Sentencias Avanzado

1. Funciones y Procedimientos : DELIMITER

Para definir un procedimiento almacenado o función es necesario modificar temporalmente el carácter separador que se utiliza para delimitar las sentencias SQL.

El carácter separador que se utiliza por defecto en SQL es el punto y coma (;). En los ejemplos que vamos a realizar en esta unidad vamos a utilizar los caracteres $$ para delimitar las instrucciones SQL, pero es posible utilizar cualquier otro carácter.

Ejemplo:

En este ejemplo estamos configurando los caracteres $$ como los separadores entre las sentencias SQL.

DELIMITER $$

En este ejemplo volvemos a configurar que el carácter separador es el punto y coma.

DELIMITER ;

O mas claro :

Entendamos el problema del ; (punto y coma), normalmente, el ; (punto y coma) le dice a MySQL: "Terminé, ejecuta esto ahora".

Cuando escribes un Procedimiento Almacenado o función, este tiene varias líneas adentro, y cada una de esas líneas termina con ; (punto y coma). Si intentas crearlo normalmente, MySQL se detendría en el primer ; que encuentre dentro del procedimiento, pensaría que la orden está incompleta y te daría un error.

La solución es el DELIMITER, que se usa para evitar ese problema.

Este comando le dice a MySQL: "Oye, deja de hacerme caso cuando veas un ; (punto y coma). A partir de ahora, solo ejecuta cuando veas este nuevo símbolo (por ejemplo $$ o cualquier otro – bajo condiciones -)".

De esta forma, puedes escribir todo el cuerpo del procedimiento o función con sus ; internos sin que MySQL intente ejecutarlo antes de tiempo.

1.1 Explicación con "Cajas"

Imagina que el procedimiento o función es una caja que estás armando:

- **DELIMITER $$:** Abres un contenedor especial. Le dices a MySQL: "No cierres la caja hasta que veas $$".
- **CREATE PROCEDURE… :** Empiezas a meter las piezas (el código SQL) dentro de la caja.
- **; (Punto y coma) :** Son las conexiones de las piezas dentro de la caja. Como cambiaste el delimitador, MySQL los ignora y te deja seguir armando.
- **END$$:** Aquí es donde MySQL ve el nuevo símbolo, cierra la caja y guarda el procedimiento completo.
- **DELIMITER ; :** Muy importante. Le devuelves a MySQL su comportamiento normal para que el ; vuelva a ser el comando de ejecución.

Ejemplo Procedimiento Almacenado:

```
DELIMITER $$ -- 1. "Olvida el punto y coma por un momento"

CREATE PROCEDURE mi_ejemplo()
BEGIN
   SELECT 'Paso 1'; -- MySQL ve esto y NO lo ejecuta gracias al DELIMITER
   SELECT 'Paso 2'; -- MySQL sigue leyendo...
END$$          -- 2. "¡Ahora sí! Ejecuta todo este bloque"

DELIMITER ;  -- 3. "Vuelve a la normalidad, usa el punto y coma otra vez"
```

2. Diferencias entre funciones y procedimientos almacenados

En muchos lenguajes de programación, las funciones se denominan procedimientos. En otros lenguajes de programación, como SQL, existen varias diferencias clave entre estos dos términos.

Sin embargo, es importante darse cuenta de que ambos términos representan el mismo concepto: agrupar o encapsular código dentro del cuerpo de una función o de un procedimiento. Luego, esta función o procedimiento se llama para realizar una operación específica invocando su nombre de identificador.

2.1. El Valor de Retorno (La gran diferencia)

Funciones: Siempre deben retornar un único valor (un número, un texto, una fecha). Se usan para realizar cálculos.

Procedimientos: No están obligados a retornar nada. Pueden devolver varias tablas de datos (varios SELECT) o simplemente realizar acciones (borrar, insertar).

2.2. Cómo se llaman (Invocación)

Funciones: Se llaman directamente dentro de una sentencia SQL, como si fueran una columna más.

Ejemplo: *SELECT calcular_iva(precio) FROM facturas;*

Procedimientos: Se llaman con la palabra clave CALL.

Ejemplo: *CALL list_users('Carlos');*

2.3. Qué pueden hacer (Acciones)

Funciones: Normalmente son de "solo lectura" para procesar datos. Tienen muchas restricciones para modificar tablas (dependiendo de la versión de SQL).

Procedimientos: Son "todoterreno". Pueden hacer inserciones, actualizaciones, borrar datos y ejecutar lógica compleja de negocio.

3. Procedimientos almacenados

El propósito principal de crear procedimientos almacenados y funciones es crear código reutilizable que pueda invocarse y ejecutarse de forma eficiente. Así, en lugar de escribir el mismo código repetidamente, puedes guardar tus bloques de código en un procedimiento almacenado o en una función. Después, puedes llamar a estos bloques cuando necesites usar tu código.

Esto hace que tu código sea más coherente, mejor organizado, reutilizable y más fácil de mantener.

3.1 Parámetros de entrada, salida y entrada/salida

En los procedimientos almacenados podemos tener tres tipos de parámetros:

- **Entrada (IN)**: Se indican poniendo la palabra reservada IN delante del nombre del parámetro. Estos parámetros no pueden cambiar su valor dentro del procedimiento, es decir, cuando el procedimiento finalice estos parámetros tendrán el mismo valor que tenían cuando se hizo la llamada al procedimiento. En programación sería equivalente al paso por valor de un parámetro.

- **Salida (OUT)**: Se indican poniendo la palabra reservada OUT delante del nombre del parametro. Estos parámetros cambian su valor dentro del procedimiento. Cuando se hace la llamada al procedimiento empiezan con un valor inicial y cuando finaliza la ejecución del procedimiento pueden terminar con otro valor diferente. En programación sería equivalente al paso por referencia de un parámetro.

- **Entrada/Salida (IN/OUT)**: Es una combinación de los tipos IN y OUT. Estos parámetros se indican poniendo la palabra reservada IN/OUT delante del nombre del parámetro.

3.2 Estructura:

```
CREATE PROCEDURE nombre_proc(IN entrada TIPO, OUT salida TIPO)
BEGIN
   -- Lógica
   SELECT columna INTO salida FROM tabla ...; -- ¡Opcional!
END;
```

3.3 Ejemplos

La siguiente consulta puede usarse para devolver los nombres de todos los clientes de la tabl*a use*rs de la base de datos *universi*ty:

*select * from users;*

Puedes incluir esta sentencia en un procedimiento almacenado de la siguiente manera:

```
-- Eliminar el procedimiento si ya existe
DROP PROCEDURE IF EXISTS GetAllClients;

-- Cambiar el delimitador para poder usar ;
DELIMITER //

-- Crear el procedimiento almacenado
CREATE PROCEDURE GetAllClients()
BEGIN
 SELECT * FROM university.users;
END //

-- Restaurar el delimitador predeterminado
DELIMITER ;
```

Puede invocar este *procedimiento* simplemente llamando al nombre del identificador, tal y como se muestra en la siguiente captura de pantalla. Este es el resultado de llamar al *procedimiento* GetAllClients().

