**SQL**

Gestor de base datos es aquel que nos permite interactuar con una base de datos

**Relacional**

Colección de tablas almacenando información de manera estructurada. Compartir datos redundantes. Cada tabla es una instancia, a su vez esta instancia tiene atributos de diferentes naturalezas que son representadas en columnas y para que haya relación entre tablas deben almacenar datos redundantes.

SQL

Prender server

→ mysql.server start -u root -p -h 127.0.0.1

Limpiar pantalla

→ system clear

Declarar variables

→ SET @curso = "Base de datos" @gestor = "MySQL";

Visualizar las variables

**→** SELECT @nombre, @curso, @gestor;

Variables sólo pertenecen a la sesión, otros usuarios que entren al server no podrán visualizarlas, así como si se apagan se pierde la info.

Mostrar las bases de datos

→ SHOW DATABASES;

Crear base de datos

→ CREATE DATABASE users;

**Seleccionar base de datos para trabajar**

→ USE <database name>;

**Eliminar la base de datos**

→ DROP DATABASE <database name>;

**Normalización**

Asegura evitar datos duplicados o columnas compuesta

**Crear una tabla**

CREATE TABLE authors (

author\_id INT,

nombre VARCHAR(25),

apellido VARCHAR(25),

genero CHAR(1),

fecha\_nacimiento DATE,

pais\_origen VARCHAR(40)

);

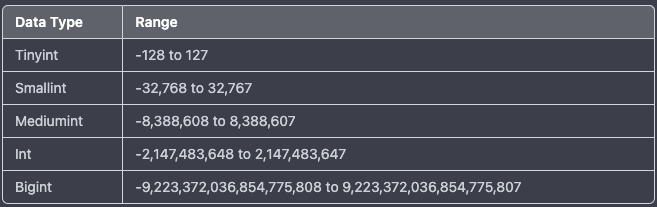
**Tipos de datos**

Alfanuméricos:

→ CHAR (0-255)

→ VARCHAR (0-2^16)

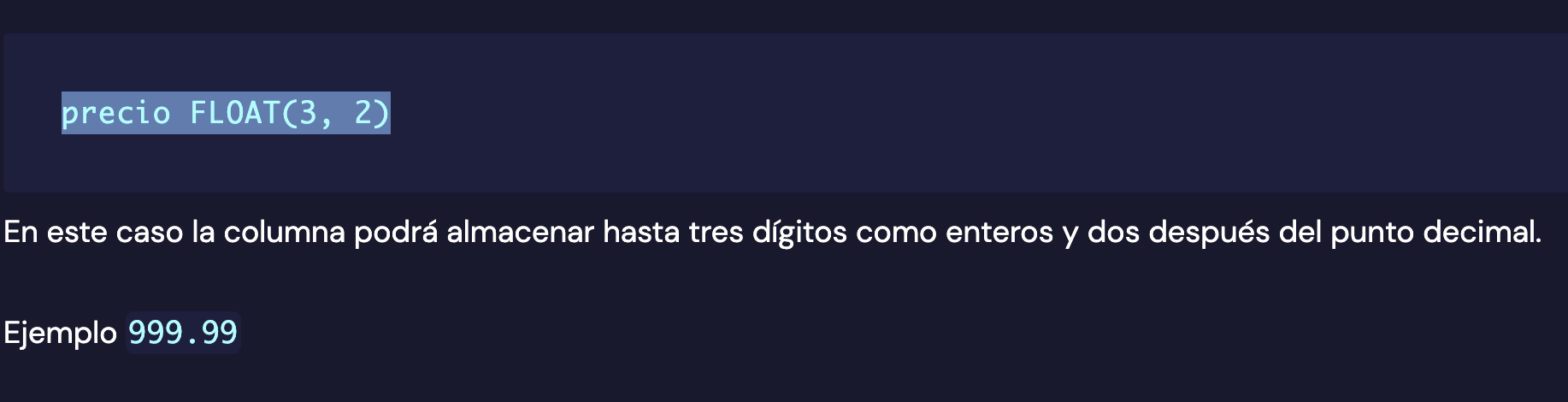
Números enteros:



Números flotantes:

→ Float

→ Double



Date



**Mirar las tablas y las columnas de la base de datos**

→ SHOW TABLES;

**Mostrar cuál es la base de datos actual que se está usando.**

**→** SELECT DATABASE();

**Eliminar tabla de base de datos:**

→ DROP TABLE <table\_name>

**Detallar las columnas de la tabla**

→ DESC <table\_name>

→ SHOW COLUMNS FROM <table\_name>

**Crear tablas a partir de otras**

→ CREATE TABLE <new\_table\_name> LIKE <old\_table\_name>

**Insertar registros**

**→** INSERT INTO authors (author\_id, nombre, apellido, genero, fecha\_nacimiento. pais\_origen)

**Insertar registros multilinea**

→ INSERT INTO authors (

author\_id,

nombre,

apellido,

genero,

fecha\_nacimiento,

pais\_origen)

VALUES (3, 'Alejandro', 'Estrada', 'M', '1997-09-30', 'Colombia'),

(4, 'Carolina', 'Ramirez', 'F', '1997-09-30', 'Colombia'),

(5, 'Isabel', 'Jurado', 'F', '1997-09-30', 'Colombia'),

(6, 'Meliza', 'Sanchez', 'F', '1997-09-30', 'Colombia'),

(7, 'Milena', 'Cruz', 'F', '1997-09-30', 'Colombia');

→ INSERT INTO authors() VALUES ();

**Listar registros**

**→** SELECT \* FROM authors;

Listado condicional:

SELECT nombre,

apellido,

IF(genero='M', 'Mero macho', 'Mera Nena') AS status\_migratorio

FROM authors;

**Ejecutar en archivos.sql sin estar autenticado**

→ mysql -u root -p < create-library-table.sql

Si estoy autenticado, posicionarse encima de la carpeta y poner

→ source <sql\_file> ;

**Condicionar**

Se puede condicionar la eliminación y creación de las bases de datos

→ DROP DATABASE IF EXISTS library\_aem;

→ CREATE DATABASE IF NOT EXISTS library\_aem;

→ CREATE TABLE IF NOT EXISTS authors (

author\_id INT,

nombre VARCHAR(25),

apellido VARCHAR(25),

genero CHAR(1),

fecha\_nacimiento DATE,

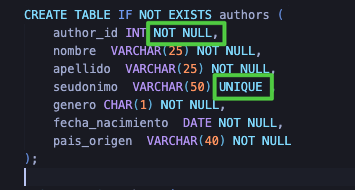
pais\_origen VARCHAR(40)

