## 1. Les Fonctions en MySQL

### Syntaxe Générale

```
DELIMITER $
DROP FUNCTION IF EXISTS nom_fonction;
CREATE FUNCTION nom_fonction(paramètres)
RETURNS type_retour
DETERMINISTIC | READS SQL DATA | MODIFIES SQL DATA | CONTAINS SQL
BEGIN
  -- instructions
  RETURN valeur;
END;
$
```

### Détails :

| Mot-clé | Description |
|---|---|
| DETERMINISTIC | Retourne toujours le même résultat pour les mêmes arguments. |
| READS SQL DATA | Lit des données depuis la base sans les modifier. |
| MODIFIES SQL DATA | Modifie les données dans la base (via UPDATE, INSERT, etc.). |
| CONTAINS SQL | Contient du SQL, mais ne lit ni ne modifie les données (ex: SELECT NOW()). |

### Qu'est-ce que DELIMITER en MySQL ?

Par défaut, MySQL utilise le point-virgule ; comme délimiteur de fin d'instruction. Mais quand tu veux écrire des blocs complets comme une fonction ou une procédure avec plusieurs instructions SQL, il faut changer ce délimiteur temporairement pour éviter que MySQL pense que l'instruction est terminée trop tôt.

C'est pour cela qu'on utilise :

```
DELIMITER $
```

À la fin, on remet le délimiteur par défaut :

```
DELIMITER ;
```

Problème si on n'utilise pas DELIMITER avec plusieurs instructions

Prenons une fonction sans DELIMITER :

```
CREATE FUNCTION test()
RETURNS INT
BEGIN
 DECLARE x INT;
 SET x = 5;
 RETURN x;
END;
```

MySQL va s'arrêter à SET x = 5; car il voit le ; comme fin du bloc.

tu n'es pas obligé d'utiliser toujours $ comme délimiteur.
$ est juste un exemple de nouveau délimiteur. Tu peux mettre ce que tu veux :
$$, //, %%, ###, etc.

### Exemple 4 : nombre_stg2(nombre)

```
DELIMITER $

CREATE FUNCTION nombre_stg2(nombre INT)
RETURNS INT
MODIFIES SQL DATA
DETERMINISTIC
BEGIN
 IF (nombre = 0) THEN
   BEGIN
     UPDATE stg SET moy = moy + 1 WHERE mat = 20;
     SET nombre = 1;
   END;
 ELSE
   BEGIN
     SET nombre = 2;
   END;
 END IF;

 RETURN nombre;
END;
$
```

Explication :

- Si l'utilisateur donne 0, alors :
  - la moyenne de l'étudiant avec mat = 20 augmente de 1
  - la fonction retourne 1
- Si c'est autre chose → elle retourne 2

Appels :

```
SELECT nombre_stg2(0);  -- met à jour la table et retourne 1
SELECT nombre_stg2(5);  -- ne modifie rien et retourne 2
```

### Exemple 5 : myfunction()

## 3. Les variables dans MySQL

### Déclaration :

```
DECLARE nom_variable TYPE DEFAULT valeur;
```

Exemple :

```
DECLARE s INT DEFAULT 0;
```

### Affectation :

```
SET nom_variable = nouvelle_valeur;
```

## 4. Les boucles (WHILE)

Syntaxe :

```
WHILE condition DO
   instructions;
END WHILE;
```

Exemple :

```
DECLARE i INT DEFAULT 1;
WHILE i <= 100 DO
  SET i = i + 1;
END WHILE;
```

## 5. Les conditions (IF, ELSE)

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

## 6. Exemples expliqués

```
CREATE FUNCTION somme()
RETURNS INT
BEGIN
  DECLARE s INT DEFAULT 0;
  DECLARE i INT DEFAULT 1;
  WHILE i <= 100 DO
    SET s = s + i;
    SET i = i + 1;
  END WHILE;
  RETURN s;
END;
```

But : Calculer la somme de 1 à 100.
Retourne : 5050

### Exemple 2 : mysomme(N)

Même logique, mais la limite est un paramètre N donné par l'utilisateur.

```
DELIMITER $

CREATE FUNCTION mysomme(N INT)
RETURNS INT
DETERMINISTIC
BEGIN
 DECLARE s INT DEFAULT 0;    -- s va stocker la somme
 DECLARE i INT DEFAULT 1;    -- compteur de 1 jusqu'à N

 WHILE(i <= N) DO          -- tant que i est <= N
  SET s = s + i;          -- on ajoute i à la somme
  SET i = i + 1;          -- on augmente le compteur
 END WHILE;

 RETURN s;               -- on retourne la somme
END;
$
```