- *CALL GetAllClients();*

Ejemplo de un procedimiento con parámetros de entrada

Que hace este procedimiento?

```
-- Cambiar el delimitador
DELIMITER $$

-- Eliminar el procedimiento si ya existe
DROP PROCEDURE IF EXISTS list_users;

-- Crear el procedimiento almacenado y usamos la variable con p_ significando parametro
CREATE PROCEDURE list_users(IN p_nombre VARCHAR(50))
BEGIN
 SELECT * FROM users WHERE users.name = p_nombre;
END$$
```

```
-- Restaurar el delimitador predeterminado
DELIMITER ;
```

Invocamos el procedimiento

```
CALL list_users('Claudia');
```

Ejemplo de un procedimiento con parámetro de salida

A diferencia del IN (que es para darle datos al procedimiento), el OUT sirve para que el procedimiento "escriba" un resultado en una variable que tú le entregues, para que puedas usar ese valor más tarde.

Este procedimiento recibe el ID (entrada) y nos devuelve la edad (salida).

```
DELIMITER //

-- Eliminar el procedimiento si ya existe
DROP PROCEDURE IF EXISTS obtener_edad_usuario//

CREATE PROCEDURE obtener_edad_usuario(
    IN p_user_id INT,    -- El ID que queremos buscar
    OUT p_age_result INT -- Donde guardaremos la edad encontrada
)
BEGIN
    SELECT age INTO p_age_result FROM users WHERE user_id = p_user_id;
END //

DELIMITER ;
```

Para probarlo con los datos de la base de datos University (por ejemplo, para Claudia que tiene ID 3 y 29 años), hacemos lo siguiente:

SQL

```
-- 1. Llamamos al procedimiento pasando el ID 3 y una variable vacía @edad_detectada
CALL obtener_edad_usuario(3, @edad_detectada);

-- 2. Consultamos nuestra variable para ver qué nos devolvió el procedimiento
SELECT @edad_detectada AS 'Edad de Claudia';
```

Explicación:

- IN p_user_id INT: Es el filtro. El procedimiento busca en la tabla al usuario que coincida con este número.
- SELECT age INTO p_age_result: Esta es la parte clave. En lugar de mostrar la edad en una tabla de resultados normal, la "inyecta" dentro de nuestro parámetro de salida.
- OUT p_age_result INT: Actúa como un sobre. Tú le entregas el sobre vacío al procedimiento (@edad_detectada), él mete el dato adentro y te lo devuelve cerrado para que tú lo abras cuando quieras con un SELECT.

Ejemplo de un procedimiento con dos parámetros de salida

Este procedimiento recibe el ID (entrada) y nos devuelve la edad (salida) y nombre (salida).

```
DELIMITER //

-- Eliminar el procedimiento si ya existe
DROP PROCEDURE IF EXISTS obtener_edad_usuario//

CREATE PROCEDURE obtener_edad_usuario(
IN p_user_id INT, -- El ID que queremos buscar
OUT p_age_result INT, -- Donde guardaremos la edad encontrada
OUT p_name_result VARCHAR(50))
BEGIN
SELECT age INTO p_age_result FROM users WHERE user_id = p_user_id;
SELECT name INTO p_name_result FROM users WHERE user_id = p_user_id;
```

```
END //
DELIMITER ;
```

Lo ejecutamos

```
CALL obtener_edad_usuario(1,@edad,@nombre);
SELECT @edad,@nombre;
```

Analizar este ejemplo:

```
DELIMITER //
-- Eliminar el procedimiento si ya existe
DROP PROCEDURE IF EXISTS obtener_datos_contacto//

CREATE PROCEDURE obtener_datos_contacto(
    IN p_user_id INT,              -- Entrada: ID del usuario
    OUT p_nombre_completo VARCHAR(150), -- Salida 1: Nombre + Apellido
    OUT p_email_usuario VARCHAR(100)    -- Salida 2: Correo
)
BEGIN
    SELECT
        CONCAT(name, ' ', IFNULL(surname, '')) ,
        email
    INTO
        p_nombre_completo,
        p_email_usuario
    FROM users
    WHERE user_id = p_user_id;
```

```
END //


DELIMITER ;
```

Como ahora tenemos dos "sobres" (parámetros de salida), necesitamos preparar dos variables con @:

SQL

```
-- 1. Llamamos al procedimiento para el ID 1 (Marcela Hernandez)
CALL obtener_datos_contacto(1, @nombre, @correo);

-- 2. Consultamos ambas variables al mismo tiempo
SELECT @nombre AS 'Usuario', @correo AS 'Contacto';
```

4. Funciones

Una función almacenada es un conjunto de instrucciones SQL que se almacena asociado a una base de datos. Es un objeto que se crea con la sentencia CREATE FUNCTION y se invoca con la sentencia SELECT o dentro de una expresión. Una función puede tener cero o muchos parámetros de entrada y siempre devuelve un valor, asociado al nombre de la función.

4.1 Parámetros de entrada

En una función todos los parámetros son de entrada, por lo tanto, no será necesario utilizar la palabra reservada IN delante del nombre de los parámetros.

Ejemplo:

A continuación se muestra la cabecera de la función contar_productos que tiene un parámetro de entrada llamado gama.

```
CREATE FUNCTION contar_usuarios(nombre VARCHAR(50))
```

4.2 Estructura

```
CREATE FUNCTION nombre_func(parametro TIPO)
RETURNS TIPO_DATO DETERMINISTIC
BEGIN
   -- Lógica
   RETURN valor; -- ¡Obligatorio!
END;
```

4.2 Resultado de salida

Una función siempre devolverá un valor de salida asociado al nombre de la función. En la definición de la cabecera de la función hay que definir el tipo de dato que devuelve con la palabra reservada RETURNS y en el cuerpo de la función debemos incluir la palabra reservada RETURN para devolver el valor de la función.