);

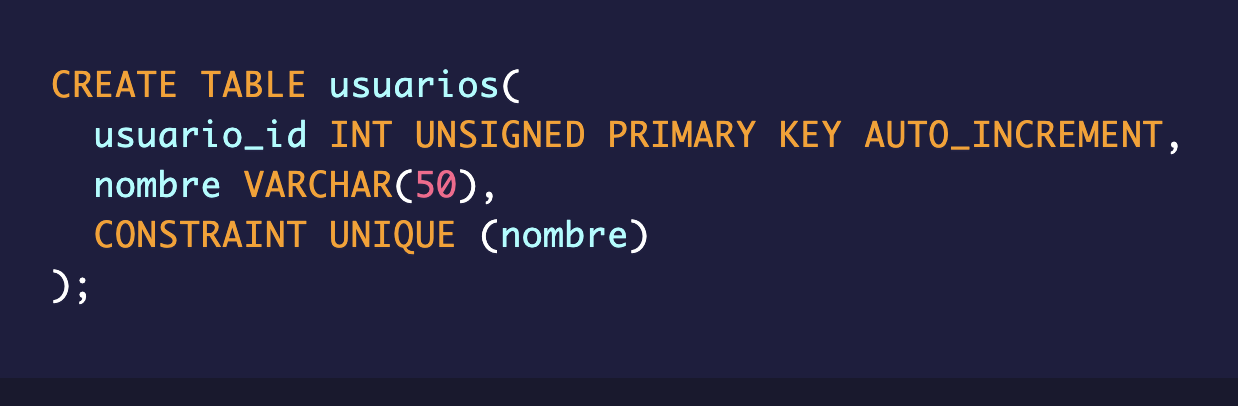
**Ejecutar sentencias sin iniciar sesión en mysql**

→ mysql -u root -p library\_aem -e “SELECT \* FROM authors”

**Evitar valores no nulos y únicos (por defecto tiene la naturaleza de null)**

****

Otra manera de poner los valores únicos

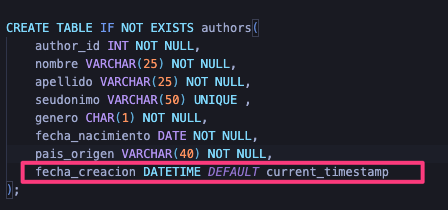


La palabra “constraint” es opcional pero se recomienda por legibilidad.

*CONSTRAINT* unique\_combinacion UNIQUE (nombre, apellido, genero)

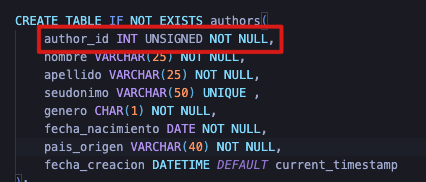
Chequea que la combinación sea unica, pero no agrega columna nueva

**Valores por defecto**

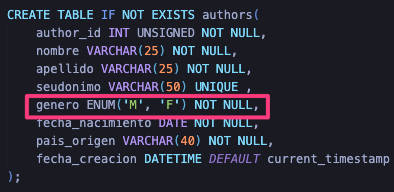


Estos valores por defecto se le pueden agregar a cualquier primitivo

**Para poner valores positivos o sin signo**



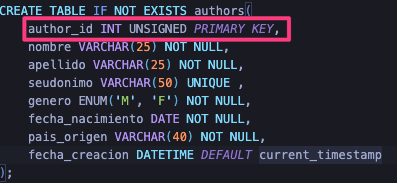
**Enum o lista de valores**

****

Tener en cuenta que es insensible en mayúsculas o minúsculas, es decir, ‘m’ o ‘f’ los identifica como tal ‘M’ o ‘F’ no hay distinción alguna.

**Primary key**

PRIMARY KEY = UNIQUE + NOT NULL



Si se quiere poner autoincrementable:

author\_id INT UNSIGNED **PRIMARY KEY AUTO\_INCREMENT**

**Relación entre tablas**

Mediante data redundante o llaves foráneas. Referencias entre tablas.

**Llaves foráneas**

Es una llave que hace referencia a una llave primaria de otra tabla. Y esa relación a la otra tabla, es decir, la llave foránea debe ser una llave primaria válida o existente. (Integridad referencial)

CREATE TABLE books(

libro\_id INT UNSIGNED *PRIMARY KEY* AUTO\_INCREMENT,

author\_id INT UNSIGNED NOT NULL,

titulo VARCHAR(50) NOT NULL,

descripcion VARCHAR(250),

paginas INT UNSIGNED,

fecha\_publicacion DATE NOT NULL,

fecha\_creacion DATETIME *DEFAULT* CURRENT\_TIMESTAMP,

*FOREIGN KEY*(author\_id) *REFERENCES* authors(author\_id)

);

INSERT INTO books(

author\_id,

titulo,

fecha\_publicacion

)

VALUES (1, 'El psicoanalista', '2012-09-30'),

(1, 'El psicoanalista 2', '2012-09-30'),

(1, 'El psicoanalista 3', '2012-10-30'),

(1, 'El psicoanalista 4', '2012-11-30');

**Agregar columnas después de estas estar creadas**

→ ALTER TABLE books ADD ventas INT UNSIGNED NOT NULL;

Como se agregó not null, genera la nueva columna con valores de 0, si no hubiéramos puesto not null tendrá todos los valores con null.

Si queremos poner valores por defecto:

→ ALTER TABLE books ADD ventas INT UNSIGNED NOT NULL DEFAULT 10;

**Eliminar columna**

→ ALTER TABLE books DROP COLUMN ventas;

**Renombrar tabla de books a stories**

→ ALTER TABLE books RENAME COLUMN stories;

El adjetivo UNIQUE funciona en estos comandos.

