ALTHER

Acción	Comando	Ejemplo
Agregar columna	ALTER TABLE nombre_tabla ADD COLUMN nombre_columna tipo_dato [restricciones];	ALTER TABLE empleados ADD COLUMN email VARCHAR(255);
Modificar columna	ALTER TABLE nombre_tabla MODIFY COLUMN nombre_columna tipo_dato [restricciones];	ALTER TABLE empleados MODIFY COLUMN email VARCHAR(100) NOT NULL;
Renombrar columna	ALTER TABLE nombre_tabla CHANGE nombre_columna_antiguo nombre_columna_nuevo tipo_dato;	ALTER TABLE empleados CHANGE COLUMN email correo VARCHAR(100);
Eliminar columna	ALTER TABLE nombre_tabla DROP COLUMN nombre_columna;	ALTER TABLE empleados DROP COLUMN email;
Agregar índice	ALTER TABLE nombre_tabla ADD INDEX (nombre_columna);	ALTER TABLE empleados ADD INDEX (nombre);
Eliminar índice	ALTER TABLE nombre_tabla DROP INDEX nombre_indice;	
Agregar clave foránea	ALTER TABLE nombre_tabla ADD CONSTRAINT nombre_constraint FOREIGN KEY (nombre_columna) REFERENCES otra_tabla(col_ref);	ALTER TABLE pedidos ADD CONSTRAINT fk_cliente FOREIGN KEY (cliente_id) REFERENCES clientes(id);
Eliminar clave foránea	ALTER TABLE nombre_tabla DROP FOREIGN KEY nombre_constraint;	ALTER TABLE pedidos DROP FOREIGN KEY fk_cliente;
Renombrar tabla	ALTER TABLE nombre_tabla_antigua RENAME TO nombre_tabla_nueva;	ALTER TABLE empleados RENAME TO trabajadores;

Cambiar motor de almacenamiento	ALTER TABLE nombre_tabla ENGINE = nuevo_motor;	ALTER TABLE empleados ENGINE = InnoDB;	

TIPOS

Categoría	Tipo de Dato	Descripción	Ejemplo
Numéricos	INT	Número entero con signo (-2,147,483,648 a 2,147,483,647)	edad INT;
	BIGINT	Número entero con signo de mayor tamaño (-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807)	poblacion BIGINT;
	FLOAT	Números de coma flotante (precisión simple)	altura FLOAT;
	DOUBLE	Números de coma flotante (precisión doble)	distancia DOUBLE;
	DECIMAL(p, s)	Número decimal con precisión fija (p = total dígitos, s = dígitos después del punto decimal)	precio DECIMAL(10, 2);
Cadenas	CHAR(n)	Cadena de longitud fija (n especifica la longitud)	codigo CHAR(5);
	VARCHAR(n)	Cadena de longitud variable (n especifica el máximo)	nombre VARCHAR(100);
	TEXT	Cadena de longitud variable más larga, hasta 65,535 caracteres	descripcion TEXT;
Fecha y hora	DATE	Fecha (formato: 'YYYY-MM-DD')	fecha_nacimiento DATE;

	TIME	Hora (formato: 'HH:MM ')	hora_inicio TIME;
	DATETIME	Fecha y hora combinadas (formato: 'YYYY-MM-DD HH:MM ')	registro DATETIME;
	TIMESTAMP	Fecha y hora combinadas, con zona horaria (formato: 'YYYY-MM-DD HH:MM ')	creado TIMESTAMP;
	YEAR	Año en formato de 4 dígitos	ano YEAR;
Lógicos	BOOLEAN	Valores lógicos (TRUE o FALSE, representados como 1 y 0)	es_activo BOOLEAN;
Binarios	BLOB	Almacena datos binarios grandes (imágenes, videos, etc.)	imagen BLOB;
Enumeración	ENUM	Lista de valores posibles. Sólo puede tomar uno de ellos	sexo ENUM('M', 'F');
Conjuntos	SET	Lista de valores posibles, puede contener más de uno al mismo tiempo	opciones SET('A', 'B', 'C');
Autonumérico	AUTO_INCREMENT	Genera un valor numérico incremental automáticamente (normalmente en combinación con PRIMARY KEY)	id INT AUTO_INCREMENT;

TRIGERS

—--SINTAXYS—----

—PARA UNA SOLA INSTRUCIÓN SQL—
```
CREATE TRIGGER nombre_trigger
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON nombre_tabla
FOR EACH ROW
BEGIN
```

```
        -- Instrucciones SQL
END;


—---- PARA MAS DE UNA INSTRUCIÓN SQL—---

DELIMITER $$

CREATE TRIGGER trigger_name
AFTER INSERT ON table_name
FOR EACH ROW
BEGIN
        -- Instrucciones SQL aquí
        UPDATE table_name
        SET column_name = value
        WHERE condition;

        -- Más instrucciones si es necesario
END $$

DELIMITER ;

—----ELIMINAR TRIGER—----
DROP TRIGGER IF EXISTS nombre_trigger;
```

—----EVENTOS—---

Evento	Descripción
INSERT	Se dispara cuando se inserta una fila en la tabla.
UPDATE	Se dispara cuando se actualiza una fila existente.
DELETE	Se dispara cuando se elimina una fila de la tabla.

—----MOMENTOS—------

Momento	Descripción

BEFORE El trigger se ejecuta **antes** de que la operación (INSERT, UPDATE, DELETE) afecte a la tabla.

AFTER El trigger se ejecuta **después** de que la operación ya haya modificado los datos en la tabla.