Ejemplo:

En este ejemplo se muestra una definición incompleta de una función donde se se puede ver el uso de las palabras reservadas RETURNS y RETURN.

```
-- Eliminar la función si ya existe
DROP FUNCTION IF EXISTS contar_usuarios;
```

```
-- Cambiar el delimitador
DELIMITER $$

-- Crear la función
CREATE FUNCTION contar_usuarios(nombre VARCHAR(50))
RETURNS INT UNSIGNED
READS SQL DATA
DETERMINISTIC
BEGIN
   DECLARE total INT UNSIGNED;

   -- Contar cuántos usuarios tienen ese nombre
   SET total = (SELECT COUNT(*) FROM university.users WHERE name = nombre);

   RETURN total;
END$$

-- Restaurar el delimitador predeterminado
DELIMITER ;


SELECT contar_usuarios('Claudia');
```

Explicación

- *contar_usuarios(nombre VARCHAR(50))*: Aquí se define que la función recibe un dato de entrada (el nombre que quieres buscar).
- *RETURNS INT UNSIGNED:* Se le indica a MySQL que el resultado final será un número entero positivo (sin signos negativos).
- *READS SQL DATA:* Declaramos que esta función entrará a las tablas a leer información (el SELECT).
- *DETERMINISTIC:* Aseguramos que para un mismo nombre, el resultado será consistente.
- DECLARE total INT UNSIGNED;: Creamos una variable temporal llamada total que solo vive dentro de la función. Es como un papel en blanco donde anotarás el resultado antes de entregarlo.
- SET total = (...): Aquí es donde ocurre la acción. Ejecutamos la consulta y el número resultante se guarda en tu variable total.
- RETURN total;: Es la instrucción final. La función "entrega" el valor almacenado hacia quien la llamó.

Si queremos saber cuántas "Marcela" hay:

```
SELECT contar_usuarios('Marcela') AS Cantidad;
```

O lo podemos volver mas complejo:

SELECT IF(contar_usuarios('Marcela') > 1, 'Hay varias', 'Hay una o ninguna');

4.3 Características de la función

Después de la definición del tipo de dato que devolverá la función con la palabra reservada RETURNS, tenemos que indicar las características de la función. Las opciones disponibles son las siguientes:

- **DETERMINISTIC**: Indica que la función siempre devuelve el mismo resultado cuando se utilizan los mismos parámetros de entrada.
- **NOT DETERMINISTIC**: Indica que la función no siempre devuelve el mismo resultado, aunque se utilicen los mismos parámetros de entrada. Esta es la opción que se selecciona por defecto cuando no se indica una característica de forma explícita. Ejemplo : Una función que use `NOW()` (la hora actual) o `RAND()` (un número aleatorio). Si la llamas ahora te da un valor, y si la llamas en 5 minutos te da otro, aunque no hayas tocado la base de datos.
- **CONTAINS SQL**: Indica que la función contiene sentencias SQL, pero no contiene sentencias de manipulación de datos. Algunos ejemplos de sentencias SQL que pueden aparecer en este caso son operaciones con variables (Ej: SET @x = 1) o uso de funciones de MySQL (Ej: SELECT NOW();) entre otras. Pero en ningún caso aparecerán sentencias de escritura o lectura de datos.
- **NO SQL**: Indica que la función no contiene sentencias SQL.
- **READS SQL DATA**: Indica que la función no modifica los datos de la base de datos y que contiene sentencias de lectura de datos, como la sentencia SELECT.
- **MODIFIES SQL DATA**: Indica que la función sí modifica los datos de la base de datos y que contiene sentencias como INSERT, UPDATE o DELETE.

Para poder crear una función en MySQL es necesario indicar al menos una de estas tres características:

- *DETERMINISTIC ,NO SQL, READS SQL DATA*

Si no se indica al menos una de estas características obtendremos el siguiente mensaje de error.

*ERROR 1418 (HY000): This function has none of DETERMINISTIC, NO SQL, or READS SQL DATA in its declaration and binary logging is enabled (you *might* want to use the less safe log_bin_trust_function_creators variable)*

Es posible configurar el valor de la variable global log_bin_trust_function_creators a 1, para indicar a MySQL que queremos eliminar la restricción de indicar alguna de las características anteriores cuando definimos una función almacenada. Esta variable está configurada con el valor 0 por defecto y para poder modificarla es necesario contar con el privilegio SUPER.

- *SET GLOBAL log_bin_trust_function_creators = 1;*

En lugar de configurar la variable global en tiempo de ejecución, es posible modificarla en el archivo de configuración de MySQL.

¿Para qué sirve poner el *DETERMINISTIC*?

MySQL utiliza esta información para optimizar el rendimiento:

- Caché: Si MySQL sabe que la función es determinista, puede guardar el resultado en memoria. Si la vuelves a llamar con los mismos parámetros, no vuelve a calcular todo, simplemente te entrega el resultado que ya conoce.
- Replicación: Si tienes varios servidores de base de datos sincronizados, esta marca le ayuda a MySQL a asegurarse de que el resultado sea el mismo en todos los servidores.
- Seguridad: En algunas configuraciones de MySQL, si no especificas si es DETERMINISTIC, NO DETERMINISTIC o READS SQL DATA, el sistema no te permitirá crear la función por razones de seguridad (especialmente si la replicación binaria está activada – MySQL replica).

4.4 Ejemplos:

La siguiente consulta es un ejemplo de una *función* que devuelve el coste medio de todos los pedidos de la tabla Pedidos de la base de datos *university.*

```
SELECT AVG(Cost) FROM Orders;
```

Puede envolver esta instrucción en una función almacenada de la siguiente manera:

```
-- Cambiar el delimitador
DELIMITER $$

-- Eliminar la función si ya existe
DROP FUNCTION IF EXISTS AgeAverage$$

-- Crear la función
CREATE FUNCTION AgeAverage()
        RETURNS DECIMAL(5,2) DETERMINISTIC
        READS SQL DATA
BEGIN
   RETURN (SELECT AVG(age) FROM users);
```

```
END$$

-- Restaurar el delimitador predeterminado
DELIMITER ;
```

Puede invocar esta función simplemente con una instrucción select, para llamar al procedimiento AgeAverage():

```
SELECT AgeAverage();
```

Ejemplo 2

Imaginemos que necesitamos clasificar a los estudiantes en categorías (Infantil, Joven, Adulto) dependiendo de su edad. Hacer esto con un CASE en cada consulta es tedioso, así que una función es la solución perfecta.

