Funciones y Triggers

En MySQL, las funciones son bloques de código reutilizables que devuelven un valor a partir de una entrada. Existen dos tipos principales de funciones en MySQL:

1. **Funciones internas (Built-in Functions)**: Son funciones predefinidas que MySQL ofrece para manipulación de datos.
2. **Funciones definidas por el usuario (User-defined Functions - UDFs)**: Son funciones personalizadas que los usuarios pueden crear con `CREATE FUNCTION`.

1️⃣ Funciones Internas en MySQL

Son las funciones que MySQL ya tiene definidas y se pueden utilizar directamente en consultas SQL. Se dividen en varias categorías:

🔹 Funciones de cadena (String Functions)

Estas funciones permiten manipular cadenas de texto:

```
sqlCopiarEditarSELECT CONCAT('Hola', ' ', 'Mundo');  -- Resultado: 'Hola Mundo'
SELECT LENGTH('MySQL');               -- Resultado: 5
SELECT SUBSTRING('Hola Mundo', 1, 4); -- Resultado: 'Hola'
SELECT LOWER('MYSQL');                -- Resultado: 'mysql'
SELECT UPPER('mysql');                -- Resultado: 'MYSQL'
```

🔹 Funciones matemáticas (Mathematical Functions)

Son funciones para cálculos matemáticos:

```
sqlCopiarEditarSELECT ABS(-10);          -- Resultado: 10
SELECT ROUND(3.1416, 2);  -- Resultado: 3.14
SELECT CEIL(3.1);         -- Resultado: 4
SELECT FLOOR(3.9);        -- Resultado: 3
SELECT RAND();            -- Resultado: número aleatorio entre 0 y 1
```

🔹 Funciones de fecha y hora (Date and Time Functions)

Permiten trabajar con valores de fecha y hora:

```
sqlCopiarEditarSELECT NOW();            -- Devuelve la fecha y hora actual
SELECT CURDATE();        -- Devuelve la fecha actual
SELECT CURTIME();        -- Devuelve la hora actual
SELECT YEAR(NOW());      -- Devuelve el año actual
SELECT DATE_ADD(NOW(), INTERVAL 7 DAY); -- Suma 7 días a la fecha actual
```

🔹 Funciones de agregación (Aggregate Functions)

Se usan con `GROUP BY` para resumir datos:

```
sqlCopiarEditarSELECT COUNT(*) FROM empleados;  -- Cuenta registros en la tabla empleados
SELECT SUM(salario) FROM empleados;  -- Suma de todos los salarios
SELECT AVG(salario) FROM empleados;  -- Promedio de los salarios
SELECT MAX(salario) FROM empleados;  -- Máximo salario
SELECT MIN(salario) FROM empleados;  -- Mínimo salario
```

2️⃣ Funciones Definidas por el Usuario (UDFs)

Las **Funciones Definidas por el Usuario (UDFs)** en MySQL permiten encapsular lógica personalizada en una función reutilizable que devuelve un valor. Son útiles cuando necesitas realizar cálculos o transformaciones que no están cubiertas por las funciones predefinidas de MySQL.

Sintaxis

```
CREATE FUNCTION nombre_funcion(parametro1 TIPO_DATO, parametro2 TIPO_DATO, ...)
RETURNS TIPO_DATO
[DETERMINISTIC | NOT DETERMINISTIC]
[CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA]
BEGIN
    -- Declaraciones y lógica de la función
    RETURN valor;
END;
```

📌 **Explicación de la sintaxis**:

- `CREATE FUNCTION nombre_funcion(...)` → Define el nombre de la función y sus parámetros.
- `RETURNS TIPO_DATO` → Especifica el tipo de dato que devolverá la función.
- `[DETERMINISTIC | NOT DETERMINISTIC]:`
  - `DETERMINISTIC`: La función siempre devuelve el mismo resultado si la entrada es la misma.
  - `NOT DETERMINISTIC`: Puede devolver valores diferentes para la misma entrada (Ej: usar `NOW()` o `RAND()`).
- `[CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA]:`
  - `CONTAINS SQL`: Usa SQL, pero no lee ni modifica datos.
  - `NO SQL`: No usa SQL (poco común en MySQL).
  - `READS SQL DATA`: Solo lee datos (sin modificar).
  - `MODIFIES SQL DATA`: Modifica datos (pero MySQL **no permite modificar tablas en funciones**).