Appel :

```
SELECT mysomme(50);  -- 1 + 2 + 3 + ... + 50 = 1275
```

### Exemple 3 : nombre_stg(v_nom)

Objectif :

Retourner le nombre de stagiaires dont le nom est v_nom.

```
DELIMITER $
```

📌 Objectif :
Retourner simplement la date actuelle.

```
DELIMITER $

CREATE FUNCTION myfunction()
RETURNS DATE
CONTAINS SQL
DETERMINISTIC
BEGIN
 DECLARE d DATE;
 SET d = CURRENT_DATE();  -- récupère la date d'aujourd'hui
 RETURN d;            -- retourne la date
END;
$
```

📌 Appel :

```
SELECT myfunction();  -- Par exemple : 2025-04-24
```

## 7. Comment appeler une fonction MySQL ?

C'est très simple !
Il suffit d'écrire :

```
SELECT nom_fonction(valeurs);
```

📝 Résumé tableau

| Élément | Description |
|---|---|
| DELIMITER $ | Change le délimiteur pour écrire plusieurs instructions SQL dans un bloc |
| DECLARE | Crée une variable locale dans la fonction |
| SET | Change la valeur d'une variable |
| WHILE ... DO ... | Boucle tant que la condition est vraie |
| IF ... THEN ... | Condition (si, sinon) |
| RETURN valeur | Valeur que la fonction retourne |
| SELECT ... INTO | Met le résultat d'un SELECT dans une variable |

```
CREATE FUNCTION nombre_stg(v_nom VARCHAR(50))
RETURNS INT
READS SQL DATA
BEGIN
 DECLARE nombre INT;    -- pour stocker le résultat

 SELECT COUNT(*) INTO nombre    -- on compte les lignes
 FROM stg
 WHERE nom = v_nom;          -- seulement ceux avec le nom
donné

 RETURN nombre;             -- on retourne ce nombre
END;
$
```

📌 Appel :

```
SELECT nombre_stg('Ahmed');
```

🟫 Si dans la table stg il y a 3 Ahmed → Résultat : 3

## Remarques sur les fonctions et procédures stockées MySQL :

- On ne peut pas renvoyer une table depuis une fonction MySQL.
- Une fonction peut uniquement renvoyer une chaîne, un entier, un caractère, etc.
- Pour renvoyer une table dans MySQL, on utilise une procédure stockée, pas une fonction.

## 🌟 Qu'est-ce qu'une procédure stockée en MySQL ?

✅ Définition simple :
Une procédure stockée (stored procedure) est un bloc de code SQL (plusieurs instructions) enregistré dans la base de données.
Tu peux l'appeler plusieurs fois pour exécuter des actions (sélection, insertion, mise à jour...) automatiquement.

✅ Pourquoi utiliser une procédure stockée ?

- Gagner du temps.
- Automatiser les traitements.
- Réutiliser le code facilement.
- Sécuriser certaines opérations.
-

## 🔄 Comment fonctionne une procédure stockée ?

1. Création d'une procédure
Tu écris la procédure une seule fois et l'enregistres dans MySQL.

```
DELIMITER $
CREATE PROCEDURE afficher_stagiaires()
BEGIN
  SELECT * FROM stg;
END;
$
```

## 🖊️ Exemples pratiques

➡️ Avec paramètre IN (envoyer une valeur)

```
DELIMITER $
CREATE PROCEDURE chercher_stagiaire(IN v_nom
VARCHAR(50))
BEGIN
  SELECT * FROM stg WHERE nom = v_nom;
END;
$
```

Appel :

```
CALL chercher_stagiaire('rajae');
```

➡️ Avec paramètre OUT (recevoir une valeur)

```
DELIMITER $
CREATE PROCEDURE compter_stagiaires(IN v_nom
VARCHAR(50), OUT nombre INT)
BEGIN
  SELECT COUNT(*) INTO nombre FROM stg WHERE nom =
v_nom;
END;
$
```

Appel :

```
CALL compter_stagiaires(' ahmed ', @total);
 SELECT @total;
```

🛠 Ici :