Función: obtener_categoria_edad
Esta función recibirá la edad y devolverá un texto con la categoría.

```
DELIMITER $$

DROP FUNCTION IF EXISTS obtener_categoria_edad$$

CREATE FUNCTION obtener_categoria_edad(p_age INT)
RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN
   DECLARE categoria VARCHAR(20);

   IF p_age IS NULL THEN
      SET categoria = 'No definida';
   ELSEIF p_age < 12 THEN
      SET categoria = 'Infantil';
   ELSEIF p_age BETWEEN 12 AND 17 THEN
      SET categoria = 'Adolescente';
   ELSEIF p_age BETWEEN 18 AND 29 THEN
      SET categoria = 'Joven';
   ELSE
      SET categoria = 'Adulto';
   END IF;

   RETURN categoria;
END$$
```

```
DELIMITER ;
```

Ojo en la segunda linea.
Si hubieramos escrito:
DROP FUNCTION IF EXISTS obtener_categoria_edad;

MySQL se quedaría esperando. Para él, ese ; (punto y coma) ya no significa nada; es como si hubieras dejado la frase a medias. No borraría la función porque seguiría esperando su nuevo símbolo de "enviar" o "terminar".

Al poner $$ al final de esa línea, obligo a MySQL a ejecutar el DROP de inmediato antes de pasar a la creación de la función.

O… podriamos haberlo hecho de esta manera :

```
DROP FUNCTION IF EXISTS obtener_categoria_edad;

DELIMITER $$

CREATE FUNCTION obtener_categoria_edad(p_age INT)
RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN
   DECLARE categoria VARCHAR(20);

   IF p_age IS NULL THEN
      SET categoria = 'No definida';
   ELSEIF p_age < 12 THEN
      SET categoria = 'Infantil';
   ELSEIF p_age BETWEEN 12 AND 17 THEN
      SET categoria = 'Adolescente';
   ELSEIF p_age BETWEEN 18 AND 29 THEN
      SET categoria = 'Joven';
   ELSE
      SET categoria = 'Adulto';
   END IF;

   RETURN categoria;
END$$

DELIMITER ;
```

Ahora procedemos a ejecutarlo:

SELECT name AS Nombre, age AS Edad, obtener_categoria_edad(age) AS Categoria FROM university.users;

Ejemplo 3 Función con JOIN usando la base de datos universidad

Objetivo: Crear una función que reciba el código de un estudiante y devuelva el nombre de la facultad a la que pertenece.

Esta función debe unir las tablas estudiantes, carreras y facultades.

```
DELIMITER $$

DROP FUNCTION IF EXISTS obtener_facultad_estudiante$$

CREATE FUNCTION obtener_facultad_estudiante(p_codigo VARCHAR(20))
RETURNS VARCHAR(100)
DETERMINISTIC
READS SQL DATA
BEGIN
    DECLARE v_facultad VARCHAR(100);

    SELECT f.nombre INTO v_facultad
    FROM estudiantes e
    JOIN carreras c ON e.id_carrera = c.id_carrera
    JOIN facultades f ON c.id_facultad = f.id_facultad
    WHERE e.codigo = p_codigo;

    RETURN IFNULL(v_facultad, 'No encontrado');
END$$

DELIMITER ;
```

Explicación

- Normalmente, cuando hamos un SELECT f.nombre, MySQL muestra el resultado en una tabla en la pantalla. Sin embargo, dentro de una función o procedimiento, no queremos que el dato se "muestre", sino que se guarde en algún lugar para seguir trabajando con él.

- La palabra clave INTO toma el valor que encontró la consulta y lo deposita dentro de la variable v_facultad.

- La consulta debe devolver exactamente una fila: Si el SELECT encuentra dos facultades, MySQL lanzará un error porque no sabe cuál de los dos nombres meter en la variable.

¿Cómo se usa?
Puedes usarla en un reporte para ver el estudiante junto a su facultad:

```
SELECT nombres, apellidos, obtener_facultad_estudiante(codigo) AS facultad
FROM estudiantes;
```

Ejemplo 4 Procedimiento con JOIN y OUT usando la base de datos universidad

Objetivo: Crear un procedimiento que reciba el ID de una materia y devuelva mediante parámetros OUT cuántos estudiantes hay inscritos en ella y el nombre de la carrera a la que pertenece esa materia.

```
DELIMITER $$

DROP PROCEDURE IF EXISTS obtener_estadisticas_materia$$

CREATE PROCEDURE obtener_estadisticas_materia(
 IN p_id_materia INT,
 OUT p_total_inscritos INT,
 OUT p_nombre_carrera VARCHAR(100)
)
BEGIN
 -- Obtenemos el total de inscritos y el nombre de la carrera en una sola consulta con JOIN
 SELECT COUNT(i.id_estudiante), c.nombre
 INTO p_total_inscritos, p_nombre_carrera
 FROM materias m
 JOIN carreras c ON m.id_carrera = c.id_carrera
 LEFT JOIN inscripciones i ON m.id_materia = i.id_materia
 WHERE m.id_materia = p_id_materia
 GROUP BY m.id_materia, c.nombre;
END$$

DELIMITER ;
```

¿Cómo se prueba?

```
-- 1. Llamamos al procedimiento para la materia con ID 1
CALL obtener_estadisticas_materia(1, @total, @carrera);

-- 2. Vemos los resultados capturados en las variables
SELECT @total AS 'Inscritos', @carrera AS 'Carrera de la Materia';
```

4.5 Diferencias clave entre funciones y procedimientos

Una función en MySQL también se llama función almacenada. Un procedimiento se llama

procedimiento almacenado. Los procedimientos y funciones almacenados básicos suelen representar operaciones que contienen parámetros vacíos, o parámetros de entrada simples, y una única sentencia SQL.

Los procedimientos y funciones más complejos requieren el uso de características adicionales como parámetros complejos, variables, cambio de delimitadores y el uso de las palabras clave `BEGIN-END`. También suelen requerir el uso de múltiples sentencias SQL dentro del cuerpo del procedimiento.

En este contexto, las diferencias clave entre funciones y procedimientos son las siguientes:

- Una función devuelve un único valor, mientras que un procedimiento puede devolver un solo valor, múltiples valores o ningún valor.

- Por lo general, las funciones encapsulan fórmulas comunes o reglas de negocio genéricas que pueden reutilizarse entre sentencias SQL y procedimientos almacenados. Los procedimientos, por otro lado, se usan principalmente para procesar, manipular y modificar datos en la base de datos.

- Las funciones solo aceptan parámetros de entrada (IN), mientras que los procedimientos almacenados pueden aceptar parámetros IN, OUT e INOUT.

- Las funciones pueden invocarse desde cualquier lugar, incluyendo sentencias `SELECT` y procedimientos almacenados. Los procedimientos almacenados se invocan únicamente mediante la sentencia `CALL`.

- Una función almacenada se crea usando la sentencia `CREATE FUNCTION`. Un procedimiento almacenado se crea usando la sentencia `CREATE PROCEDURE`.