🔹 **Ejemplo 1: Calcular Impuesto sobre un Salario**

```
DELIMITER //

CREATE FUNCTION calcular_impuesto(salario DECIMAL(10,2)) RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
    DECLARE impuesto DECIMAL(10,2);
    SET impuesto = salario * 0.15;
    RETURN impuesto;
END //

DELIMITER ;
```

📌 **Uso de la función**:

```
SELECT calcular_impuesto(5000); -- Resultado: 750.00
```

🔹 **Ejemplo 2: Calcular Edad a partir de una Fecha de Nacimiento**

```
DELIMITER //

CREATE FUNCTION calcular_edad(fecha_nacimiento DATE) RETURNS INT
DETERMINISTIC
BEGIN
    RETURN TIMESTAMPDIFF(YEAR, fecha_nacimiento, CURDATE());
END //

DELIMITER ;
```

🔹 Ejemplo 3: Obtener el Nombre Completo de un Usuario

Se tiene una tabla `usuarios` con las columnas `nombre` y `apellido`, y se requiere una función que devuelva el nombre completo.

```
DELIMITER //

CREATE FUNCTION obtener_nombre_completo(id_usuario INT) RETURNS VARCHAR(255)
DETERMINISTIC
READS SQL DATA
BEGIN
    DECLARE nombre_completo VARCHAR(255);

    SELECT CONCAT(nombre, ' ', apellido) INTO nombre_completo
    FROM usuarios
    WHERE id = id_usuario;

    RETURN nombre_completo;
END //

DELIMITER ;
```

```
SELECT obtener_nombre_completo(3); -- Devuelve el nombre completo del usuario con ID 3
```

🔹 Cuándo Usar UDFs en MySQL

✅ Casos donde se recomienda usar funciones UDFs

1. **Cuando necesitas encapsular lógica repetitiva**:
   - Ejemplo: Convertir temperaturas entre Fahrenheit y Celsius en varias consultas.
2. **Cuando la función devuelve un solo valor escalar**:
   - Ejemplo: Calcular impuestos, comisiones, bonificaciones.
3. **Cuando la función realiza cálculos matemáticos o formatea datos**:
   - Ejemplo: Redondear valores, calcular porcentajes o aplicar descuentos.
4. **Cuando la lógica no implica modificaciones en la base de datos**:
   - MySQL **no permite modificar datos dentro de funciones**. Si necesitas insertar, actualizar o eliminar registros, usa un `PROCEDURE` en lugar de una `FUNCTION`.
5. **Cuando necesitas utilizar la función dentro de un** `SELECT`:
   - Las funciones se pueden usar dentro de consultas, lo que las hace ideales para cálculos en tiempo de ejecución.

❌ Cuándo NO usar UDFs

1. **Si necesitas modificar la base de datos** (`INSERT`, `UPDATE`, `DELETE`):
   - MySQL **no permite modificar datos dentro de una función**. Para estos casos, usa **Procedimientos Almacenados**.
2. **Si el resultado depende de múltiples registros**:
   - Las funciones deben devolver un solo valor. Si necesitas devolver múltiples registros, usa una consulta `SELECT` o un `PROCEDURE`.
3. **Si la función es demasiado compleja y afecta el rendimiento**:
   - Consultas con muchas funciones pueden volverse lentas. A veces es mejor hacer el cálculo en la aplicación y no en MySQL.