**Modificar el tipo de dato de una columna**

→ ALTER TABLE usuarios MODIFY telefono VARCHAR(50);

**Agregar una llave foránea a la tabla de usuarios con referencia a la tabla de grupos**

→ ALTER TABLE usuarios ADD FOREIGN KEY(grupo\_id) REFERENCES grupos(grupo\_id);

**Eliminar llaves foráneas**

→ ALTER TABLE usuarios DROP FOREIGN KEY grupo\_id;

**Sentencias en SQL**

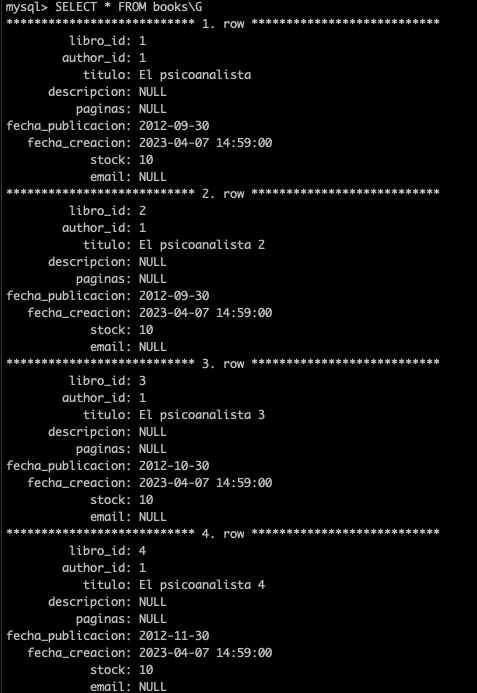
→ SELECT \* FROM books

**Para obtener la salida del select en forma de carta:**

→ SELECT \* FROM books\G

**Esto no es posible porque tag es una columna condicionada**

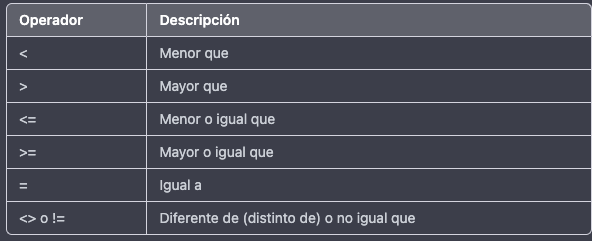
→ SELECT nombre, IF(genero='F', 'LOCA', 'LOCO') AS tag FROM authors WHERE tag = 'LOCO';



**WHERE**

SELECT \* FROM books WHERE titulo = ‘Carrie’ ;

en las “where” clause podemos utilizar los siguientes operadores condicionales



Tambien adicionado los condicionales semanticos

AND

OR

NOT

Example

→ SELECT \* FROM books WHERE (autor\_id = 1 AND titulo = ‘Carrie’) OR (autor\_id = 5 AND titulo = ‘El hobbit’)

**null no puede tratarse como un tipo de dato, para usar lo en condicionales podemos poner “IS” y a su vez “NOT” para negar el null.**

→ SELECT \* FROM autores WHERE seudonimo IS NOT NULL;

**Otra estrategia para trabajar con los tipo null es usando el “ si solo si ” <=>**

→ SELECT \* FROM autores WHERE seudonimo <=> NOT NULL;

**Recomendable usar “IS”, mejor legibilidad y también puede usarse con NOT**

**Para trabajar con rango BETWEEN**

**→** SELECT titulo FROM libros WHERE fecha\_publicacion BETWEEN '2000-01-01' AND '2020-12-31';

**Obtener registros a través de una lista como referencia**

**SELECT \* FROM libros WHERE titulo IN ('Ojos de fuego', 'Cujo', 'El hobbit', 'La torre oscura 7 La torre oscura');**

**Evitar valores repetidos**

→ SELECT DISTINCT titulo FROM libros;

**Alias son nombre que se le pueden asignar a tablas, columnas o funciones**

→ SELECT DISTINCT titulo **AS** libro FROM libros;

→ SELECT 9/3 AS resultado;

**Actualizar registros**

→ UPDATE libros SET descripcion = 'Nuevo cambio', ventas = 50; (cuidado si no se le pone where actualiza todos los registros con los atributos del set, para filtrar la cantidad de registros debemos de usar WHERE)

→ UPDATE libros SET descripcion = 'Nuevo cambio', ventas = 50 WHERE titulo = “El hobbit” ;

**Protip:**

Se puede actualizar condicionado:

→ UPDATE libros SET descripcion = IF(getSells() > 300, 'Disponible', 'No disponible' );

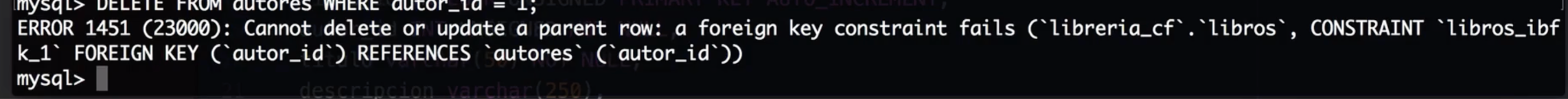
**Eliminar un grupo de registros**

→ DELETE FROM libros; (eliminaría todos los registros de la tabla pues no específica ningún condicional)

→ DELETE FROM libros WHERE author\_id = 1;

→ DELETE FROM libros WHERE libro\_id = 55;

Llaves primarias que están siendo como foráneas entre tablas, no se pueden eliminar de manera directa, para esto se debe hacer:

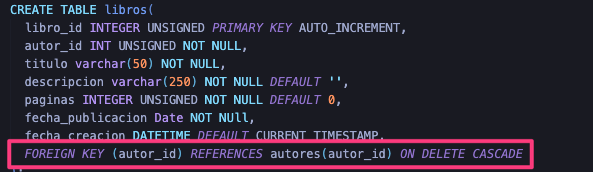


Se deben eliminar los registros que utilice su llave primaria.

1. DELETE FROM libros WHERE autor\_id = 1;
2. DELETE FROM autores WHERE autor\_id = 1;

Sin embargo, esto repercute en usar más queries, para ello usaremos eliminación en cascada

* Eliminación en cascada:



Autores → Libros (autor puede tener más de un libro, a libro le pongo “ON DELETE CASCADE”

**Pero eliminar libros no implica eliminar el autor**

**Para realizar en cascada sin tener que recrear la tabla:**

→ ALTER TABLE libros ADD FOREIGN KEY (autor\_id) REFERENCES autores(autor\_id) ON DELETE CASCADE;

**Para eliminar todos los registros de una tabla:**

→ *TRUNCATE TABLE libros* (no puede usar where, registros y metadatos se eliminan); == *DELETE FROM libros*; (aquí los metadatos se conservan entonces los ids los tiene en cuenta por lo que si se agrega un registro nuevo aunque se haya eliminado la tabla con 20 registros un nuevo insert hará que empiece la tabla por el id 21, en cambio con truncate si empezaria con un 1)

Con delete es posible recuperar los datos eliminados.

**Funciones**

**Concat**

→ CONCAT(nombre,' ',apellido) Se utiliza para concatenar una cantidad n de caracteres y columnas.