- Al crear una función, debes especificar si es una función `DETERMINISTIC` o no. Esto significa que debes decidir si la función siempre devuelve el mismo resultado para los mismos parámetros de entrada. Si no usas `DETERMINISTIC`, MySQL usa la opción `NOT DETERMINISTIC` por defecto.

- Para crear una función, debes especificar el tipo de dato del valor de retorno en la sentencia `RETURNS`. Este puede ser cualquier tipo de dato válido en MySQL. Sin embargo, no es necesario hacer esto con los procedimientos almacenados.

La siguiente tabla ofrece un resumen de las principales diferencias entre los procedimientos almacenados y las funciones almacenadas.

Functions	Procedures
Created using CREATE FUNCTION command	Created using the CREATE PROCEDURE command
Se usa dentro de un SELECT, WHERE o SET.	Se usa con la palabra clave CALL.
Takes IN parameters only	Takes IN, OUT and INOUT parameters
Typically encapsulates common formulas or generic business rules	Typically used to process, manipulate and modify data in the database
Must specify the data type of the return value	User must specify the OUT parameter type
Obligatorio. Siempre devuelve un único valor con RETURN.	No usa RETURN. Devuelve datos mediante parámetros OUT.
Solo acepta parámetros de entrada (IN).	Acepta parámetros de entrada (IN), salida (OUT) e híbridos (INOUT).

Las funciones y los procedimientos se utilizan para encapsular código que puede ejecutarse para implementar tareas repetitivas, como ecuaciones, fórmulas o reglas de negocio.

Además, las funciones y los procedimientos almacenados hacen que tu código sea más consistente, reutilizable, más fácil de usar y de mantener.

Sin embargo, debes conocer las diferencias clave entre funciones y procedimientos en MySQL para saber cuándo utilizar uno u otro.

4.6 ¿Cuándo usar cada uno? (Regla de oro)

Usa una FUNCIÓN cuando:

- Necesites un "calculador". Si quieres algo que tome un dato y te devuelva un resultado para usarlo en una consulta.
- Ejemplo: Calcular el IVA de un producto, obtener la inicial de un nombre o convertir monedas.
- Sintaxis de uso: *SELECT nombre, calcular_iva(precio) FROM productos;*

Usa un PROCEDIMIENTO cuando:

- Necesites un "operador". Si quieres ejecutar una serie de pasos que pueden o no devolver información, o si necesitas devolver múltiples valores.
- Ejemplo: Registrar una venta (que implica bajar stock, crear factura y enviar correo).
- Sintaxis de uso: *CALL registrar_venta(id_producto, cantidad);*

5. Index

Un índice (index) en MySQL es una estructura de datos especial que se crea en una o más columnas de una tabla con el propósito principal de **mejorar la velocidad de las operaciones de búsqueda y acceso a los datos.

Piensa en un índice como el índice de un libro: en lugar de leer todo el libro para encontrar un tema, consultas el índice para ir directamente a la página correcta. De manera similar, un índice en MySQL permite al motor de base de datos encontrar filas específicas sin tener que escanear toda la tabla.

5.1 ¿Para qué sirve un índice?

- Acelerar consultas (`SELECT`, `WHERE`, `JOIN`, `ORDER BY`).
- Mejorar el rendimiento de claves foráneas y restricciones.
- Evitar duplicados cuando se usa un índice `UNIQUE`.
- Soportar búsquedas por rangos o patrones (como con `LIKE 'texto%'`).

Ejemplo sin y con índice

Supón que tienes esta consulta:

*SELECT * FROM users WHERE email = 'maria@miemail.com';*

- Sin índice en *email*: MySQL debe revisar cada fila de la tabla (`type: ALL` en `EXPLAIN`) → lento.
- Con índice en *email*: MySQL va directamente a la fila deseada usando el índice → rápido.

5.2 Tipos comunes de índices en MySQL

Tipo	Descripción
INDEX (o KEY)	Índice normal; mejora la velocidad de búsquedas. Puede contener valores duplicados.
UNIQUE	Asegura que todos los valores en la columna sean diferentes.
PRIMARY KEY	Es un *UNIQUE INDEX* que además identifica cada fila de forma única. No puede ser nulo.
FULLTEXT	Usado para búsquedas de texto completo en columnas tipo *CHAR*, *VARCHAR* o *TEXT*.

INDEX compuesto	Creado sobre dos o más columnas. Ej: *CREATE INDEX idx_name ON users(name, surname);*

5.3 Cuándo crear un índice

- Crea un índice en columnas que:
- Se usan frecuentemente en cláusulas `WHERE`.
- Son parte de relaciones `JOIN`.
- Se usan en `ORDER BY` o `GROUP BY`.
- Deben ser únicas (`UNIQUE`).

5.4 No abusar de los índices

- Cada índice consume espacio en disco y memoria.
- Las operaciones *INSERT*, *UPDATE* y *DELETE* pueden volverse más lentas, ya que los índices también deben actualizarse.
- Demasiados índices pueden perjudicar el rendimiento general.

5.5 Sintaxis básica para crear un índice

Crear un índice simple

CREATE INDEX idx_email ON users(email);

Crear un índice único

CREATE UNIQUE INDEX idx_user_id ON users(user_id);

Ver los índices de una tabla

SHOW INDEX FROM users;

Eliminar un índice

DROP INDEX idx_email ON users;

La sentencia EXPLAIN en MySQL es una herramienta que muestra el plan de ejecución que el optimizador del motor utiliza para ejecutar una consulta SELECT, UPDATE, INSERT o DELETE.

Su propósito principal es ayudarte a entender cómo procesa MySQL tu consulta, con el fin de identificar cuellos de botella y optimizar su rendimiento.

- Identificar como se encuentra estructurada la columna company_id en el usuario Claudia.

```
EXPLAIN select company_id
from users
where name = 'Claudia';
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | users | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    5 |    20.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
```

- select_type = SIMPLE en MySQL significa que tu consulta es simple y directa — no tiene subconsultas, UNIONs ni joins complejos que requieran procesamiento adicional.

Aunque solo buscamos name = 'Claudia', MySQL revisó las 5 filas una por una. Con millones de registros esto sería catastrófico. Filtered es un porcentaje 20% y quiere decir que solo el 20% cumple las condiciones de la sentencia. Es decir: 5 filas × 20% = ~1 fila que realmente coincide.