🔹 Diferencias entre Funciones (UDFs) y Procedimientos Almacenados

Característica	**Funciones (`CREATE FUNCTION`)**	**Procedimientos (`CREATE PROCEDURE`)**
Devuelve un valor	Sí, siempre devuelve un solo valor	No devuelve valor por defecto, pero puede usar `OUT`
Uso en consultas	Se puede usar en `SELECT`	No se puede usar dentro de `SELECT`
Puede modificar datos	❌ No puede modificar datos en tablas	✅ Puede ejecutar `INSERT`, `UPDATE`, `DELETE`

Característica	Funciones (`CREATE FUNCTION`)	Procedimientos (`CREATE PROCEDURE`)
Puede devolver múltiples registros	❌ No	✅ Sí (con `CURSOR` o `SELECT`)

Taller de Funciones Definidas por el Usuario (UDFs) en MySQL 🚀

📌 Introducción

En este taller, desarrollarás varias funciones definidas por el usuario (UDFs) en MySQL para resolver problemas comunes en bases de datos. Aprenderás a crear funciones, utilizarlas en consultas y comprender en qué casos son útiles.

Cada ejercicio presenta un caso de estudio con datos y requerimientos específicos. ¡Ponte a prueba y mejora tus habilidades en SQL! 💡

🔹 Caso 1: Cálculo de Bonificaciones de Empleados

Escenario:

La empresa **TechCorp** otorga bonificaciones a sus empleados según su salario base. La bonificación se calcula así:

- Si el salario es menor a 2,000 USD → Bonificación del **10%**.
- Si el salario está entre 2,000 y 5,000 USD → Bonificación del **7%**.
- Si el salario es mayor a 5,000 USD → Bonificación del **5%**.

Tarea:

1. Crea una función llamada `calcular_bonificacion` que reciba un `DECIMAL(10,2)` con el salario y devuelva la bonificación correspondiente.
2. Usa la función en un `SELECT` sobre la tabla `empleados`.

```
CREATE TABLE empleados (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(50),
    salario DECIMAL(10,2)
);

INSERT INTO empleados (nombre, salario) VALUES
('Juan Pérez', 1500.00),
('Ana Gómez', 3000.00),
('Carlos Ruiz', 6000.00);
```

🔹 Caso 2: Cálculo de Edad de Clientes

Escenario:

En la empresa **MarketShop**, se necesita calcular la edad de los clientes a partir de su fecha de nacimiento para determinar estrategias de marketing.

Tarea:

1. Crea una función llamada `calcular_edad` que reciba un `DATE` y devuelva la edad del cliente.
2. Usa la función en un `SELECT` sobre la tabla `clientes`.

```
CREATE TABLE clientes (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(50),
    fecha_nacimiento DATE
);

INSERT INTO clientes (nombre, fecha_nacimiento) VALUES
('Luis Martínez', '1990-06-15'),
('María López', '1985-09-20'),
('Pedro Gómez', '2000-03-10');
```

🔹 Caso 3: Formatear Números de Teléfono

Escenario:

En **CallCenter Solutions**, los números de teléfono están almacenados sin formato y se necesita presentarlos en el formato `(XXX) XXX-XXXX`.

Tarea:

1. Crea una función llamada `formatear_telefono` que reciba un número de teléfono en formato `XXXXXXXXXX` y lo devuelva en formato `(XXX) XXX-XXXX`.
2. Usa la función en un `SELECT` sobre la tabla `contactos`.

🔹 Caso 4: Clasificación de Productos por Precio

Escenario:

En la tienda **E-Shop**, los productos se categorizan en tres niveles según su precio:

- **Bajo**: Menos de 50 USD.
- **Medio**: Entre 50 y 200 USD.
- **Alto**: Más de 200 USD.

Tarea:

1. Crea una función llamada `clasificar_precio` que reciba un `DECIMAL(10,2)` y devuelva un `VARCHAR(10)` con la clasificación del producto (`Bajo`, `Medio`, `Alto`).
2. Usa la función en un `SELECT` sobre la tabla `productos`.

`CASE`, `LOOPS` y `CURSOR` en Funciones de MySQL

🔹 Uso de `CASE` en Funciones

El `CASE` en MySQL es similar a un `switch` en otros lenguajes. Permite evaluar condiciones y devolver diferentes resultados según el valor de una expresión.