→ SELECT CONCAT(nombre,' ',apellido) AS 'Nombre\_Completo' FROM autores;

**Length**

→ SELECT LENGTH(“Hola Mundo”);

→ SELECT \* FROM autores WHERE LENGTH(nombre) > 7;

**Upper & Lower & Trim**

**→** SELECT UPPER(nombre) as 'NAME', UPPER(apellido) as 'LAST\_NAME', TRIM(CONCAT(UPPER(nombre), ' ', LOWER(apellido), ' ', 'siuu')) AS 'union' FROM autores;

**Left & Right**

**→** SELECT LEFT("hoLA COMOE SASTDA", 4) AS left\_substring, RIGHT("HOAL BABY HACE RATO",5)

→ SELECT \* FROM libros WHERE LEFT(titulo, LENGTH("Harry Potter")) = "Harry Potter";

**Rand (0-1) #Random number y round aproxima.**

→ SELECT ROUND(RAND() \* 100);

ROUND(2.345, 1) = 2.3 (aproximado por debajo)

ROUND(2.345, 2) = 2.35 (aproximado por encima)

ROUND(2.345, 3) = 2.345 (sin redondeo)

**Truncate (deja la cantidad de números después del punto)**

→ SELECT TRUNCATE(1.23456,2); => 1.23

**Pow numero elevado al segundo argumento POW(2,16) 2^16**

→ SELECT POW(2,16);

**Funciones sobre fechas**

→ SET @now = NOW();

→ SELECT SECOND(@now),

MINUTE(@now),

HOUR(@now ),

MONTH(@now),

YEAR(@now)

→ SELECT DAYOFWEEK(@now ), DAYOFMONTH(@now ), DAYOFYEAR(@now );

Change timestamp → to date

→ SELECT DATE(NOW()); (permite cambiar el timestamp a date “2032-02-22”

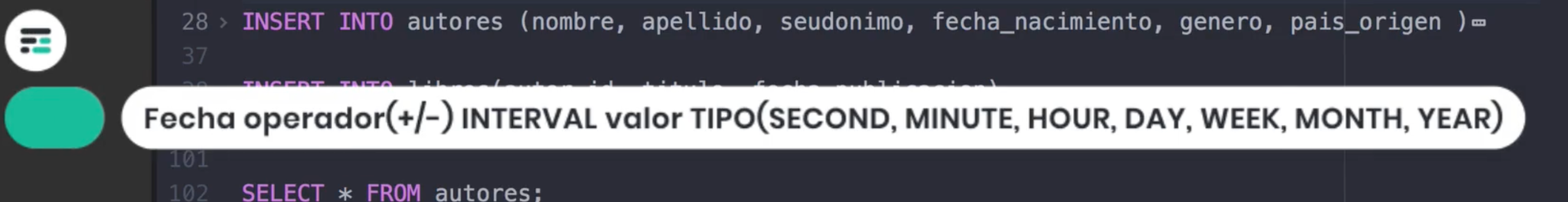
→ SELECT \* FROM libros WHERE DATE(fecha\_creacion) > "2023-04-11" ;

→ SELECT \* FROM libros WHERE DATE(fecha\_creacion) = CURDATE() ;

CURDATE → Fecha de hoy. Importante resaltar que tanto CURDATE como NOW deben ser usados en funciones tipo SELECT como mínimo

**Sumar días**

→



**Funciones sobre condicionales**

**IF clause**

→ SELECT IF(10 > 90, “El número es mayor a 10”, “El número es mayor a 10”);

El if tambien se puede asociar a columnas

→ SELECT IF(columna\_b = 0, ‘No posee paginas’, ‘Posee paginas’);

**IFNULL()**

→ SELECT IFNULL(seudonimo, 'El autor no cuenta con seudonimos') FROM autores;

En casi que si tenga seudonimos devuelve el nombre del autor

**Crear funciones**

→ SET GLOBAL log\_bin\_trust\_function\_creators = 1;

→ DELIMITER %

→ CREATE FUNCTION add\_days(fecha DATE, dias INT)

RETURNS DATE

BEGIN

RETURN fecha + INTERVAL dias DAY;

END%

→ DELIMITER ;

→ SELECT add\_days(NOW(), 60);

**Listar funciones**

**→** SHOW FUNCTION STATUS WHERE db = database() AND type = 'FUNCTION';

**Eliminar funcion**

**→** DROP FUNCTION add\_days;

**Sentencias avanzadas**

**Like**

→ SELECT \* FROM libros WHERE titulo LIKE 'Harry Potter%'; (El porcentaje aparece en la parte derecha porque se quiere obtener todos los registros que empiecen con los strings Harry Potter)

→ SELECT \* FROM libros WHERE titulo LIKE '%anillo'; (El porcentaje aparece en la parte izquierda porque se quiere obtener todos los registros que terminen con el string anillo);

**Si no sabes donde se encuentra, ponemos %%**

→ SELECT \* FROM libros WHERE titulo LIKE '%Harry%'; saca los mismos resultados que

SELECT \* FROM libros WHERE titulo LIKE 'Harry Potter%';

**→ Enunciado:**

**Todos los títulos cuyo palabra tiene 5 caracteres ]y la tercera sea la letra b**

→ SELECT \* FROM libros WHERE titulo LIKE **'\_\_b\_\_';**

**Todos los títulos cuyo segundo carácter tenga la letra a**

**→** SELECT \* FROM libros WHERE titulo LIKE '\_ \_a%';

**Expresiones Regulares**

→ Libros cuyo título empiece con H o con L

**Para evitar muchos usos LIKE concatenados o con LEFT y RIGHT**

SELECT titulo FROM libros WHERE titulo REGEXP '^[HL]';

**Ordenar Registros**

→ SELECT titulo FROM libros ORDER BY titulo DESC; ( de la z-a)

→ SELECT titulo FROM libros ORDER BY titulo; ( de la a-z, por defecto en ascedente)

→ SELECT libro\_id, titulo FROM libros ORDER BY libro\_id DESC;

**Limitar la cantidad de registros**

→ SELECT titulo FROM libros limit 10;

→ SELECT titulo FROM libros WHERE autor\_id = 2 LIMIT 5;

podemos paginar resultados con limit

→ SELECT libro\_id, titulo FROM libros LIMIT 0, 5; ( 0 → desde cual, 5 → Cantidad de registros)

**Funciones de agregación**

Funciones que se ejecutan cuando se obtiene un set de datos a través de una sentencia inicial.