OPERADORES DE MANIPULACIÓN DE DATOS

Operador	Función	Uso Principal	Sintaxis
INSERT	Inserta nuevas filas en una tabla.	Añadir nuevos registros.	INSERT INTO tabla (columna1, columna2, ...) VALUES (valor1, valor2, ...);
UPDATE	Modifica valores de columnas existentes en una tabla.	Cambiar información de registros existentes.	UPDATE tabla SET columna1 = valor1, columna2 = valor2 WHERE condición;
DELETE	Elimina filas que cumplen una condición.	Borrar registros específicos.	DELETE FROM tabla WHERE condición;
MERGE	Combina INSERT y UPDATE. Actualiza si existe, inserta si no.	Sincronizar tablas o evitar duplicación.	MERGE INTO tabla_destino USING tabla_fuente ON condición WHEN MATCHED THEN UPDATE ...

REPLACE	Inserta nuevas filas, pero reemplaza si hay un conflicto con claves.	Insertar sin duplicar claves.	REPLACE INTO tabla (columna1, columna2, ...) VALUES (valor1, valor2, ...);
TRUNCATE	Elimina todas las filas de una tabla rápidamente, sin condiciones.	Borrar todos los registros de la tabla.	TRUNCATE TABLE nombre_tabla;
SELECT	Recupera datos de una tabla.	Consultar información almacenada.	SELECT columna1, columna2 FROM tabla WHERE condición;
CALL	Ejecuta un procedimiento almacenado (stored procedure).	Ejecutar procedimientos almacenados.	CALL nombre_procedimiento();
LOCK	Bloquea filas para evitar que otros usuarios las modifiquen.	Controlar acceso concurrente a los datos.	LOCK TABLE tabla IN modo_bloqueo;

CREAR TABLAS

Sintaxis	Descripción
Crear tabla básica	CREATE TABLE nombre_tabla (columna1 tipo_dato1, columna2 tipo_dato2, ...); Crea una tabla sin restricciones adicionales.
Con clave primaria	CREATE TABLE nombre_tabla (columna1 tipo_dato1 PRIMARY KEY, columna2 tipo_dato2); Define una columna como clave primaria, asegurando la unicidad de sus valores.
Con clave primaria compuesta	CREATE TABLE nombre_tabla (columna1 tipo_dato1, columna2 tipo_dato2, PRIMARY KEY (columna1, columna2)); Establece una clave primaria que abarca más de una columna.

Con clave única	CREATE TABLE nombre_tabla (columna1 tipo_dato1, columna2 tipo_dato2 UNIQUE); Asegura que los valores en una columna sean únicos.
Con valor por defecto	CREATE TABLE nombre_tabla (columna1 tipo_dato1 DEFAULT valor, columna2 tipo_dato2); Establece un valor predeterminado para la columna si no se especifica uno al insertar.
Con auto incremento	CREATE TABLE nombre_tabla (columna1 INT AUTO_INCREMENT PRIMARY KEY, columna2 tipo_dato2); Incrementa automáticamente el valor de la columna con cada nueva inserción.
Con restricciones NOT NULL	CREATE TABLE nombre_tabla (columna1 tipo_dato1 NOT NULL, columna2 tipo_dato2); Asegura que la columna no pueda contener valores nulos.
Con clave foránea	CREATE TABLE nombre_tabla (columna1 tipo_dato1, columna2 INT, FOREIGN KEY (columna2) REFERENCES otra_tabla(columna_referenciada)); Establece una relación con otra tabla, asegurando la integridad referencial.
Crear tabla temporal	CREATE TEMPORARY TABLE nombre_tabla (columna1 tipo_dato1, columna2 tipo_dato2); Crea una tabla que solo existe durante la sesión actual.
Crear tabla con opciones de almacenamiento	CREATE TABLE nombre_tabla (columna1 tipo_dato1, columna2 tipo_dato2) ENGINE=InnoDB; Especifica el motor de almacenamiento a utilizar para la tabla.

PROCEDIMIENTOS

```
DELIMITER $$

CREATE PROCEDURE modifyData(IN input_value INT, OUT output_value INT, INOUT
modify_value INT)
BEGIN
        SET output_value = input_value * 2;
        SET modify_value = modify_value + 10;
END $$
```

DELIMITER ;

Concepto	Descripción	Sintaxis/Ejemplo
Procedimiento Almacenado	Un bloque de código SQL almacenado en el servidor, que se puede ejecutar en cualquier momento.	CREATE PROCEDURE nombre_procedimiento ...
Crear un Procedimiento	Define un procedimiento con parámetros opcionales (IN, OUT, INOUT).	DELIMITER $$ CREATE PROCEDURE nombre() BEGIN -- código END $$ DELIMITER ;
Ejecutar un Procedimiento	Llama al procedimiento almacenado.	CALL nombre_procedimiento(parámetros);
IN	Parámetro de entrada: pasa un valor al procedimiento.	IN param_name tipo_dato
OUT	Parámetro de salida: devuelve un valor desde el procedimiento.	OUT param_name tipo_dato
INOUT	Parámetro de entrada y salida: pasa un valor al procedimiento, y lo puede modificar.	INOUT param_name tipo_dato
Ejemplo de Procedimiento con Parámetros	Procedimiento con parámetros IN, OUT, y INOUT.	DELIMITER $$ CREATE PROCEDURE modifyData(IN val INT, OUT res INT, INOUT modif INT) BEGIN SET res = val * 2; SET modif = modif + 10; END $$ DELIMITER ;
Condicionales IF	Estructura condicional que ejecuta código si una condición es verdadera.	IF condición THEN -- código END IF;
Ejemplo de IF	Condicional que evalúa si el stock es mayor que 0.	IF stock > 0 THEN SET status = 'In Stock'; ELSE SET status = 'Out of Stock'; END IF;