- DELIMITER $ : on change le délimiteur pour écrire plusieurs lignes sans erreur.
- CREATE PROCEDURE : pour créer une procédure appelée afficher_stagiaires.
- BEGIN ... END : tout le code de la procédure est entre BEGIN et END.
- SELECT * FROM stg; : affiche tous les stagiaires.
-

2. Appel d'une procédure
Après l'avoir créée, tu peux l'appeler comme ceci :

```
CALL afficher_stagiaires();
```

## Les types de paramètres dans les procédures

| Type | Description |
|---|---|
| IN | Tu envoies une valeur à la procédure. |
| OUT | La procédure te renvoie une valeur. |
| INOUT | Tu envoies une valeur, et elle est modifiée par la procédure. |

👉 Qu'est-ce qu'une variable de session dans MySQL ?
C'est une variable temporaire que MySQL garde uniquement pour ta session (ta connexion actuelle).
Elle commence toujours par @.
Tu n'as pas besoin de la déclarer (DECLARE), tu peux l'utiliser directement.

```
CALL get_total("ahmed", @total);
SELECT @total;
```

## Exemple :

```
SET @mon_nom = ' ahmed ';
SELECT @mon_nom;
```

| Terme | Explication |
|---|---|
| `@variable` | Variable de session MySQL, accessible pendant ta connexion |
| Utilité | Recevoir des résultats d'une procédure, stocker des données temporaires |

Parce que dans MySQL, si tu veux créer une variable hors d'une procédure (c'est-à-dire une variable de session), tu dois obligatoirement mettre le @ devant

➡ Avec paramètre INOUT (envoyer et modifier)

```
DELIMITER $
CREATE PROCEDURE incrementer_compteur(INOUT counter
INT, IN inc INT)
BEGIN
  SET counter = counter + inc;
END;
$
```

Appel :

```
SET @counter = 1;
CALL incrementer_compteur(@counter, 5);
SELECT @counter; -- Résultat sera 6
```

## Attention

DELIMITER est nécessaire pour écrire plusieurs lignes dans une procédure.

Pour supprimer une procédure :

```
DROP PROCEDURE IF EXISTS nom_de_la_procédure;
```

## Résumé rapide

Une fonction retourne un seul résultat simple (pas une table).

Une procédure peut exécuter plusieurs instructions et retourner des données (par OUT ou via un SELECT).

IN → envoyer une valeur.
OUT → récupérer une valeur.
INOUT → envoyer et modifier une valeur.

## Important :

- DELIMITER $ → tu changes temporairement le délimiteur (sinon MySQL croit que ta procédure est finie dès qu'il voit ;).
- Le $ marque la fin de la procédure.
- DELIMITER ; → tu reviens au délimiteur normal ; après avoir fini.

## 1. Définition du verrou (LOCK) dans MySQL

### Qu'est-ce qu'un verrou ?

Un verrou est un mécanisme de sécurité dans une base de données. Il permet à une session d'utilisateur (appelée session client) de bloquer l'accès à une table pendant qu'elle y travaille, pour éviter les conflits ou les accès concurrents.

### Exemple expliqué dans l'image :

- Session 1 fait LOCK TABLES → Elle bloque la table pour son propre usage.
- Session 2 veut faire un SELECT → Elle doit attendre car la table est verrouillée par Session 1.

### Commandes :

```
LOCK TABLES nom_table READ;  -- Lecture uniquement
LOCK TABLES nom_table WRITE;  -- Lecture + écriture
UNLOCK TABLES;          -- Libère les verrous
```

### Règles importantes :

- Une session peut verrouiller et déverrouiller uniquement pour elle-même.
- Une session ne peut pas contrôler les verrous des autres sessions.

## 2. Variables dans MySQL

a) Variables locales (dans procédures/fonctions)
Elles sont utilisées dans les blocs BEGIN ... END.

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

b) Variables de session (externes) – visibles seulement par une session
Utilisent le symbole @. Exemple :

```
SET @max_prix := 100;      -- affectation directe
SELECT @max_prix := MAX(pu) FROM produits;  -- à partir d'une requête
```

Puis on peut les utiliser :

```
SELECT * FROM produit WHERE pu = @max_prix;
```

## 3. Structures conditionnelles IF

a) IF simple

```
IF condition THEN
instructions;
END IF;
b) IF ... ELSE
IF condition THEN
instructions_si_vrai;
ELSE
instructions_si_faux;
END IF;
c) IF ... ELSEIF ... ELSE
IF condition1 THEN
instructions1;
ELSEIF condition2 THEN
instructions2;
ELSE
instructions3;
END IF;
```

Exemple :

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

## 4. Fonction stockée (Function)

Une fonction en SQL permet de calculer une valeur et la retourner.