**COUNT()**

Cuenta de registros

→ SELECT COUNT(\*) AS total FROM autores WHERE seudonimo IS NOT NULL;

Funciones de agregación solo actúan con registros que tienen el valor diferente de NULL, por lo que la anterior sentencia es igual a:

→ SELECT COUNT(seudonimo) AS total FROM autores;

**SUM(), MAX(), MIN(), AVG()**

DELIMITER %

CREATE FUNCTION getSells()

RETURNS INT

BEGIN

RETURN (SELECT (ROUND(RAND() \* 100) \* 6)) ;

END%

DELIMITER ;

UPDATE libros SET ventas = getSells();

SELECT ventas FROM libros;

SELECT SUM(ventas) FROM libros;

SELECT **MAX**(ventas) FROM libros;

SELECT **MIN**(ventas) FROM libros;

SELECT **AVG**(ventas) FROM libros;

**Agrupamiento**

Si queremos visualizar columnas que vienen de funciones de agregación, entonces se debe de agrupar (GROUP BY), es decir:

→ SELECT autor\_id, SUM(ventas) FROM libros GROUP BY autor\_id;

El autor que mas vendio;

→ SELECT autor\_id, SUM(ventas) as total FROM libros GROUP BY autor\_id ORDER BY total DESC LIMIT 1;

**Condiciones bajo agrupamiento HAVING()**

Autores que haya vendido más de 100 libros, el HAVING() es usado cuando se agrupa un set de datos mediante GROUP BY() y usando funciones de agregacion.

→ SELECT autor\_id, SUM(ventas) as total FROM libros GROUP BY autor\_id HAVING SUM(ventas) > 500;

**Operador UNION**

Combinar múltiples datasets

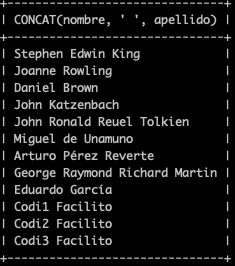
→ SELECT CONCAT(nombre, ' ', apellido) FROM autores



→



→



→ SELECT CONCAT(nombre, ' ', apellido) FROM autores

UNION

SELECT CONCAT(nombre, ' ', apellidos) FROM usuarios;

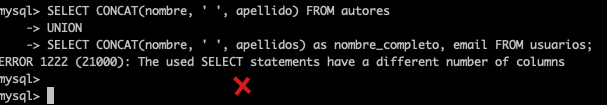
Es la unión de registros, los pone debajo del último del primer agrupamiento

Tener en cuenta que en la unión ambos deben de contar con la misma cantidad de columnas, por lo que si en una sentencia tenemos más columnas que en la otra, simplemente agregar espacios vacíos.

→ SELECT CONCAT(nombre, ' ', apellido) FROM autores

UNION

SELECT CONCAT(nombre, ' ', apellidos) as nombre\_completo, email FROM usuarios;



**Solucion:**

**→** SELECT CONCAT(nombre, ' ', apellido) AS nombre\_completo, "" AS email FROM autores UNION SELECT CONCAT(nombre, ' ', apellidos) as nombre\_completo, email FROM usuarios;



**Sub Consultas**

Consultas anidadas

**Nombre completo de los autores cuyas ventas hayan superado el promedio**

SELECT AVG(ventas) FROM libros;

SET @promedio = (SELECT AVG(ventas) FROM libros);

SELECT autor\_id

FROM libros

GROUP BY autor\_id

HAVING SUM(ventas) > @promedio;

**Si queremos hacer consulta anidad**

SELECT autor\_id

FROM libros

GROUP BY autor\_id

HAVING SUM(ventas) > (SELECT AVG(ventas) FROM libros);

**Ahora Para consultar el nombre completo, otra subconsulta**

SELECT CONCAT(nombre, ' ', apellido)

FROM autores

WHERE autor\_id

IN (

SELECT autor\_id

FROM libros

GROUP BY autor\_id

HAVING SUM(ventas) > (SELECT AVG(ventas) FROM libros));

**SELECT AVG(ventas) FROM libros;**

**SET @promedio = (SELECT AVG(ventas) FROM libros);**

**SELECT autor\_id**

**FROM libros**

**GROUP BY autor\_id**

**HAVING SUM(ventas) > @promedio;**

**Si queremos hacer consulta anidad**

**SELECT autor\_id**

**FROM libros**

**GROUP BY autor\_id**

**HAVING SUM(ventas) > (SELECT AVG(ventas) FROM libros);**

**Ahora Para consultar el nombre completo, otra subconsulta**

**SELECT CONCAT(nombre, ' ', apellido)**

**FROM autores**

**WHERE autor\_id**

**IN (**

**SELECT autor\_id**

**FROM libros**

**GROUP BY autor\_id**

**HAVING SUM(ventas) > (SELECT AVG(ventas) FROM libros));**

Validar registros

→ SELECT IF(

EXISTS(SELECT libro\_id FROM libros WHERE titulo = 'El hobbit' ),

'Disponible',

'No disponible'

) AS validator;

**Unir dos columnas sin ninguna relacion**

**SELECT \* FROM**

**(SELECT SUM(con\_seudonimo) AS con\_seudonimo FROM (SELECT COUNT(\*) AS con\_seudonimo FROM libros WHERE autor\_id IN (**

**(SELECT autor\_id FROM autores WHERE seudonimo IS NOT NULL)**

**) GROUP BY autor\_id) AS seudonimo\_tabla) AS seudonimo\_tabla\_2 ,**

**(SELECT SUM(sin\_seudonimo) AS sin\_seudonimo FROM (SELECT COUNT(\*) AS sin\_seudonimo FROM libros WHERE autor\_id IN (**

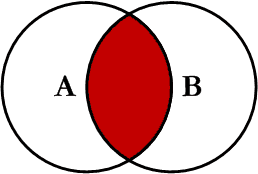
**(SELECT autor\_id FROM autores WHERE seudonimo IS NULL)**

**) GROUP BY autor\_id) AS sin\_seudonimo\_tabla) AS sin\_seudonimo\_tabla\_2;**

**JOINS**

**inner join**

Título de todos libros y nombre completo del autor



→ SELECT titulo FROM libros

INNER JOIN autores ON libros.autor\_id = autores.autor\_id;

Una vez hayamos usado las tablas con gracias a la cláusula ON, podremos visualizar columnas de ambas tablas