Ejemplo: Clasificar una Nota

Supongamos que tenemos una tabla `estudiantes` con calificaciones y queremos una función que devuelva la clasificación de la nota:

Rango de Nota	Clasificación
0 - 59	Reprobado
60 - 79	Regular
80 - 89	Bueno
90 - 100	Excelente

```
DELIMITER //

CREATE FUNCTION clasificar_nota(nota INT) RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN
    DECLARE resultado VARCHAR(20);

    SET resultado = CASE
        WHEN nota BETWEEN 0 AND 59 THEN 'Reprobado'
        WHEN nota BETWEEN 60 AND 79 THEN 'Regular'
        WHEN nota BETWEEN 80 AND 89 THEN 'Bueno'
        WHEN nota BETWEEN 90 AND 100 THEN 'Excelente'
        ELSE 'Nota inválida'
    END;

    RETURN resultado;
END //

DELIMITER ;
```

```
SELECT clasificar_nota(85); -- Resultado: 'Bueno'
SELECT clasificar_nota(45); -- Resultado: 'Reprobado'
```

🔹 Uso de `LOOP` en Funciones

Los `LOOPS` permiten repetir un bloque de código dentro de una función. Sin embargo, en MySQL **las funciones no pueden contener loops**. 😳

🔹 Uso de `CURSOR` en Funciones

Los `CURSOR` se usan en MySQL para recorrer múltiples filas de una consulta dentro de procedimientos almacenados. Pero al igual que `LOOPS`, **no se pueden usar en funciones**. 😕

💡 **Alternativa**: Usar `CURSOR` en un **Procedimiento Almacenado**. Ejemplo:

Ejemplo: Recorrer empleados y calcular aumento

```
DELIMITER //

CREATE PROCEDURE aumentar_salario(porcentaje DECIMAL(5,2))
BEGIN
    DECLARE fin INT DEFAULT 0;
    DECLARE emp_id INT;
    DECLARE emp_salario DECIMAL(10,2);

    -- Cursor para seleccionar empleados
    DECLARE cur CURSOR FOR SELECT id, salario FROM empleados;

    -- Controlador de final de cursor
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin = 1;

    OPEN cur;

    loop_label: LOOP
        FETCH cur INTO emp_id, emp_salario;
        IF fin = 1 THEN
            LEAVE loop_label;
        END IF;

        -- Actualizar salario
        UPDATE empleados SET salario = emp_salario * (1 + porcentaje / 100) WHERE
id = emp_id;
    END LOOP;

    CLOSE cur;
END //

DELIMITER ;
```

📌 Conclusión

Estructura	¿Se puede usar en una Función?	Alternativa
`CASE`	✅ Sí	Se usa para condicionales dentro de la función.
`LOOP`	❌ No	Usar `WHILE` o Procedimientos Almacenados.
`CURSOR`	❌ No	Usar Procedimientos Almacenados para recorrer filas.

Si necesitas estructuras iterativas (`LOOP`, `WHILE`, `CURSOR`), **usa procedimientos almacenados**. Pero si solo necesitas hacer cálculos y retornar un valor, las **funciones con `CASE` son la mejor opción**. 🚀

TRIGGERS en MySQL

Los **triggers (disparadores)** en MySQL son objetos de base de datos que se ejecutan automáticamente cuando ocurre un evento (`INSERT`, `UPDATE`, `DELETE`) en una tabla.

📌 **Sintaxis de un Trigger en MySQL**

```
CREATE TRIGGER nombre_trigger
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON nombre_tabla
FOR EACH ROW
BEGIN
    -- Cuerpo del trigger (acciones que ejecutará)
END;
CREATE TRIGGER nombre_trigger
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON nombre_tabla
FOR EACH ROW
BEGIN
    -- Cuerpo del trigger (acciones que ejecutará)
END;
```

🔹 **Explicación**:

- `CREATE TRIGGER nombre_trigger` → Define el nombre del trigger.
- `{BEFORE | AFTER}` → Define si se ejecuta antes (`BEFORE`) o después (`AFTER`) del evento.
- `{INSERT | UPDATE | DELETE}` → Especifica el evento que activará el trigger.
- `ON nombre_tabla` → Indica en qué tabla se aplica el trigger.
- `FOR EACH ROW` → Se ejecuta **por cada fila afectada** por el evento.
- `BEGIN ... END` → Contiene el código que se ejecutará.