Condicionales CASE	Ejecuta diferentes bloques de código dependiendo de un valor o expresión.	CASE WHEN condición THEN -- código ELSE -- código END CASE;
Ejemplo de CASE	Evalúa una columna para definir el estado.	CASE WHEN age > 18 THEN 'Adult' WHEN age > 12 THEN 'Teenager' ELSE 'Child' END CASE;
Bucles WHILE	Repite un bloque de código mientras una condición sea verdadera.	WHILE condición DO -- código END WHILE;
Ejemplo de WHILE	Bucle que cuenta hacia atrás.	DECLARE counter INT DEFAULT 10; WHILE counter > 0 DO SELECT counter; SET counter = counter - 1; END WHILE;
Manejo de Errores	Usa DECLARE HANDLER para manejar excepciones o errores en el procedimiento.	DECLARE CONTINUE HANDLER FOR SQLEXCEPTION -- código
Transacciones	Los procedimientos pueden manejar transacciones usando START TRANSACTION, COMMIT, y ROLLBACK.	START TRANSACTION; -- código COMMIT;
Eliminar un Procedimiento	Para actualizar un procedimiento, primero debes eliminarlo y luego crearlo nuevamente.	DROP PROCEDURE IF EXISTS nombre_procedimiento;
Consultar Procedimientos	Ver los procedimientos almacenados en la base de datos.	SELECT * FROM information_schema.ROUTINES WHERE ROUTINE_TYPE = 'PROCEDURE';
Declarar Variables Locales	Usa DECLARE para declarar variables locales dentro del procedimiento.	DECLARE var_name tipo_dato;

Procedimientos Almacenados en MySQL

Un **procedimiento almacenado** es un conjunto de instrucciones SQL que se almacenan en el servidor de bases de datos y pueden ser reutilizadas. A diferencia de las funciones, los procedimientos pueden no devolver un valor y pueden realizar operaciones más complejas como insertar, actualizar, o eliminar registros.

Características clave de los procedimientos almacenados:

1. **No siempre devuelven un valor**: A diferencia de las funciones, un procedimiento almacenado puede devolver múltiples resultados o no devolver nada.
2. **Pueden modificar datos**: Los procedimientos almacenados pueden realizar operaciones `INSERT`, `UPDATE`, `DELETE`, entre otras.
3. **Pueden aceptar múltiples tipos de parámetros**:
    - `IN`: Parámetros de entrada, usados para recibir valores.
    - `OUT`: Parámetros de salida, usados para devolver valores.
    - `INOUT`: Parámetros que pueden tanto recibir como devolver valores.
4. **Reutilización**: Al igual que las funciones, se pueden llamar repetidamente para realizar las mismas tareas en diferentes escenarios.

Sintaxis básica de un procedimiento almacenado

sql

```
CREATE PROCEDURE procedure_name([parameters])
BEGIN
    -- cuerpo del procedimiento
END;
```

Partes de la sintaxis:

- **`CREATE PROCEDURE procedure_name`**: Define el nombre del procedimiento.
- **`parameters`**: Lista de parámetros opcionales, cada uno con un tipo (pueden ser `IN`, `OUT` o `INOUT`).
- **`BEGIN...END`**: Delimita el bloque de código del procedimiento donde se colocan las operaciones que se realizarán.

Parámetros de los procedimientos almacenados

- **IN**: Los parámetros de entrada reciben valores cuando se llama al procedimiento. No se pueden modificar dentro del procedimiento.
  sql

```
DELIMITER $$

CREATE TRIGGER update_stock_after_insert
AFTER INSERT ON orders
FOR EACH ROW
BEGIN
    UPDATE products
    SET stock = stock - NEW.quantity
    WHERE product_id = NEW.product_id;
END$$

DELIMITER ;
```

En este caso, cuando se inserta un nuevo pedido en la tabla `orders`, el trigger se activa y actualiza el stock del producto en la tabla `products`.

---

2. Trigger `BEFORE UPDATE`: Validar los cambios antes de actualizar.

sql

```
DELIMITER $$

CREATE TRIGGER validate_salary_before_update
BEFORE UPDATE ON employees
FOR EACH ROW
BEGIN
    IF NEW.salary < 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Salary cannot be negative';
    END IF;
END$$

DELIMITER ;
```

Este trigger verifica que el salario de un empleado no sea negativo antes de actualizar el valor. Si es negativo, lanza un error.

---

3. Trigger `AFTER DELETE`: Insertar en un historial cuando se elimina un registro.

sql

```
DELIMITER $$

CREATE TRIGGER log_deletion_after_delete
AFTER DELETE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO deletion_log(employee_id, deleted_at)
    VALUES (OLD.employee_id, NOW());
END$$

DELIMITER ;
```

Este trigger inserta un registro en la tabla `deletion_log` cada vez que se elimina un empleado en la tabla `employees`.

Eventos comunes para triggers:

Evento	Descripción
`INSERT`	Se activa cuando se inserta un registro en la tabla.
`UPDATE`	Se activa cuando se actualiza un registro en la tabla.
`DELETE`	Se activa cuando se elimina un registro de la tabla.

Tipos de triggers:

Tipo de Trigger	Descripción
`BEFORE INSERT`	Se ejecuta antes de insertar un registro en la tabla.

`AFTER INSERT`	Se ejecuta después de insertar un registro en la tabla.
`BEFORE UPDATE`	Se ejecuta antes de actualizar un registro en la tabla.
`AFTER UPDATE`	Se ejecuta después de actualizar un registro en la tabla.
`BEFORE DELETE`	Se ejecuta antes de eliminar un registro de la tabla.
`AFTER DELETE`	Se ejecuta después de eliminar un registro de la tabla.