→ SELECT

libros.titulo,

CONCAT(autores.nombre, ' ', autores.apellido) AS nombre\_autor,

libros.fecha\_creacion

FROM libros

INNER JOIN autores ON libros.autor\_id = autores.autor\_id;

Si las tablas están con nombres muy complejos podemos ponerle alias y trabajar con ellos

**Using**

Si el nombre de las columnas tienen el mismo nombre podemos usar USING en vez de ON

→ Sin Alias

SELECT

libros.titulo,

CONCAT(autores.nombre, ' ', autores.apellido) AS nombre\_autor,

libros.fecha\_creacion

FROM libros

INNER JOIN autores **ON libros.autor\_id = autores.autor\_id;**

**Manejando el shortcut**

**→**

SELECT

libros.titulo,

CONCAT(autores.nombre, ' ', autores.apellido) AS nombre\_autor,

libros.fecha\_creacion

FROM libros

INNER JOIN autores USING(autor\_id);

Pero se usará más el **ON** porque esta cláusula permite condicionar

→

SELECT

libros.titulo,

CONCAT(autores.nombre, ' ', autores.apellido) AS nombre\_autor,

libros.fecha\_creacion

FROM libros

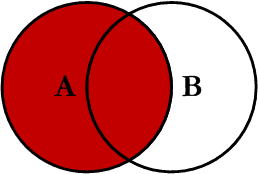
INNER JOIN autores ON libros.autor\_id = autores.autor\_id

AND autores.seudonimo IS NOT NULL;

con **using**  no se podría condicionar columnas

**Left Join or Left outer Join The same**

Cuando hay relacion muchos a muchos se debe generar otra tabla



Nombre completo de los usuarios y el id de los libros prestados

→ SELECT

CONCAT(nombre, " ", apellidos),

libros\_usuarios.libro\_id

FROM usuarios

LEFT JOIN libros\_usuarios ON usuarios.usuario\_id = libros\_usuarios.usuario\_id;

Trae tanto los usuarios que cumplen con la condición como los que no. Para filtrarlo sin los usuarios que no han prestado

→

SELECT

CONCAT(nombre, " ", apellidos),

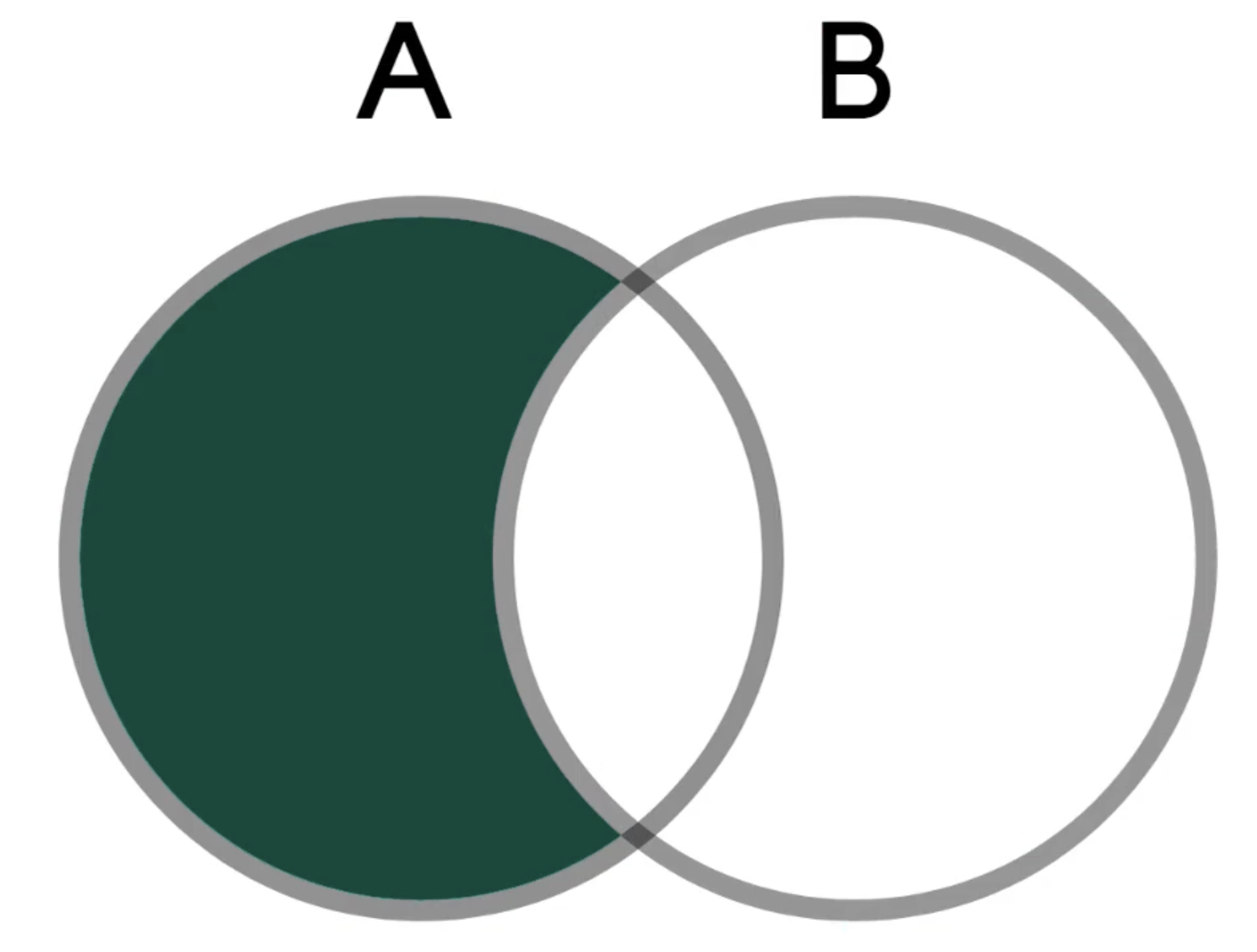
libros\_usuarios.libro\_id

FROM usuarios

LEFT JOIN libros\_usuarios ON usuarios.usuario\_id = libros\_usuarios.usuario\_id