Columna	Tu valor	¿Qué dice?	¿Está bien?
type	ALL	Escaneo completo de tabla	Mal
possible_keys	NULL	No encontró índices útiles	Mal
key	NULL	No usó ningún índice	Mal
rows	5	Revisó 5 filas	Aceptable por ahora, pero escala mal
Extra	Using where	Filtró después de leer	Ineficiente

Una consulta con *type: ALL* y *possible_keys: NULL* indica que no hay índices disponibles, lo cual es ineficiente. La solución es crear un índice adecuado.

5.6 Explicación del resultado comando Explain

Columna 01: ID

La columna ID es un identificador secuencial para cada sentencia `SELECT` dentro de la consulta. Esta información tiene más sentido cuando hay consultas anidadas o subconsultas.

En el ejemplo, el valor es 1, ya que se trata de un ejemplo sencillo con una sola consulta.

Columna 02: SELECT_TYPE

Esta columna muestra el tipo de consulta `SELECT` que se va a ejecutar.

Existen varios valores posibles para este campo:

Valor	Descripción
SIMPLE	Consulta `SELECT` simple sin subconsultas ni uniones (`UNION`)
PRIMARY	El `SELECT` está en la consulta más externa de un `JOIN`
DERIVED	El `SELECT` forma parte de una subconsulta dentro de una cláusula `FROM`
SUBQUERY	Es el primer `SELECT` en una subconsulta
DEPENDENT SUBQUERY	El `SELECT` es una subconsulta dependiente de una consulta externa
UNCACHEABLE SUBQUERY	Subconsulta que no puede almacenarse en caché (existen ciertas condiciones para que una consulta sea cacheable)
UNION	El `SELECT` es la segunda o posterior sentencia de una `UNION`
DEPENDENT UNION	La segunda o posterior sentencia `SELECT` de una `UNION` depende de una consulta externa
UNION RESULT	El `SELECT` es el resultado de una operación `UNION`

La consulta devuelve el valor SIMPLE, porque es una consulta básica que no contiene subconsultas ni uniones.

Columna 03: Table

Esta columna muestra el nombre de la tabla a la que hace referencia la consulta `SELECT`. En este caso, la consulta hace referencia a la tabla *users*.

Columna 04: Partitions

Esta columna muestra la partición en la que residen los datos (el área de almacenamiento físico que se escanea). La partición permite distribuir partes de los datos de una tabla a través del sistema de archivos. Si las consultas acceden solo a una fracción de los datos, hay menos registros que escanear y las consultas pueden ejecutarse más rápido. Sin embargo, la partición es más relevante cuando se trabaja con conjuntos de datos grandes.

La tabla *users* de *univesirty* no está particionada, por lo tanto, esta columna muestra un valor NULL.

Columna 05: type

Escaneo de la tabla significa realizar una operación de búsqueda o encontrar coincidencias especificadas por la consulta `SELECT`.

La siguiente tabla describe los valores más comunes posibles:

Valor	Descripción
system	La tabla tiene una sola fila o cero filas. Este valor indica típicamente que la búsqueda se realizó en una tabla del sistema
const	Indica que el valor de la columna buscada puede tratarse como una constante (existe una sola fila que coincide con la consulta)
eq_ref	Indica que se utiliza un índice agrupado (ya sea clave primaria o índice único con todas las columnas NOT NULL)
ref	Indica que la columna indexada fue accedida usando un operador de igualdad (`=`). Un ejemplo de esto aparece en el resultado de `EXPLAIN` de la consulta optimizada de *university*, donde la columna indexada `*name*` se compara usando `=` en la cláusula `WHERE`
full text	El escaneo usa el índice FULLTEXT de la tabla. Los índices de texto completo se crean en columnas basadas en texto (`CHAR`, `VARCHAR` o `TEXT`)
index	Se escanea todo el índice para encontrar coincidencias con la consulta
all	Se escanea toda la tabla para encontrar filas coincidentes. Este es el peor tipo de escaneo y generalmente indica la falta de índices adecuados en la tabla

La consulta devuelve el valor ALL, lo que indica que MySQL escanea toda la tabla (cada fila) para encontrar las filas coincidentes. Esto sugiere un problema al ejecutar esta consulta. En otras palabras, se trata de una consulta subóptima. Esto ocurre porque no existe un índice definido en la tabla.

Columna 06: possible_keys

Esta columna muestra las claves (índices) que podrían usarse por MySQL para encontrar filas en la tabla. Sin embargo, estos índices pueden o no ser usados en la práctica. Si el valor de la columna es NULL, indica que no se encontraron índices relevantes.

En el ejemplo, el valor es NULL, lo que indica que no existen claves o índices en la tabla que MySQL pueda usar para encontrar o filtrar filas. Esto señala un problema que debe abordarse.

Columna 07: key

Indica el índice real utilizado por MySQL.

En el ejemplo, esta columna devuelve NULL, lo que significa que no hay ningún índic en la tabla que el optimizador pueda usar. Este es un problema que debe corregirse, ya que se necesita un índice para mejorar el rendimiento.

Columna 08: key_len

Esta columna indica la longitud del índice que el Optimizador de Consultas decide utilizar. Por ejemplo, un valor de `key_len = 4` significa que se requieren 4 bytes para almacenar la clave.

La base de datos de *university* vuelve a devolver NULL, porque no hay ningún índice disponible para usar en esta consulta.

Columna 09: ref

Esta columna muestra qué columnas de tabla han sido comparadas con el índice para realizar la búsqueda. Un valor de const significa que se usó una constante; un valor de func significa que el valor provino de una función.

En la base de datos de *university*, ninguna columna ni constante se compara con un índice, porque la tabla no tiene uno. Por lo tanto, el valor es NULL. Esto vuelve a destacar el problema de no usar un índice en la tabla.

Columna 10: rows

Lista el número de registros que fueron examinados para producir el resultado.

Los resultados de *university* indican que se examinaron 5 registros. Esto significa que cada fila de la tabla fue revisada. Este es un uso ineficiente de los recursos de la base de datos.

Si hubiera cientos o incluso miles de registros en la tabla, una consulta ineficiente tendría que examinar cada uno, lo cual tomaría mucho más tiempo y consumiría más recursos que una consulta eficiente que solo examine los registros necesarios.

Columna 11: filtered

Esta columna indica un porcentaje aproximado de las filas de la tabla que han sido filtradas por una condición específica. Cuanto mayor sea el porcentaje, mejor será el rendimiento de la consulta.

En la consulta el 20% de las filas en la tabla *users* son filtradas por la condición de la cláusula `WHERE`.

Columna 12: Extra

Contiene información adicional sobre el plan de ejecución de la consulta. Presta atención si aparecen valores como Using temporary o Using filesort, ya que indican una consulta problemática. Veamos rápidamente el significado de estos valores.