Exemple : IncomeLevel

Fonction qui classe le revenu mensuel :

```
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 ;
```

## C'est quoi une structure CASE ?

La structure CASE permet d'exécuter des instructions différentes selon une ou plusieurs conditions. C'est l'équivalent du IF...ELSE mais plus lisible quand on a plusieurs cas à traiter.Il y a deux types de CASE :

### 1. CASE avec sélecteur

Idée :On compare une seule variable à plusieurs valeurs possibles.

Syntaxe :

```
CASE variable
  WHEN valeur1 THEN instructions1;
  WHEN valeur2 THEN instructions2;
  ...
  ELSE instructionsParDéfaut;
END CASE;
```

Exemple réel :

```
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;
```

Explication :

- Tu déclares une procédure qui prend le numéro d'un client (pCustomerNumber).
- Tu récupères son pays (customerCountry).
- Ensuite tu compares ce pays :
  - Si c'est 'USA', tu choisis la livraison en 2 jours.
  - Si c'est 'Canada', en 3 jours.
  - Sinon, en 5 jours.

C'est très utile quand tu veux choisir selon une valeur fixe.

### 2. CASE sans sélecteur

Idée :Tu n'utilises pas de variable spécifique à comparer. Tu fais des tests logiques directs.

Syntaxe :

```
CASE
  WHEN condition1 THEN instructions1;
  WHEN condition2 THEN instructions2;
  ...
  ELSE instructionsParDéfaut;
END CASE;
```

Exemple 1 : Avec une requête SELECT

```
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 AS QuantityStatus
FROM OrderDetails;
```

Explication :

- Tu ajoutes une colonne calculée appelée QuantityStatus.
- Elle affiche un message différent selon la quantité (Quantity) de chaque commande.

Exemple 2 : Affectation d'une mention selon une note

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

Explication :

- Tu déclares une variable v_note qui contient une note.
- Tu veux attribuer une mention :
  - TB : Très Bien
  - B : Bien
  - AB : Assez Bien
  - P : Passable
  - R : Refusé

Ici, on teste plusieurs conditions plutôt qu'une valeur exacte.

## 🔁 1. BOUCLE LOOP

🧠 **Idée :**

La boucle LOOP exécute des instructions jusqu'à ce qu'on l'arrête avec LEAVE. C'est une boucle infinie si tu n'as pas de condition d'arrêt.

📌 **Syntaxe :**

```
[label:] LOOP
  instructions
  IF condition THEN
    LEAVE label;
  END IF;
END LOOP;
```

✅ **Exemple pratique :**

```
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$$

DELIMITER ;

CALL ps_nombres();
SELECT * FROM test;
```

📄 **Explication :**

- La boucle s'appelle simple_loop.
- Elle insère les nombres de 1 à 10 dans la table test.
- Quand a = 11, on quitte la boucle avec LEAVE.

## 🟨 Commande spéciale : LEAVE

- LEAVE nom_de_boucle; = Sort immédiatement de la boucle (comme break en PHP, Java...).

## 🔁 2. Commande ITERATE

🧠 **Idée :**

Permet de revenir au début de la boucle, sans exécuter ce qui suit (comme continue dans d'autres langages).

✅ **Exemple avec LEAVE et ITERATE :**

```
DELIMITER $$

CREATE PROCEDURE LoopDemo()
BEGIN
 DECLARE x INT DEFAULT 1;
 DECLARE str VARCHAR(255) DEFAULT '';

 loop_label: LOOP
  IF x > 10 THEN
   LEAVE loop_label;
  END IF;

  IF x MOD 2 = 0 THEN
   SET x = x + 1;
   ITERATE loop_label; -- On saute les nombres pairs
  END IF;

  SET str = CONCAT(str, x, ','); -- On ajoute les nombres impairs
  SET x = x + 1;
 END LOOP;

 SELECT str;
END$$

DELIMITER ;

CALL LoopDemo();
```

🔍 **Résultat :**

1,3,5,7,9,

## 🔁 3. BOUCLE WHILE

🧠 **Idée :**

Le bloc de code est exécuté tant que la condition est vraie.