WHERE libros\_usuarios.libro\_id ∫

Si queremos los usuarios que no han prestado libros



SELECT

CONCAT(nombre, " ", apellidos),

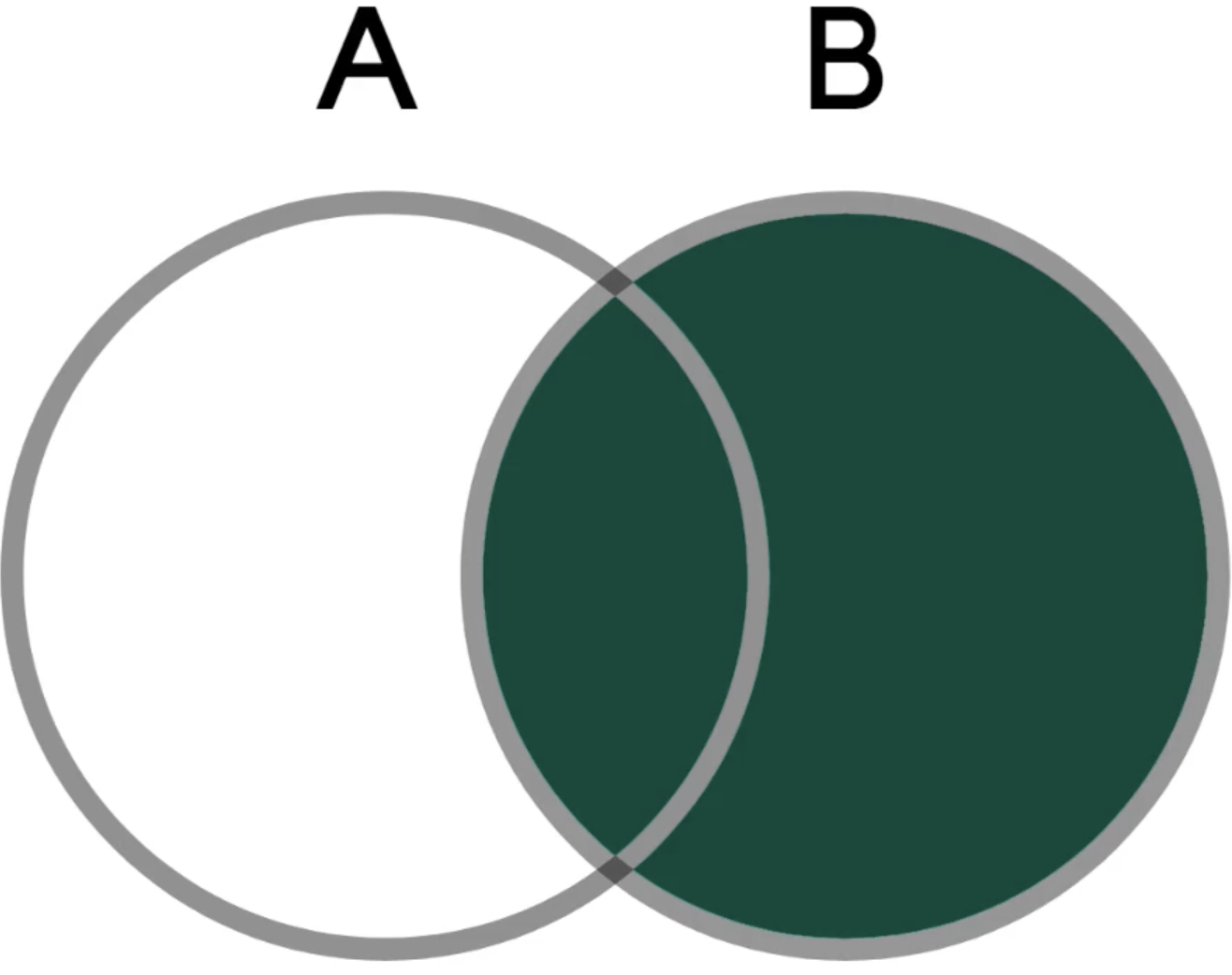
libros\_usuarios.libro\_id

FROM usuarios

LEFT JOIN libros\_usuarios ON usuarios.usuario\_id = libros\_usuarios.usuario\_id

WHERE libros\_usuarios.libro\_id **IS NULL;**

**Right Join or Right outer Join The same**

****

Intersección o valores de la tabla b

**→ Rigth**

Siguiendo el mismo ejemplo debe hacer cambios

libros\_usuarios = a

usuarios = b

SELECT

CONCAT(usuarios.nombre, " ", usuarios.apellidos),

libro\_id

FROM libros\_usuarios

RIGHT JOIN usuarios ON usuarios.usuario\_id = libros\_usuarios.usuario\_id;

SELECT

CONCAT(usuarios.nombre, " ", usuarios.apellidos),

libro\_id

FROM libros\_usuarios

RIGHT JOIN usuarios ON usuarios.usuario\_id = libros\_usuarios.usuario\_id

WHERE libros\_usuarios.libro\_id IS NOT NULL;

**→ Rigth outer join**

SELECT

CONCAT(usuarios.nombre, " ", usuarios.apellidos),

libro\_id

FROM libros\_usuarios

RIGHT JOIN usuarios ON usuarios.usuario\_id = libros\_usuarios.usuario\_id

WHERE libros\_usuarios.libro\_id IS NULL;

**Multiple joins**

Obtener nombre completo de usuarios que hayan prestado un libro, el libro debe estar escrito por un autor con seudónimo, además de que el prestamo se tuvo que realizar el dia de hoy

Las tablas que se usaran son:

1. usuarios

2. libros\_usuarios

3. libros

4. autores

→ SELECT DISTINCT

CONCAT(usuarios.nombre, ' ', usuarios.apellidos) as "Nombre Completo reservador",

libros.titulo

FROM usuarios

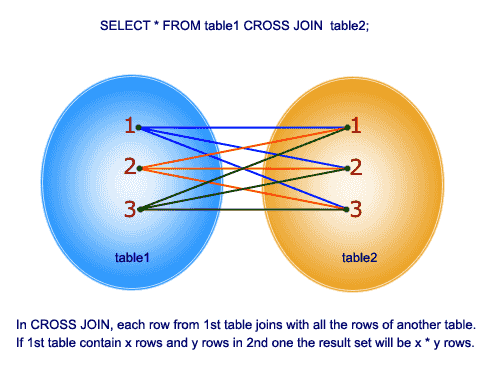
INNER JOIN libros\_usuarios ON libros\_usuarios.usuario\_id = usuarios.usuario\_id AND CURDATE() = DATE(libros\_usuarios.fecha\_creacion)

INNER JOIN libros ON libros\_usuarios.libro\_id = libros.libro\_id

INNER JOIN autores ON libros.autor\_id = autores.autor\_id AND autores.seudonimo IS NOT NULL;

**Productos cartesianos**

Unir múltiples tablas sin especificar funciones de relación



→ SELECT usuarios.username, libros.titulo FROM usuarios CROSS JOIN libros;

Puede servir para rellenar tablas

**Vistas**

Datos sensibles y mantener información de tabla. Mecanismo para consultar vistas filtradas y no almacenan u ocupan espacio en memoria.