- Using temporary: Indica que MySQL necesita crear una tabla temporal para almacenar el resultado de esta consulta. Esto suele ocurrir cuando la consulta contiene cláusulas `GROUP

BY` y `ORDER BY` que listan columnas diferentes (es decir, cuando las columnas en `GROUP BY` y `ORDER BY` no coinciden).

- Using filesort: Indica que MySQL debe realizar un paso adicional para determinar cómo recuperar las filas en orden. El ordenamiento se realiza examinando todas las filas según el tipo (mencionado anteriormente), almacenando la clave de orden y un puntero a la fila para todas las filas que cumplan con la cláusula `WHERE`. Luego se ordenan las claves y se recuperan las filas en orden.

El ejemplo tiene el valor *Using where*. Esto indica que MySQL primero lee cada fila de la tabla antes de filtrarlas. Esta no es una manera eficiente de usar la cláusula `WHERE`. Si se hubiera definido un índice en la tabla *users*, este valor sería diferente.

5.7 Ejemplo

- Crea un índice llamado "idx_name" en la tabla "users" asociado al campo "name"

  CREATE INDEX idx_name ON users(name);

- Usamos la instrucción Explain, para poder analizar el query

  EXPLAIN select company_id
  from users
  where name = 'Claudia';

```
+----+-------------+-------+------------+------+---------------+-------------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key         | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+-------------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | users | NULL       | ref  | idx_name      | idx_name | 202     | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+-------------+---------+-------+------+----------+-------+
```

Comparar los resultados.

- Crea un índice único llamado "idx_name" en la tabla "users" asociado al campo "name"
  CREATE UNIQUE INDEX idx_name ON users(name);

- Crea un índice llamado "idx_name_surname" en la tabla "users" asociado a los campos "name" y "surname"
  CREATE UNIQUE INDEX idx_name_surname ON users(name, surname);

- Ver índices de una tabla específica
  SHOW INDEX FROM users;
  ó
  SHOW INDEX FROM university.users;

Non_unique = 0 → índice único (como PRIMARY)
Non_unique = 1 → índice no único

- Usando DESCRIBE
  DESCRIBE users;
  ó
  DESC users;

  Muestra las columnas y marca las que son clave en *KEY: PRI = Primary Key, UNI = Unique, MUL = Índice múltiple*

- Elimina el índice llamado "idx_name"
  DROP INDEX idx_name ON users;

6. Triggers

Un trigger (disparador) es un bloque de código SQL que se ejecuta automáticamente cuando ocurre un evento específico en una tabla.

Analogía: Es como una alarma automática — no la activas manualmente, se dispara sola cuando pasa algo.

6.1 ¿Cuándo se activa?

Momento	Evento
BEFORE (antes)	INSERT, UPDATE, DELETE
AFTER (después)	INSERT, UPDATE, DELETE

6.2 Usos comunes

Uso	Ejemplo
Auditoría	Registrar quién y cuándo modificó datos
Validación	Rechazar datos inválidos antes de guardar
Sincronización	Actualizar otra tabla relacionada
Cálculos automáticos	Generar totales o códigos automáticamente

6.3 Reglas importantes

Regla	Explicación
Un trigger pertenece a una tabla	No puede vigilar varias tablas
Se ejecuta por cada fila afectada	FOR EACH ROW
No puedes modificar la misma tabla	Evita bucles infinitos
Puedes acceder a los valores viejos/nuevos	OLD.columna / NEW.columna

6.4 OLD vs NEW

Palabra	Significado	Disponible en
OLD	Valor antes del cambio	UPDATE, DELETE
NEW	Valor después del cambio	INSERT, UPDATE

Un trigger es código SQL que corre solo cuando insertas, actualizas o borras datos en una tabla — sin que nadie lo llame manualmente.

6.5 Ejemplo :

Crea una tabla de historial para usar en el ejemplo

```
-- Asegúrate de estar en la base de datos correcta
USE university;

-- Crear la tabla de historial de correos
CREATE TABLE email_history (
 email_history_id INT NOT NULL AUTO_INCREMENT,
 user_id INT NOT NULL,
 email VARCHAR(100) NULL,
 change_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP, -- Opcional: fecha del cambio
 PRIMARY KEY (email_history_id)
);
```

Crea un trigger llamado "tg_email" que guarda el email previo en la tabla "email_history" siempre, que se actualiza el campo "email" en la tabla "users"

```
-- Cambiamos el delimitador para poder usar ';' dentro del cuerpo del trigger
DELIMITER //

-- Eliminamos el trigger si existe
DROP TRIGGER IF EXISTS tg_email;

-- Creamos un trigger llamado 'tg_email'
-- Este trigger se activará después de una actualización (UPDATE) en la tabla 'users'
CREATE TRIGGER tg_email
 AFTER UPDATE ON users        -- Se ejecuta DESPUÉS de que se actualice una fila
 FOR EACH ROW                 -- Se aplica a CADA fila que sea modificada
BEGIN
 -- Comparamos el valor ANTIGUO del email con el NUEVO valor
 -- OLD.email: contiene el valor del email ANTES de la actualización
 -- NEW.email: contiene el valor del email DESPUÉS de la actualización
 -- Usamos '<>' para verificar si son diferentes (es lo mismo que '!=')
 IF OLD.email <> NEW.email THEN

 -- Si el email ha cambiado, insertamos un registro en la tabla de historial
 -- Guardamos el user_id y el email antiguo (OLD) para auditar el cambio
 INSERT INTO email_history (user_id, email)
```

```
 VALUES (OLD.user_id, OLD.email);  -- No usamos NEW.email porque queremos el valor anterior

    END IF;  -- Fin de la condición IF

END//  -- Fin del bloque BEGIN...END del trigger

-- Restauramos el delimitador predeterminado (;) para sentencias normales
DELIMITER ;
```

Actualizamos el campo "email" del usuario 1 la tabla "users" para probar el trigger

```
UPDATE users SET email = 'superdev@miemail.com' WHERE user_id = 1;
```

Visualizar el registro:

```
select * from email_history;
```

Ver TODOS los triggers de la base de datos

```
SHOW TRIGGERS;
```

Ver triggers de una tabla específica

```
SHOW TRIGGERS FROM university WHERE `Table` = 'users';
```

O más corto (si ya estamos usando la base de datos):

```
SHOW TRIGGERS LIKE 'users'; //LIKE busca en el nombre de la tabla, no en el nombre del trigger.
```

Ver el código SQL completo de un trigger

```
SHOW CREATE TRIGGER tg_email;
```

Elimina el trigger llamado "tg_email"

```
DROP TRIGGER tg_email;
```

7. Vistas

Una vista es una consulta guardada que se comporta como una tabla virtual. No almacena datos, solo la instrucción SQL. Cada vez que consultas la vista, MySQL ejecuta la consulta subyacente.