Manejo de errores en *triggers*:

Los triggers pueden usar la instrucción `SIGNAL` para lanzar errores personalizados cuando se detectan condiciones no deseadas.

Ejemplo de manejo de errores:

sql

```
DELIMITER $$

CREATE TRIGGER check_positive_quantity
BEFORE INSERT ON orders
FOR EACH ROW
BEGIN
    IF NEW.quantity <= 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Quantity must be greater than zero';
    END IF;
END$$

DELIMITER ;
```

Consulta de *triggers* existentes:

Puedes consultar los triggers definidos en una base de datos con la siguiente sentencia:

sql

```
SHOW TRIGGERS FROM database_name;
```

Eliminación de un *trigger*:

Para eliminar un trigger existente, utiliza el comando `DROP TRIGGER`:

sql

```
DROP TRIGGER IF EXISTS trigger_name;
```

Consideraciones al usar *triggers* en MySQL:

1. **No puedes modificar la tabla en la que se define el trigger**: No puedes insertar, actualizar o eliminar datos de la misma tabla en la que se activa el trigger, ya que esto puede causar bucles de activación infinita.
2. **No se permiten transacciones**: Los triggers no pueden contener comandos de control de transacciones como `COMMIT` o `ROLLBACK`.
3. **Afecta el rendimiento**: Los triggers pueden ralentizar el rendimiento de la base de datos si no se usan con cuidado, ya que se ejecutan por cada fila afectada por la consulta que los activa.

MANEJADORES

Los manejadores (o handlers) en MySQL son componentes utilizados para gestionar condiciones especiales o errores que pueden surgir durante la ejecución de procedimientos almacenados o triggers. Aquí te ofrezco un resumen de lo que necesitas saber sobre ellos:

1. Definición

Un manejador es una estructura que permite especificar una acción a tomar cuando ocurre una condición específica, como un error o una advertencia.

2. Tipos de Manejadores

- **CONTINUE:** Continúa la ejecución del bloque después de que se activa el manejador.
- **EXIT:** Sale del bloque de código actual (como un bucle o un procedimiento) cuando se activa el manejador.

3. Declaración de Manejadores

Se declaran dentro de un bloque de código (como un procedimiento almacenado o un trigger) utilizando la siguiente sintaxis:

```
DECLARE handler_type HANDLER FOR condition_value BEGIN
   -- Acción a tomar
END;
```

Donde:

- **handler_type** : Puede ser CONTINUE el EXIT.
- **condition_value** : Especifica la condición que activa el manejador (puede ser un error específico, un código de SQLSTATE, o un tipo de señal como SQLWARNING, SQLEXCEPTION, etc.).

4. Ejemplos de Uso

a. Manejador de Errores

```
DELIMITER //

CREATE PROCEDURE example_procedure()
BEGIN
        DECLARE EXIT HANDLER FOR SQLEXCEPTION
        BEGIN
        -- Acción en caso de error
        SELECT 'Se produjo un error.';
        END;

        -- Código que puede causar un error
        INSERT INTO nonexistent_table (column) VALUES ('data');
END //

DELIMITER ;
```

b. **Manejador de Advertencias**

```
DELIMITER //

CREATE PROCEDURE example_procedure()
BEGIN
        DECLARE CONTINUE HANDLER FOR SQLWARNING
        BEGIN
        -- Acción en caso de advertencia
        SELECT 'Se produjo una advertencia.';
        END;

        -- Código que puede causar una advertencia
        INSERT INTO my_table (column) VALUES ('data');
END //

DELIMITER ;
```

5. Consideraciones

- **Orden de Manejadores:** Si hay múltiples manejadores para la misma condición, MySQL ejecutará el primero que encuentre en el orden de declaración.
- **Alcance:** Los manejadores son específicos del bloque en el que se declaran y no afectan a otros bloques.

Cursores

Los cursores en MySQL son una herramienta fundamental para manejar resultados de consultas de forma más controlada y eficiente. Aquí tienes un resumen de todo lo que necesitas saber sobre ellos:

1. Definición

Un cursor es un objeto que permite acceder a las filas de un conjunto de resultados de forma secuencial. Es útil cuando se necesita realizar operaciones fila por fila.

2. Tipos de Cursores

Cursores implícitos: MySQL los crea automáticamente al ejecutar una consulta SELECT, y son utilizados internamente.

Cursores explícitos: Son definidos por el usuario y ofrecen un control más detallado sobre el procesamiento de resultados.

3. Uso de Cursores

Para usar cursores en MySQL, sigue estos pasos:

a. Declarar un Cursor

SQL

```
DECLARE cursor_name CURSOR FOR SELECT statement;
```

Esto define un cursor que se vincula a una consulta SELECT.

b. Abrir el Cursor

SQL

```
OPEN cursor_name;
```

Esto ejecuta la consulta y prepara el cursor para que se puedan recuperar las filas.

c. Leer de un Cursor

Se utilizan las instrucciones FETCH para obtener las filas una a una:

SQL

```
FETCH cursor_name INTO variable1, variable2, ...;
```

Esto almacena los resultados en las variables especificadas.

d. Cerrar el Cursor

SQL

```
CLOSE cursor_name;
```

Esto libera los recursos asociados al cursor.

e. Eliminar el Cursor

SQL

```
DEALLOCATE PREPARE cursor_name;
```

Esto elimina el cursor, aunque en MySQL esto generalmente se maneja automáticamente al cerrar el cursor.