🔹 Ejemplo 1: Auditoría con un Trigger `AFTER INSERT`

Escenario:

Queremos registrar cada vez que un usuario es agregado a la tabla `usuarios`, guardando el ID, la fecha y la acción en una tabla `auditoria`.

```sql
CREATE TABLE usuarios (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(50),
    email VARCHAR(80)
);

CREATE TABLE auditoria (
    id INT PRIMARY KEY AUTO_INCREMENT,
    accion VARCHAR(50),
    id_usuario INT,
    fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Trigger

```sql
DELIMITER //

CREATE TRIGGER after_usuario_insert
AFTER INSERT ON usuarios
FOR EACH ROW
BEGIN
    INSERT INTO auditoria (accion, id_usuario)
    VALUES ('INSERT', NEW.id);
END //

DELIMITER ;
```

`AFTER INSERT ON usuarios` → Se ejecuta después de un `INSERT` en `usuarios`.

`NEW.id` → Hace referencia al **nuevo ID insertado** en `usuarios`.

Insertar Datos y Verificar la Auditoría

```sql
INSERT INTO usuarios (nombre, email) VALUES ('Juan Pérez', 'juan@example.com');
INSERT INTO usuarios (nombre, email) VALUES ('Ana López', 'ana@example.com');

SELECT * FROM auditoria;
```

🔹 Ejemplo 2: Validación con un Trigger `BEFORE INSERT`

Escenario:

Queremos evitar que se inserten productos con un precio negativo en la tabla `productos`.

```sql
CREATE TABLE productos (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(50),
    precio DECIMAL(10,2)
);
```

Crear el Trigger

```
DELIMITER //

CREATE TRIGGER before_producto_insert
BEFORE INSERT ON productos
FOR EACH ROW
BEGIN
    IF NEW.precio < 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'El precio no puede ser negativo';
    END IF;
END //
DELIMITER ;
```

`BEFORE INSERT` → Se ejecuta antes de insertar un producto.

`NEW.precio < 0` → Valida si el precio es negativo.

`SIGNAL SQLSTATE '45000'` → Genera un error y **evita la inserción**.

```
INSERT INTO productos (nombre, precio) VALUES ('Teclado', 50.00);  -- ✅ Se inserta bien
INSERT INTO productos (nombre, precio) VALUES ('Monitor', -100.00); -- ❌ Genera error
```

🔹 Ejemplo 3: Control de Cambios con un Trigger `BEFORE UPDATE`

Escenario:

Queremos registrar cada vez que se cambia el salario de un empleado en la tabla `historial_salarios`.

```
CREATE TABLE empleados (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(60),
    salario DECIMAL(10,2)
);

CREATE TABLE historial_salarios (
    id INT PRIMARY KEY AUTO_INCREMENT,
    id_empleado INT,
    salario_anterior DECIMAL(10,2),
    salario_nuevo DECIMAL(10,2),
    fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
DELIMITER //

CREATE TRIGGER before_salario_update
BEFORE UPDATE ON empleados
FOR EACH ROW
BEGIN
    INSERT INTO historial_salarios (id_empleado, salario_anterior, salario_nuevo)
    VALUES (OLD.id, OLD.salario, NEW.salario);
END //

DELIMITER ;
```

`BEFORE UPDATE` → Se ejecuta antes de actualizar el salario.

`OLD.salario` → Guarda el salario **antes** de la actualización.

`NEW.salario` → Guarda el salario **después** de la actualización.

Probar el Trigger