📌 **Syntaxe :**

```
[label:] WHILE condition DO
 instructions;
END WHILE;
```

✅ **Exemple :**

```
DELIMITER $$

CREATE PROCEDURE dowhile()
BEGIN
 DECLARE v1 INT DEFAULT 5;

 WHILE v1 > 0 DO
  SET v1 = v1 - 1;
 END WHILE;
END$$

DELIMITER ;
```

📄 **Explication :**

- La boucle exécute SET v1 = v1 - 1 jusqu'à ce que v1 soit 0.
- Condition testée avant d'entrer dans la boucle.

## 🔁 4. BOUCLE REPEAT

🧠 **Idée :**

Le bloc est toujours exécuté au moins une fois, puis répété jusqu'à ce que la condition soit vraie.

📌 **Syntaxe :**

```
REPEAT
 instructions;
UNTIL condition
END REPEAT;
```

✅ **Exemple :**

```
DELIMITER $$

CREATE PROCEDURE dorepeat(p1 INT)
BEGIN
 DECLARE x INT DEFAULT 0;

 REPEAT
  SET x = x + 1;
 UNTIL x > p1
 END REPEAT;

 SELECT x;
END$$

DELIMITER ;

CALL dorepeat(5);
```

📄 **Explication :**

- x est incrémenté jusqu'à ce qu'il dépasse p1.
- Le code à l'intérieur de la boucle s'exécute au moins une fois même si la condition est déjà vraie.

## 🔷 Définition trigger :

Un trigger (ou déclencheur) est un bloc de code SQL qui s'exécute automatiquement en réponse à un événement (INSERT, UPDATE, DELETE) sur une table.

### 🔷 Syntaxe générale :

```
CREATE TRIGGER nom_declencheur
[BEFORE | AFTER]  -- moment de déclenchement
{INSERT | UPDATE | DELETE}  -- type d'opération
ON nom_table
FOR EACH ROW      -- exécution pour chaque ligne
BEGIN
  -- Corps du déclencheur (instructions SQL)
END;
```

### 🔷 Types de déclencheurs :

| Événement | BEFORE | AFTER |
|---|---|---|
| INSERT | ✅ | ✅ |
| UPDATE | ✅ | ✅ |
| DELETE | ✅ | ✅ |

### 🔷 Exemple concret : contrôle de l'âge minimum (18 ans)

```
DELIMITER //
CREATE TRIGGER tr_containte_age
BEFORE INSERT ON clients
FOR EACH ROW
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 être âgée de plus de 18
ans.';
  END IF;
END;
//
DELIMITER ;
```

▶️ Ce déclencheur empêche d'insérer un client de moins de 18 ans.

### 🔷 Tester le déclencheur :

```
INSERT INTO clients (nom, prenom, date_naissance, adresse)
VALUES ("Slami", "saida", "2010-05-22", "casa");
```

🔴 Cela va échouer avec un message d'erreur car la personne a moins de 18 ans.

### 🔍 Exemple avec valeur réelle SELECT DATE_SUB(CURDATE(), INTERVAL MinAge YEAR)):

Si on a :

```
SET MinAge := 18;
```

Et qu'aujourd'hui on est le :

```
CURDATE() = '2025-06-06'
```

Alors :

```
SELECT DATE_SUB('2025-06-06', INTERVAL 18 YEAR)
```

→ Retournera :

```
'2007-06-06'
```

### 🔷 Pseudo-tables OLD et NEW :

| Type de Trigger | Accès à OLD | Accès à NEW |
|---|---|---|
| INSERT | ❌ | ✅ |
| UPDATE | ✅ | ✅ |
| DELETE | ✅ | ❌ |

NEW.colonne : nouvelle valeur (après insertion ou mise à jour).
OLD.colonne : ancienne valeur (avant mise à jour ou suppression).

### 🔷 Restrictions importantes :

- ⚠️ Un seul déclencheur par événement (par exemple, un seul BEFORE INSERT par table).
- 🚫 Les déclencheurs ne retournent pas de valeurs.
- 🚫 Ils ne peuvent pas utiliser CALL, ni créer de tables ou vues temporaires.
- ⚠️ Les comportements peuvent varier selon le moteur de base de données utilisé (InnoDB, MyISAM...).

## 🔷 Définition d'un curseur

Un curseur est un mécanisme qui permet de parcourir ligne par ligne un résultat d'une requête SELECT.Il est utilisé dans les procédures stockées, quand on a besoin de traiter chaque ligne individuellement (ex. : pour générer un texte, faire des calculs, ou envoyer des emails ligne par ligne).