Analogía: Es como un atajo o un filtro permanente sobre tus tablas.

7.1 Crear una Vista Básica

Vista que muestra solo usuarios con email válido:

sql

```
USE university;

CREATE VIEW usuarios_con_email AS
SELECT
 user_id,
 name,
 surname,
 email
FROM users
WHERE email IS NOT NULL;
```

Usamos la vista:

sql

```
SELECT * FROM usuarios_con_email;
```

7.2. Vista con Cálculos

Vista que muestra nombre completo y edad calculada:

sql

```
CREATE VIEW usuarios_detallados AS
SELECT
 user_id,
 CONCAT(name, ' ', IFNULL(surname, '')) AS nombre_completo,
 age,
 init_date,
 email,
```

```
 CASE
 WHEN age < 18 THEN 'Menor de edad'
 WHEN age BETWEEN 18 AND 60 THEN 'Adulto'
 ELSE 'Adulto mayor'
 END AS categoria_edad,
    YEAR(CURDATE()) - YEAR(init_date) AS años_registrado
FROM users
WHERE init_date IS NOT NULL;
```

Realizamos la consulta:

sql

```
SELECT * FROM usuarios_detallados WHERE categoria_edad = 'Adulto';
```

7.3. Vista para Reportes Frecuentes

Vista que agrupa por nombre para contar repetidos:

sql

```
CREATE VIEW resumen_nombres AS
SELECT
 name,
 COUNT(*) AS total_usuarios,
 GROUP_CONCAT(DISTINCT surname ORDER BY surname) AS apellidos_asociados
FROM users
GROUP BY name;
```

Realizamos la consulta:

sql

```
SELECT * FROM resumen_nombres WHERE total_usuarios > 1;
```

7.4. Vista con JOIN (si tuvieras más tablas)

Simulando que tienes una tabla de departamentos:

sql

```
-- Vista hipotética (requiere tabla departments)
CREATE VIEW usuarios_por_departamento AS
SELECT
 u.user_id,
 CONCAT(u.name, ' ', u.surname) AS usuario,
 d.nombre AS departamento
FROM users u
LEFT JOIN departments d ON u.user_id = d.user_id;
```

7.5. Modificar una Vista

Sql

```
-- Reemplazar vista existente 7.1
CREATE OR REPLACE VIEW usuarios_con_email AS
SELECT
 user_id,
 name,
 surname,
 email,
 init_date
FROM users
WHERE email IS NOT NULL AND email LIKE '%@%';
```

7.6. Eliminar una Vista

sql

```
DROP VIEW IF EXISTS usuarios_con_email;
```

7.7. Ver Vistas Existentes

sql

```
-- Listar todas las vistas de la base de datos
```

```
SHOW FULL TABLES WHERE Table_type = 'VIEW';

-- Ver definición de una vista
SHOW CREATE VIEW usuarios_detallados;

-- Ver en INFORMATION_SCHEMA
SELECT TABLE_NAME
FROM INFORMATION_SCHEMA.VIEWS
WHERE TABLE_SCHEMA = 'university';
```

7.8. ¿Cuándo usar Vistas?

Situación	**¿Usar Vista?**
Consulta compleja que usas frecuentemente	Sí
Ocultar columnas sensibles a ciertos usuarios	Sí
Simplificar reportes para no programadores	Sí
Mejorar rendimiento (la vista no indexa por sí sola)	Revisar
Necesitas modificar datos con restricciones complejas	Mejor tabla temporal

8. Complementos

8.1 Acceder a los archivos WSL desde Windows

- Usamos el Explorador de archivos de Windows accediendo a la ruta \\wsl$\

8.2 Crear y Restaurar copias usando mysqldump (línea de comandos)

8.2.1 Exportar una base de datos completa

mysqldump -u usuario -p nombre_base_datos > archivo_backup.sql

Ejemplo

sudo mysqldump -u root -p university > university_backup.sql

8.2.2 Exportar solo la estructura (sin datos)

mysqldump -u root -p --no-data universidad > estructura.sql

8.2.3 Exportar una tabla específica

mysqldump -u root -p universidad estudiantes > tabla_estudiantes.sql

8.2.4 Exportar varias bases de datos

mysqldump -u root -p --databases universidad tienda > multiples_bases.sql

8.2.5 Exportar TODAS las bases de datos

mysqldump -u root -p --all-databases > todas_las_bases.sql

8.2.6 Opciones útiles adicionales

Opción	Qué hace
--routines	Incluye procedimientos almacenados y funciones
--triggers	Incluye triggers (por defecto sí, pero explícito es mejor)
--events	Incluye eventos programados
--single-transaction	Backup consistente sin bloquear tablas (InnoDB)

Opción	Qué hace
--lock-all-tables	Bloquea tablas durante el backup (MyISAM)
--add-drop-table	Agrega DROP TABLE IF EXISTS antes de cada CREATE
--if-not-exists	Usa CREATE TABLE IF NOT EXISTS

Ejemplo completo:

mysqldump -u root -p --single-transaction --routines --triggers --events universidad > universidad_completo.sql

8.3 Limpiar terminal SQL

Usamos dentro de la terminal de MySQL

\! clear

9. MySQL Workbench

Creamos un usuario para la conexión remota

CREATE USER 'wp_user' IDENTIFIED BY 'Unicatolica2025+';
*GRANT ALL PRIVILEGES ON *.* to 'wp_user'@'%';*
FLUSH PRIVILEGES;

Identicamos que los usuarios esten creados y que en host wp_user tenga %

SELECT user,host FROM mysql.user;

EXIT;

Permitimos que MySQL escuche ip externas, para eso editamos el archivo mysqld.cnf para cambiar la directiva bind-address a 0.0.0.0 o a colocamos la IP del servidor.

Usamos la terminal:

sudo vi /etc/mysql/mysql.conf.d/mysqld.cnf

Comentamos esta linea, adicionando # al inicio

 # bind-address = 127.0.0.1

Adicionamos esta linea

bind-address = 0.0.0.0

Reiniciamos MySQL para que tengan efectos los cambios.

En la terminal:

sudo systemctl restart mysql

Ahora probamos la conexión en MySQL Workbench.

Identificamos la IP de nuestro Ubuntu con :

En la terminal:

hostname -I

Con la IP, usuario y clave podremos probar la conexión en nuestro MySQL Workbench.

Fuentes:
https://josejuansanchez.org/bd/unidad-12-teoria/index.html