4. Ejemplo Completo

Aquí tienes un ejemplo práctico:

SQL

```
DELIMITER //

CREATE PROCEDURE example_procedure()
BEGIN
	DECLARE done INT DEFAULT FALSE;
	DECLARE var1 INT;
	DECLARE var2 VARCHAR(255);

	DECLARE example_cursor CURSOR FOR SELECT id, name FROM employees;
	DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

	OPEN example_cursor;

	read_loop: LOOP
	FETCH example_cursor INTO var1, var2;
	IF done THEN
	LEAVE read_loop;
	END IF;

	-- Realiza alguna operación con var1 y var2
	SELECT CONCAT('Employee ID: ', var1, ', Name: ', var2);
	END LOOP;

	CLOSE example_cursor;
END //

DELIMITER ;
```

En este procedimiento almacenado, se declara un cursor para obtener los id y name de la tabla employees, y luego se procesan fila por fila.
5. Consideraciones

Rendimiento: Los cursores pueden afectar el rendimiento si se utilizan en grandes conjuntos de datos, ya que procesan fila por fila. Es preferible usar operaciones de conjunto siempre que sea posible.

Bloqueo: Los cursores pueden bloquear filas durante su uso, lo que puede afectar el rendimiento en entornos concurrentes.

6. Referencias

Para más información, consulta la documentación oficial de MySQL sobre cursores y su uso en procedimientos almacenados .

Si necesitas aclaraciones sobre algún punto específico o ejemplos adicionales, házmelo saber.

ROLES

Roles en MySQL

1. **Definición** :
   Un rol es un conjunto de privilegios que se puede asignar a uno o más usuarios. Facilita la gestión de permisos al agruparlos bajo un mismo nombre.
2. **Creación de Roles** :
   Para crear un rol, utiliza el comando:
   CREATE ROLE 'nombre_rol';
3. **Tipos de Privilegios** :
   Los privilegios que se pueden asignar a los roles incluyen, pero no se limitan a:
   - **Privilegios de Administrador** :
     - ALL PRIVILEGES: Todos los privilegios.
     - GRANT OPTION: Permite otorgar o revocar privilegios a otros.
   - **Privilegios sobre Bases de Datos** :
     - CREATE: Crear nuevas bases de datos.
     - DROP: Eliminar bases de datos.
     - ALTER: Modificar bases de datos.
   - **Privilegios sobre Tablas** :
     - SELECT: Leer datos de tablas.
     - INSERT: Insertar datos en tablas.
     - UPDATE: Modificar datos en tablas.
     - DELETE: Eliminar datos de tablas.
     - INDEX: Crear y eliminar índices en tablas.
     - CREATE VIEW: Crear vistas.
     - DROP VIEW: Eliminar
   - **Privilegios sobre Procedimientos Almacenados** :
     - EXECUTE: Ejecutar procedimientos almacenados.
   - **Privilegios de Sistema** :
     - LOCK TABLES: Bloquear tablas para operaciones de lectura y escritura.

4. **Asignación de Privilegios** :
   Para otorgar privilegios a un rol:
   GRANT privilegios ON objeto TO 'nombre_rol';
5. **Asignación de Roles a Usuarios** :
   Para asignar un rol a un usuario:
   GRANT 'nombre_rol' TO 'usuario'@'host';
6. **Activación de Roles** :
   Para activar un rol en una sesión:
   SET ROLE 'nombre_rol';
7. **Verificación de Roles Activos**:
   Para ver los roles activos de un usuario, usa:
   SELECT CURRENT_ROLE();
8. **Revocación de Roles** :
   Para revocar un rol de un usuario:
   REVOKE 'nombre_rol' FROM 'usuario'@'host';
9. **Eliminación de Roles** :
   Para eliminar un rol:
   DROP ROLE 'nombre_rol';
10. **Efecto de la Eliminación** :
    Si un rol es eliminado mientras está activo en un usuario, el rol se desactivará automáticamente y los privilegios asociados se perderán para ese usuario.
11. **Problemas Comunes** :
    Si un rol eliminado sigue apareciendo como activo, puede
    - Sesiones activas donde el rol estaba asignado.
    - Falta de revocación del rol antes de la eliminación.
    - Problemas de permisos de visualización.

FUNCIONES DE COMPARACIÓN PARA TIPO DATE

Función	Descripción	Uso

FECHA()	Extrae la parte de fecha de una fecha o datetime.	SELECT DATE(column_name) FROM table_name;
FECHADIFF()	Calcula la diferencia en días entre dos fechas.	SELECT DATEDIFF(date1, date2);
FECHA_ADD()	Suma un intervalo de tiempo a una fecha.	SELECT DATE_ADD(date, INTERVAL 10 DAY);
FECHA_SUB()	Resta un intervalo de tiempo de una fecha.	SELECT DATE_SUB(date, INTERVAL 10 DAY);
FECHA_FORMATO()	Formatea una fecha según el formato especificado.	SELECT DATE_FORMAT(date, '%Y-%m-%d');
STR_TO_DATE()	Convierte una cadena en un formato específico a una fecha.	SELECT STR_TO_DATE('30-12-2024', '%d-%m-%Y');
NOW()	Devuelve la fecha y hora actuales.	SELECT NOW();
FECHA ACTUAL()	Devuelve la fecha actual.	SELECT CURDATE();
AGREGAR()	Suma un intervalo de tiempo a una fecha, sin modificar el tiempo.	SELECT ADDDATE(date, INTERVAL 10 DAY);
SUBFECHA()	Resta un intervalo de tiempo de una fecha, sin modificar el tiempo.	SELECT SUBDATE(date, INTERVAL 10 DAY);