### 🔷 Caractéristiques d'un curseur MySQL

| Propriété | Description |
|---|---|
| Read-only | On **ne peut pas modifier** directement les lignes avec le curseur. |
| Non-scrollable | On ne peut parcourir les lignes que **dans un seul sens (avant)**. |
| Asensitive | Les changements dans la base **ne sont pas visibles** par le curseur après ouverture (le curseur lit une **copie**). |

### 🔷 Étapes pour utiliser un curseur

1. Déclarer le curseur avec une requête SELECT
2. Ouvrir le curseur avec OPEN
3. Lire ligne par ligne avec FETCH
4. Fermer le curseur avec CLOSE

### 🔷 Syntaxe

```
DELIMITER $$
CREATE PROCEDURE nom_procedure()
BEGIN
-- 1. Déclaration des variables
  DECLARE fini INT DEFAULT 0;
  DECLARE var1 TYPE;
  DECLARE var2 TYPE;
  -- ... autant de variables que de colonnes sélectionnées
-- 2. Déclaration du curseur
  DECLARE nom_curseur CURSOR FOR
    SELECT colonne1, colonne2 FROM nom_table;
-- 3. Gestion de la fin de lecture
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET fini = 1;
-- 4. Ouverture du curseur
  OPEN nom_curseur;
-- 5. Lecture ligne par ligne avec FETCH
  nom_boucle: LOOP
    FETCH nom_curseur INTO var1, var2;
    IF fini = 1 THEN
      LEAVE nom_boucle;
    END IF;
    -- Traitement à faire pour chaque ligne
  END LOOP nom_boucle;
  -- 6. Fermeture du curseur
  CLOSE nom_curseur;
END $$
DELIMITER ;
```

### ✅ Exemple Générale d'un Curseur en MySQL

```
DELIMITER $$
CREATE PROCEDURE lister_clients()
BEGIN
  -- Déclarations des variables
  DECLARE finished INT DEFAULT 0;
  DECLARE v_id INT;
  DECLARE v_nom VARCHAR(100);
  DECLARE v_prenom VARCHAR(100);
  DECLARE info VARCHAR(400) DEFAULT "";
  DECLARE resultat_txt TEXT DEFAULT "";
-- Déclaration du curseur
  DECLARE cur_info_client CURSOR FOR
    SELECT id, nom, prenom FROM clients;
-- Gestion de fin de curseur
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET finished = 1;
  -- Ouvrir le curseur
  OPEN cur_info_client;
  -- Boucle pour parcourir chaque ligne
  boucle_parcours_clients: LOOP
    FETCH cur_info_client INTO v_id, v_nom, v_prenom;
    IF finished = 1 THEN
      LEAVE boucle_parcours_clients;
    END IF;
    -- Traitement (ex : concaténer les infos)
    SET info = CONCAT(v_id, " - ", v_nom, " ", v_prenom);
    SET resultat_txt = CONCAT(resultat_txt, info, "; ");
  END LOOP boucle_parcours_clients;
-- Fermer le curseur
  CLOSE cur_info_client;
  -- Affichage (facultatif)
  SELECT resultat_txt;
END $$
DELIMITER ;
```

## ✅ Introduction aux Transactions MySQL

Par défaut, le client MySQL fonctionne en mode autocommit, c'est-à-dire que chaque instruction DML (INSERT, UPDATE, DELETE) est immédiatement validée (committed) après exécution.
Cela signifie qu'on ne peut pas revenir en arrière (pas d'annulation possible).
Mais si on veut pouvoir annuler certaines opérations (par exemple si une erreur survient), il faut désactiver le mode autocommit et utiliser une transaction manuelle.

### 🔄 Activer une transaction manuelle

Méthodes pour démarrer une transaction :

```
START TRANSACTION;
-- ou bien
BEGIN;
```

Exemple :

```
START TRANSACTION;
INSERT INTO mytable VALUES (1, 'foo', 'bar', 'baz');
-- On peut consulter l'état actuel
SELECT * FROM mytable;

-- Si tout va bien :
COMMIT;

-- Si on veut annuler :
--ROLLBACK;
```

### ⚙️ Désactiver le mode autocommit

Tu peux aussi désactiver le mode autocommit manuellement :

```
SET autocommit = 0;
-- ou
SET autocommit = OFF;
```

Cela signifie que toutes les instructions suivantes feront partie d'une transaction jusqu'à ce que tu fasses COMMIT ou ROLLBACK.

