_ROLE(), CURRENT_USER ();
```



Exclusion un utilisateur d'un rôle

- Pour exclure un utilisateur d'un rôle, on peut utiliser la commande REVOKE

  REVOKE '<role>' **FROM** '<user>'@'<host>';

REVOKE 'webdeveloppeur' **FROM** 'med34'@'localhost';