FUNCIONES DE AGREGACIÓN

Función	Descripción	Uso
CONTAR()	Cuenta el número de filas en un conjunto de resultados.	SELECT COUNT(column_name) FROM table_name;
SUMA()	Calcula la suma total de una columna numérica.	SELECT SUM(column_name) FROM table_name;
AVG()	Calcula el promedio de una columna numérica.	SELECT AVG(column_name) FROM table_name;
MÍN()	Devuelve el valor mínimo de una columna.	SELECT MIN(column_name) FROM table_name;
MÁXIMO()	Devuelve el valor máximo de una columna.	SELECT MAX(column_name) FROM table_name;
GRUPO_CONCAT()	Concatenar valores de una columna en una sola cadena.	SELECT GROUP_CONCAT(column_name SEPARATOR ', ') FROM table_name;
CONTAR(DISTINTO)	Cuenta el número de valores distintos en una columna.	SELECT COUNT(DISTINCT column_name) FROM table_name;
DIFERENCIA()	Calcula la varianza de una columna numérica.	SELECT VARIANCE(column_name) FROM table_name;
DESVEST()	Calcula la desviación estándar de una columna numérica.	SELECT STDDEV(column_name) FROM table_name;

Tabla: Procedimientos, Funciones y Triggers en MySQL

Concepto	Descripción

Procedimientos (Procedures)	Un conjunto de sentencias SQL que se almacenan en la base de datos y pueden ejecutarse repetidamente. No necesariamente devuelven un valor.
Funciones (Functions)	Similar a los procedimientos, pero **siempre** devuelven un valor único y se pueden usar en expresiones SQL.
Desencadenantes	Bloques de código que se ejecutan automáticamente en respuesta a eventos **INSERT**, **UPDATE**, o **DELETE** en una tabla.

Sintaxis y Ejemplos

Tipo	Sintaxis	Descripción
Crear un Procedimiento	`CREAR PROCEDIMIENTO nombre_procedimiento ([EN	AFUERA
Llamar un Procedimiento	CALL procedure_name(param1, param2, ...);	Se invoca un procedimiento utilizando el comando CALL. Ejemplo: CALL myProc(5, @resultado);
Crear una Función	CREATE FUNCTION function_name ([params]) RETURNS type BEGIN -- Sentencias RETURN expr; END	Las funciones deben devolver un valor con RETURN. Ejemplo: CREATE FUNCTION get_total() RETURNS INT BEGIN RETURN 100; END;
Llamar una Función	SELECT function_name(params);	Se invoca la función dentro de una consulta. Ejemplo: SELECT get_total();
Crear un Trigger	`CREAR DISPARADOR nombre_disparador ANTES	DESPUÉS DE INSERTAR

FUNCIONES

Funciones en MySQL

Las **funciones** en MySQL son bloques de código reutilizables que realizan una tarea específica y devuelven un valor. Son similares a los procedimientos almacenados, pero la diferencia clave es que las funciones siempre retornan un valor único.

Características clave de las funciones:

1. **Retorno de un valor**: Toda función debe retornar un valor con el tipo de dato definido en la declaración de la función.
2. **Determinísticas vs No Determinísticas**: Una función puede ser determinística (siempre devuelve el mismo resultado para los mismos parámetros) o no determinística.
3. **Uso en consultas**: Las funciones se pueden usar directamente en consultas SQL, al igual que las funciones integradas de MySQL como SUM(), AVG(), etc.
4. **Parámetros de entrada**: Las funciones pueden aceptar parámetros, los cuales se usan dentro del cuerpo de la función para realizar cálculos u operaciones.

Sintaxis de creación de una función

sql

```
CREATE FUNCTION function_name(parameter_list)
RETURNS data_type
[DETERMINISTIC | NOT DETERMINISTIC]
BEGIN
        -- cuerpo de la función
        RETURN valor;
END;
```

Partes de la sintaxis:

- **CREATE FUNCTION function_name**: Define el nombre de la función.
- **parameter_list**: Lista de parámetros de entrada que la función acepta.
- **RETURNS data_type**: Indica el tipo de dato que devolverá la función.
- **DETERMINISTIC | NOT DETERMINISTIC**: Opcional, especifica si la función devuelve siempre el mismo valor para los mismos parámetros (DETERMINISTIC) o si puede devolver valores diferentes (NOT DETERMINISTIC).
- **BEGIN...END**: Define el bloque donde se coloca la lógica de la función.
- **RETURN**: Es obligatorio y especifica el valor que se devolverá como resultado de la función.

Tipos de parámetros

Los parámetros en una función se especifican como IN de manera predeterminada (solo de entrada). A diferencia de los procedimientos, las funciones **no** pueden tener parámetros de tipo OUT o INOUT.

Tipos de funciones en MySQL

1. **Funciones Escalares**: Devuelven un valor único, normalmente usado en SELECT para procesar filas individualmente.
   Ejemplo:
   sql

   ```
   CREATE FUNCTION calculate_total(price DECIMAL(10,2), quantity INT)
   RETURNS DECIMAL(10,2)
   BEGIN
           RETURN price * quantity;
   END;
   ```

1. **unciones Agregadas (Custom)**: Aunque MySQL tiene funciones agregadas como SUM(), puedes crear funciones agregadas personalizadas si necesitas lógica más compleja.

Llamada de una función

Una función se puede llamar directamente desde una consulta SELECT o dentro de otras funciones o procedimientos. Ejemplo:

sql

```
SELECT calculate_total(100.50, 3);
```

Uso de funciones en consultas

Las funciones se pueden usar en las cláusulas SELECT, WHERE, ORDER BY, etc.