### 🔁 Commandes de contrôle des transactions

| Commande | Description |
|---|---|
| COMMIT | Valide définitivement les modifications |
| ROLLBACK | Annule toutes les modifications depuis le début de la transaction |

Ces commandes s'appliquent uniquement aux instructions DML (INSERT, UPDATE, DELETE), pas aux commandes DDL (CREATE, DROP, etc.).

### 💳 Solution avec transaction :

📜 Version complète en procédure stockée (avec condition IF)

```
DELIMITER //
CREATE PROCEDURE transfert(
  IN expediteur VARCHAR(10),
  IN destinataire VARCHAR(10),
  IN montant DECIMAL(10,2)
)
BEGIN
  DECLARE solde_actuel DECIMAL(10,2);

  START TRANSACTION;

  -- Récupérer le solde actuel de l'expéditeur
  SELECT solde INTO solde_actuel FROM compte WHERE accountno =
expediteur;

  -- Vérification du solde suffisant
  IF solde_actuel >= montant THEN
    UPDATE compte SET solde = solde - montant WHERE accountno =
expediteur;
    UPDATE compte SET solde = solde + montant WHERE accountno =
destinataire;

    COMMIT;

  ELSE
    ROLLBACK;
  END IF;

END //

DELIMITER ;
```

### 📌 Utilisation de la procédure

```
CALL transfert('ACC1', 'ACC2', 200.00);
```

### ✅ Résultat

- Si ACC1 a au moins 200 DH, le transfert est effectué.
- Sinon, rien ne change.

## 🔵 1. Optimisation des requêtes SQL

Voici quelques bonnes pratiques pour rendre vos requêtes SQL plus rapides et efficaces :

### ✅ Sélection intelligente des données :

- Éviter SELECT * : ne sélectionne que les colonnes nécessaires.
- Réduire les données chargées allège la mémoire et accélère l'exécution.

### ✅ Clauses WHERE et jointures :

- Remplacer WHERE ... IN par WHERE EXISTS : plus performant pour de grandes sous-requêtes.
- Éviter les fonctions dans les conditions : par exemple UPPER(colonne) empêche l'utilisation des index.
- Éviter % au début d'un LIKE : cela empêche aussi l'utilisation des index.
- Préférer les INNER JOIN aux OUTER JOIN si possible.

### ✅ Opérations coûteuses :

- Éviter DISTINCT et UNION sauf si nécessaire : très gourmands en ressources.
- Limiter l'usage de ORDER BY.

### ✅ Bonnes pratiques de développement :

- Éviter le "problème N+1" (boucles avec requêtes répétées).
- Utiliser des requêtes préparées ou procédures stockées : sécurisées et rapides.
- Utiliser EXPLAIN pour analyser la performance d'une requête et voir l'usage des index.

## 🟢 2. Instructions préparées (Prepared Statements)

Utiles pour sécurité (évite les injections SQL) et performances :

### 🧩 Exemple :

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

### 🧠 Avantages :

- Exécution rapide si réutilisée plusieurs fois.
- Paramètres protégés.
- Moins de charge côté serveur.

## 🟡 3. Optimisation via les Index

Les index sont comme l'index d'un livre : ils permettent d'aller rapidement à la ligne recherchée sans lire toute la table.

### 🗂️ Types d'index :

- PRIMARY KEY: unique, non nul, clusterisé (les données sont triées selon cette clé).
- UNIQUE: interdit les doublons.
- INDEX : index simple pour accélérer les requêtes.
- FULLTEXT: recherche dans du texte.
- SPATIAL: pour données géographiques (R-tree).
- HASH: utilisé avec les tables MEMORY.

### 📂 Création :

```
-- Lors de la création
CREATE TABLE clients (
 id INT PRIMARY KEY,
 adresse VARCHAR(255),
 INDEX (adresse)
);

-- Après création
ALTER TABLE clients ADD INDEX idx_adresse (adresse);

-- Index avancé
CREATE UNIQUE INDEX idx_clients_adresse
USING BTREE ON clients (adresse DESC);
```

### 📌 Résumé :

| Objectif | Astuce |
|---|---|
| Réduire les données | Utiliser SELECT colonne1, colonne2 |
| Réduire le temps d'exécution | Utiliser EXPLAIN, les index, éviter les fonctions dans WHERE |
| Éviter doublons | DISTINCT, UNIQUE si indispensable |
| Protéger et accélérer | Requêtes préparées (PREPARE, EXECUTE) |
| Améliorer la recherche | Utiliser FULLTEXT pour les recherches textuelles |

## Qu'est-ce qu'une exception en MySQL ?

Une exception est une erreur qui se produit pendant l'exécution d'un bloc SQL (procédure, fonction ou trigger). Plutôt que de laisser le programme s'arrêter brutalement lorsqu'une erreur survient, on peut prévoir et gérer ces erreurs à l'aide d'un mécanisme appelé handler.