```
INSERT INTO empleados (nombre, salario) VALUES ('Carlos López', 2500.00);
UPDATE empleados SET salario = 3000.00 WHERE id = 1;
SELECT * FROM historial_salarios;
```

📌 Resumen de Tipos de Triggers en MySQL

Tipo	Descripción	Usos Comunes
BEFORE INSERT	Se ejecuta antes de un `INSERT`	Validaciones, ajustes de datos
AFTER INSERT	Se ejecuta después de un `INSERT`	Auditoría, creación de registros relacionados
BEFORE UPDATE	Se ejecuta antes de un `UPDATE`	Registro de cambios, validaciones
AFTER UPDATE	Se ejecuta después de un `UPDATE`	Auditoría, cálculo de totales
BEFORE DELETE	Se ejecuta antes de un `DELETE`	Validaciones, restricciones de eliminación
AFTER DELETE	Se ejecuta después de un `DELETE`	Registro de eliminaciones en una tabla de auditoría

🚀 Taller de Triggers en MySQL

📌 Objetivo

En este taller, aprenderás a utilizar **Triggers** en MySQL a través de casos prácticos. Implementarás triggers para validaciones, auditoría de cambios y registros automáticos.

🔹 Caso 1: Control de Stock de Productos

Escenario:

Una tienda en línea necesita asegurarse de que los clientes no puedan comprar más unidades de un producto del stock disponible. Si intentan hacerlo, la compra debe **bloquearse**.

Tarea:

1. Crear las tablas `productos` y `ventas`.
2. Implementar un trigger `BEFORE INSERT` para evitar ventas con cantidad mayor al stock disponible.
3. Probar el trigger.

```
CREATE TABLE productos (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(50),
    stock INT
);

CREATE TABLE ventas (
    id INT PRIMARY KEY AUTO_INCREMENT,
    id_producto INT,
    cantidad INT,
    FOREIGN KEY (id_producto) REFERENCES productos(id)
);
```

🔹 Caso 2: Registro Automático de Cambios en Salarios

Escenario:

La empresa **TechCorp** desea mantener un registro histórico de todos los cambios de salario de sus empleados.

Tarea:

1. Crear las tablas `empleados` y `historial_salarios`.
2. Implementar un trigger `BEFORE UPDATE` que registre cualquier cambio en el salario.
3. Probar el trigger.

```
CREATE TABLE empleados (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(50),
    salario DECIMAL(10,2)
);

CREATE TABLE historial_salarios (
    id INT PRIMARY KEY AUTO_INCREMENT,
    id_empleado INT,
```

```
    salario_anterior DECIMAL(10,2),
    salario_nuevo DECIMAL(10,2),
    fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (id_empleado) REFERENCES empleados(id)
);
```

🔹 Caso 3: Registro de Eliminaciones en Auditoría

Escenario:

La empresa **DataSecure** quiere registrar toda eliminación de clientes en una tabla de auditoría para evitar pérdidas accidentales de datos.

Tarea:

1. Crear las tablas `clientes` y `clientes_auditoria`.
2. Implementar un trigger `AFTER DELETE` para registrar los clientes eliminados.
3. Probar el trigger.

```
CREATE TABLE clientes (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(50),
    email VARCHAR(50)
);

CREATE TABLE clientes_auditoria (
    id INT PRIMARY KEY AUTO_INCREMENT,
    id_cliente INT,
    nombre VARCHAR(50),
    email VARCHAR(50),
    fecha_eliminacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

🔹 Caso 4: Restricción de Eliminación de Pedidos Pendientes

Escenario:

En un sistema de ventas, no se debe permitir eliminar pedidos que aún están **pendientes**.

Tarea:

1. Crear las tablas `pedidos`.
2. Implementar un trigger `BEFORE DELETE` para evitar la eliminación de pedidos pendientes.
3. Probar el trigger.

```
CREATE TABLE pedidos (
    id INT PRIMARY KEY AUTO_INCREMENT,
    cliente VARCHAR(100),
    estado ENUM('pendiente', 'completado')
);
```