Ejemplo en una consulta:

sql

```
SELECT employee_id, calculate_total(salary, bonus) AS total_compensation
FROM employees;
```

Diferencias entre funciones y procedimientos:

Característica	Funciones	Procedimientos
Retorno de valores	**Siempre devuelven un valor**	**No siempre devuelven un valor**
Uso en consultas SQL	**Se pueden usar directamente en consultas**	**No se pueden usar directamente**
Número de valores devueltos	**Un solo valor**	**Pueden devolver varios valores**
IN/OUT parámetros	**Solo admite parámetros IN**	**Admite parámetros IN, OUT, INOUT**

Restricciones de las funciones:

- **No pueden modificar datos: No es posible ejecutar operaciones como INSERT, UPDATE, o DELETE dentro de una función. Solo pueden realizar operaciones de lectura.**
- **Un solo valor de retorno: Las funciones solo pueden devolver un único valor (a diferencia de los procedimientos, que pueden devolver múltiples resultados).**
- **No permiten transacciones: No se pueden iniciar, comprometer o deshacer transacciones dentro de una función.**

LIKE / NOT LIKE

`LIKE` y `NOT LIKE` son operadores utilizados para realizar búsquedas basadas en patrones dentro de los datos. Son útiles cuando se desea filtrar resultados que coincidan (o no) con un formato o una secuencia específica de caracteres.

Sintaxis básica de `LIKE`:

sql

```sql
SELECT column_name
FROM table_name
WHERE column_name LIKE 'pattern';
```

Sintaxis básica de `NOT LIKE`:

sql

```sql
SELECT column_name
FROM table_name
WHERE column_name NOT LIKE 'pattern';
```

Patrones usados con `LIKE`:

- `%`: Representa **cualquier número de caracteres**, incluyendo cero.
  - Ejemplo: `'A%'` coincidirá con cualquier valor que comience con la letra `A`.
- `_`: Representa **un solo carácter**.
  - Ejemplo: `'H_t'` coincidirá con `Hat`, `Hit`, `Hot`, etc.
- `[ ]`: Representa un **rango de caracteres**.
  - Ejemplo: `'H[aeiou]t'` coincidirá con `Hat`, `Hit`, `Hot`, etc.
- `[^ ]` **o** `!`: Representa **cualquier carácter excepto los que están dentro del rango**.
  - Ejemplo: `'H[^aeiou]t'` coincidirá con `Hbt`, `Hct`, etc.

Ejemplos con `LIKE`:

1. **Buscar todos los nombres que empiezan con 'A':**

sql

```sql
SELECT name
FROM users
WHERE name LIKE 'A%';
```

2. **Buscar direcciones de correo electrónico que contienen 'gmail':**

sql

```sql
SELECT email
FROM contacts
WHERE email LIKE '%gmail%';
```

3. **Buscar apellidos que terminen en 'son':**

sql

```sql
SELECT last_name
FROM employees
WHERE last_name LIKE '%son';
```

4. **Buscar números de teléfono que comiencen con 555:**

sql

```sql
SELECT phone_number
FROM directory
WHERE phone_number LIKE '555%';
```

Ejemplos con `NOT LIKE`:

1. **Buscar todos los nombres que no empiecen con 'A':**

sql

```sql
SELECT name
FROM users
WHERE name NOT LIKE 'A%';
```

2. **Buscar apellidos que no terminen en 'son':**

sql

```sql
SELECT last_name
FROM employees
WHERE last_name NOT LIKE '%son';
```

Comparación de mayúsculas y minúsculas:

- En muchas bases de datos (como **MySQL**), `LIKE` es **case-insensitive** por defecto. Esto significa que `'a'` y `'A'` son tratados como iguales.
- En otras bases de datos (como **PostgreSQL**), la búsqueda con `LIKE` puede ser **case-sensitive**.

Para **MySQL**, puedes usar **`BINARY`** si quieres hacer una búsqueda sensible a mayúsculas y minúsculas:

sql

```
SELECT name
FROM users
WHERE BINARY name LIKE 'A%';
```

Uso de `ESCAPE` en `LIKE`:

Si quieres buscar caracteres especiales como `%` o `_`, puedes usar la cláusula `ESCAPE` para definir un carácter de escape:

sql

```
SELECT name
FROM users
WHERE name LIKE '%25!%%' ESCAPE '!';
```

Este ejemplo busca cadenas que contengan el carácter `%`.

Consideraciones de rendimiento:

- Los patrones que comienzan con `%` pueden **afectar el rendimiento** porque no se pueden utilizar índices para esas búsquedas.
  Ejemplo: `LIKE '%abc'` no usa índice, mientras que `LIKE 'abc%'` sí puede beneficiarse de un índice en la columna.

Diferencias entre `LIKE` y `=`:

- `=` busca coincidencias exactas, mientras que `LIKE` permite patrones flexibles.
    - Ejemplo: `WHERE name = 'John'` solo coincide con `'John'`, mientras que `WHERE name LIKE 'Jo%'` coincidirá con `'John'`, `'Jonathan'`, etc.

Resumen:

- `LIKE` se usa para realizar búsquedas por patrones en columnas de tipo texto.
- `NOT LIKE` es lo opuesto, devuelve valores que **no** coinciden con el patrón.
- Los **patrones especiales** (`%`, `_`, `[ ]`, `[^ ]`) permiten crear filtros complejos.
- `ESCAPE` te permite buscar caracteres especiales en los patrones.
- Las búsquedas que comienzan con `%` pueden afectar el rendimiento porque los índices no pueden ser utilizados eficientemente.

GROUP BY / HAVINH

`GROUP BY` y `HAVING` son cláusulas esenciales en SQL que permiten realizar operaciones de agregación en subconjuntos de datos. Mientras que `GROUP BY` agrupa filas que tienen valores en común en columnas específicas, `HAVING` se utiliza para filtrar grupos después de haber aplicado la agregación.