## Pourquoi gérer les exceptions ?

Sans gestion des exceptions :

- La première erreur (par exemple, champ NOT NULL laissé vide) arrête immédiatement l'exécution du programme.
- Cela rend le programme fragile et peu fiable.

Avec une gestion des exceptions :

- On peut intercepter l'erreur, exécuter du code personnalisé (message d'erreur, alternative, journalisation, etc.) et continuer ou arrêter proprement.

## Syntaxe de base

```
DECLARE [CONTINUE | EXIT] HANDLER FOR [condition] instruction;
```

Les composants :

- action :
  - CONTINUE : le programme continue après l'erreur.
  - EXIT : le programme quitte immédiatement le bloc BEGIN...END.
  - UNDO *(rarement utilisé)* : annule les changements effectués.
- condition :
  - Code d'erreur MySQL (1048, 1062, etc.).
  - SQLSTATE : code d'état standard.
  - NOT FOUND, SQLWARNING, SQLEXCEPTION.

## Exemple 1 : Gestion d'une erreur avec EXIT (champ NOT NULL)

On veut insérer un client, mais le champ nom est NOT NULL.

```
DELIMITER //
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;

 DECLARE EXIT HANDLER FOR 1048 -- Code erreur "Column 'nom' cannot be null"
   SET flagNOTNULL := -1;

 INSERT INTO clients (nom, prenom, adresse)
 VALUES (p_nom, p_prenom, p_adresse);

 SELECT 'Le client est ajouté avec succès';

 IF flagNOTNULL THEN
   SELECT 'Le champ nom doit être non null' AS message;
 END IF;
END;
//
DELIMITER ;
```

Appel :

```
CALL ps_ajouter_client_Exception1(NULL, 'Saloua', 'Safi');
```

Résultat :

```
Le champ nom doit être non null
```

## Exemple 2 : Gestion avec NOT FOUND (aucun résultat trouvé)

On sélectionne un client par nom. Si aucun client n'est trouvé, l'exception est levée (NOT FOUND).

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

Appel :

```
CALL ps_exc_not_found_Exemple('Dalaj');
```

Résultat possible :

```
Il n'y a pas de client avec le nom Dalaj
```

## Résumé des mots-clés d'exception

| Terme MySQL | Description |
|---|---|
| EXIT | Quitte le bloc SQL si l'exception se produit. |
| CONTINUE | Continue l'exécution après la gestion de l'exception. |
| NOT FOUND | Utilisé souvent avec les curseurs ou SELECT INTO. |
| SQLWARNING | Pour les erreurs SQL commençant par 01. |
| SQLEXCEPTION | Pour les autres erreurs non couvertes par les deux ci-dessus. |
| SQLSTATE | Code standard des erreurs SQL (ex. 23000, 02000). |

## Remarques importantes

- Chaque DECLARE doit être écrit au début du bloc BEGIN...END.
- Toujours utiliser DELIMITER quand tu crées des procédures stockées dans le client MySQL.

## Exception FOR SQLEXCEPTION en MySQL

1. Déclaration de la procédure autreErreur

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

Cette procédure affiche simplement un message si une erreur SQL survient dans une autre procédure.

2. Procédure principale avec gestion d'erreur SQL générique

```
CREATE PROCEDURE handlerdemoSQLEXCEPTION ()
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION CALL autreErreur();

  INSERT INTO table_inexistante VALUES (1);  -- Provoque une erreur
  SET @x = 99;  -- Ne sera pas exécutée si erreur
END;
```

3. Appel et affichage

```
CALL handlerdemoSQLEXCEPTION();
SELECT @x;
CALL handlerdemoSQLEXCEPTION(); exécute la procédure.
SELECT @x;
retournera NULL car @x = 99 n'a jamais été exécuté.
```

Résultat attendu

```
Une autre erreur est survenue
NULL
```