Sintaxis básica de `GROUP BY`:

sql

```
SELECT column_name, AGGREGATE_FUNCTION(column_name)
FROM table_name
WHERE condition
GROUP BY column_name;
```

Sintaxis básica de `HAVING`:

sql

```
SELECT column_name, AGGREGATE_FUNCTION(column_name)
FROM table_name
WHERE condition
GROUP BY column_name
HAVING condition;
```

`GROUP BY`: Agrupación de datos

`GROUP BY` permite agrupar las filas que comparten los mismos valores en las columnas especificadas y aplicar funciones de agregación (como `SUM()`, `AVG()`, `COUNT()`, etc.) sobre esos grupos.

- **Reglas:**
  - Las columnas que no forman parte de una función de agregación **deben** aparecer en la cláusula `GROUP BY`.
  - `GROUP BY` suele ir acompañado de funciones de agregación, como:
    - `COUNT()` - Cuenta el número de filas.
    - `SUM()` - Suma los valores.
    - `AVG()` - Calcula el promedio.
    - `MAX()` - Encuentra el valor máximo.
    - `MIN()` - Encuentra el valor mínimo.

Ejemplo básico de `GROUP BY`:

sql

```
SELECT department, COUNT(*)
FROM employees
GROUP BY department;
```

- Esto devuelve el número de empleados por cada departamento.

`HAVING`: Filtrar grupos

`HAVING` es similar a `WHERE`, pero se utiliza para filtrar **grupos** en lugar de filas individuales. Se aplica después de que `GROUP BY` ha agrupado los datos y las funciones de agregación se han calculado.

- **Diferencias clave entre `WHERE` y `HAVING`:**
  - **`WHERE`** se utiliza para **filtrar filas** antes de aplicar las funciones de agregación.
  - **`HAVING`** se utiliza para **filtrar grupos** después de que la agregación ha sido realizada.

Ejemplo de `GROUP BY` con `HAVING`:
sql

```
SELECT department, AVG(salary)
FROM employees
GROUP BY department
HAVING AVG(salary) > 50000;
```

- Este ejemplo agrupa los empleados por departamento, calcula el salario promedio en cada departamento y luego muestra solo los departamentos donde el salario promedio es mayor a 50,000.

Casos comunes de uso de `GROUP BY` y `HAVING`:

1. **Contar elementos en grupos:**
   sql

```
SELECT country, COUNT(*)
FROM customers
GROUP BY country;
```

- Agrupa clientes por país y cuenta cuántos hay en cada país.

Filtrar por una condición de grupo:
sql

```
SELECT country, COUNT(*)
FROM customers
GROUP BY country
HAVING COUNT(*) > 10;
```

- Muestra solo los países que tienen más de 10 clientes.

Usar múltiples columnas en `GROUP BY`:
sql

```
SELECT department, job_title, AVG(salary)
FROM employees
GROUP BY department, job_title;
```

1.
    - Agrupa a los empleados por departamento y título del puesto y calcula el salario promedio para cada grupo.

Detalles adicionales:

1. **Orden en la ejecución de la consulta:**
    - Primero se evalúa el **`FROM`**, luego el **`WHERE`**, después el **`GROUP BY`**, y finalmente el **`HAVING`** y el **`SELECT`**.
2. **Uso de alias en `HAVING`:**
    - A diferencia de `WHERE`, se pueden usar alias creados en la cláusula `SELECT` en la cláusula `HAVING`.
3. sql

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department
HAVING avg_salary > 50000;
```

4.

Limitaciones de `GROUP BY`:

- No puedes usar columnas en el `SELECT` que no estén en `GROUP BY` o en una función de agregación, ya que SQL no sabría qué valor devolver para esas columnas.
  Incorrecto:
  sql

```
SELECT department, name, AVG(salary)
```

```
FROM employees
GROUP BY department;
```

Esto generará un error, ya que la columna `name` no está agrupada ni dentro de una función de agregación.

Correcto:

sql

```
SELECT department, AVG(salary)
FROM employees
GROUP BY department;
```

-

`GROUP BY` y `ORDER BY`:

Es posible usar `GROUP BY` y `ORDER BY` en la misma consulta. `GROUP BY` agrupa los datos, y `ORDER BY` ordena los resultados después de la agrupación.

Ejemplo con `ORDER BY`:
sql

```
SELECT department, AVG(salary)
FROM employees
GROUP BY department
HAVING AVG(salary) > 50000
ORDER BY AVG(salary) DESC;
```

- Esto agrupa a los empleados por departamento, filtra los grupos cuyo salario promedio es mayor a 50,000 y luego ordena los resultados de mayor a menor salario promedio.

Resumen en formato tabla:

Cláusula	Función	Sintaxis	Ejemplo

`GROUP BY`	Agrupa filas en base a columnas especificadas. Permite aplicar funciones de agregación como `COUNT()`, `SUM()`, `AVG()`, etc.	`SELECT column_name, AGGREGATE_FUNCTION(column_name) FROM table_name GROUP BY column_name;`	`SELECT department, AVG(salary) FROM employees GROUP BY department;`
`HAVING`	Filtra grupos después de que se ha aplicado `GROUP BY` y la agregación. Se usa para condiciones sobre los grupos agregados.	`SELECT column_name, AGGREGATE_FUNCTION(column_name) FROM table_name GROUP BY column_name HAVING condition;`	`SELECT department, AVG(salary) FROM employees GROUP BY department HAVING AVG(salary) > 50000;`
`ORDER BY`	Ordena los resultados después de aplicar `GROUP BY` y `HAVING`. Se puede ordenar por columnas agrupadas o funciones de agregación.	`SELECT column_name, AGGREGATE_FUNCTION(column_name) FROM table_name GROUP BY column_name ORDER BY column_name;`	`SELECT department, AVG(salary) FROM employees GROUP BY department ORDER BY AVG(salary) DESC;`