Antes de generar la vista debemos de formular nuestra consulta.

→ SELECT

usuarios.usuario\_id,

usuarios.nombre,

usuarios.email,

usuarios.username,

COUNT(usuarios.usuario\_id) AS total\_prestamos

FROM usuarios

INNER JOIN libros\_usuarios ON libros\_usuarios.usuario\_id = usuarios.usuario\_id

AND libros\_usuarios.fecha\_creacion >= CURDATE() - INTERVAL 5 DAY

GROUP BY usuarios.usuario\_id;

Para generar una vista a traves del query anterior

→ CREATE VIEW prestamos\_usuarios\_vw AS

SELECT

usuarios.usuario\_id,

usuarios.nombre,

usuarios.email,

usuarios.username,

COUNT(usuarios.usuario\_id) AS total\_prestamos

FROM usuarios

INNER JOIN libros\_usuarios ON libros\_usuarios.usuario\_id = usuarios.usuario\_id

AND libros\_usuarios.fecha\_creacion >= CURDATE() - INTERVAL 5 DAY

GROUP BY usuarios.usuario\_id;

Se renocimienda utilizar un estandar en el nombre para la identificacion

Para visualizar las VISTAS

→ SHOW TABLES;

→ SELECT \* FROM prestamos\_usuarios\_vw -> Se pueden usar todas las clausulas sql

Eliminar vista

→ DROP VIEW prestamos\_usuarios\_vw;

Para modificar una vista ya creada

→ CREATE OR REPLACE VIEW prestamos\_usuarios\_vw AS

SELECT

usuarios.usuario\_id,

usuarios.nombre,

usuarios.email,

usuarios.username,

COUNT(usuarios.usuario\_id) AS total\_prestamos

FROM usuarios

INNER JOIN libros\_usuarios ON libros\_usuarios.usuario\_id = usuarios.usuario\_id

AND libros\_usuarios.fecha\_creacion >= CURDATE() - INTERVAL 5 DAY

GROUP BY usuarios.usuario\_id;

**Ejercicios**

Las tablas que se usaran son:

1. usuarios

2. libros\_usuarios

3. libros

4. autores

**Obtener a todos los usuarios que han realizado un préstamo en los últimos diez días.**

SELECT DISTINCT CONCAT(usuarios.nombre, ' ', usuarios.apellidos) AS "Nombre Completo" FROM usuarios

INNER JOIN libros\_usuarios ON usuarios.usuario\_id = libros\_usuarios.usuario\_id AND libros\_usuarios.fecha\_creacion >= CURDATE() - INTERVAL 10 DAY;

**Obtener a todos los usuarios que no ha realizado ningún préstamo.**

SELECT DISTINCT CONCAT(usuarios.nombre, ' ', usuarios.apellidos) AS "Nombre Completo" FROM usuarios

LEFT JOIN libros\_usuarios ON usuarios.usuario\_id = libros\_usuarios.usuario\_id

WHERE libros\_usuarios.usuario\_id IS NULL;

**Listar de forma descendente a los cinco usuarios con más préstamos.**

SET @nombre\_completo = CONCAT(usuarios.nombre, ' ', usuarios.apellidos);

SELECT CONCAT(usuarios.nombre, ' ', usuarios.apellidos) AS 'Nombre Completo', COUNT(\*) AS 'Numero de prestamos' FROM usuarios

LEFT JOIN libros\_usuarios ON usuarios.usuario\_id = libros\_usuarios.usuario\_id

WHERE libros\_usuarios.usuario\_id IS NOT NULL

GROUP BY libros\_usuarios.usuario\_id

ORDER BY CONCAT(usuarios.nombre, ' ', usuarios.apellidos) DESC;

**Listar 5 títulos con más préstamos en los últimos 30 días.**

SELECT libros.titulo, COUNT(\*) AS 'Cantidad de prestamos' FROM usuarios

INNER JOIN libros\_usuarios ON usuarios.usuario\_id = libros\_usuarios.usuario\_id

INNER JOIN libros ON libros\_usuarios.libro\_id = libros.libro\_id

GROUP BY libros\_usuarios.libro\_id

ORDER BY COUNT(\*) DESC

LIMIT 5;

**Obtener el título de todos los libros que no han sido prestados.**

SELECT libro\_id, titulo FROM libros WHERE libro\_id NOT IN (SELECT DISTINCT libros\_usuarios.libro\_id FROM usuarios

INNER JOIN libros\_usuarios ON usuarios.usuario\_id = libros\_usuarios.usuario\_id AND libros\_usuarios.fecha\_creacion);

**Obtener la cantidad de libros prestados el día de hoy.**

SELECT libros.titulo, COUNT(libros\_usuarios.libro\_id) AS "Veces prestadas" FROM usuarios

INNER JOIN libros\_usuarios ON usuarios.usuario\_id = libros\_usuarios.usuario\_id AND DATE(libros\_usuarios.fecha\_creacion) = DATE(NOW()) - INTERVAL 2 DAY

INNER JOIN libros ON libros\_usuarios.libro\_id = libros.libro\_id

GROUP BY libros\_usuarios.libro\_id;

**Obtener la cantidad de libros prestados por el autor con id 1.**

SELECT DISTINCT CONCAT(usuarios.nombre, ' ', usuarios.apellidos) AS 'Nombre Completo' FROM usuarios

INNER JOIN libros\_usuarios ON usuarios.usuario\_id = libros\_usuarios.usuario\_id AND usuarios.usuario\_id = 1;

**Obtener el nombre completo de los cinco autores con más préstamos.**

SELECT CONCAT(usuarios.nombre, ' ', usuarios.apellidos) AS 'Nombre Completo', COUNT(\*) AS 'Numero de prestamos' FROM usuarios

INNER JOIN libros\_usuarios ON usuarios.usuario\_id = libros\_usuarios.usuario\_id

GROUP BY libros\_usuarios.usuario\_id

ORDER BY CONCAT(usuarios.nombre, ' ', usuarios.apellidos) DESC

LIMIT 5;

**Obtener el título del libro con más préstamos esta semana.**

SELECT libros.titulo, COUNT(libros\_usuarios.libro\_id) AS "Veces prestadas" FROM usuarios

INNER JOIN libros\_usuarios ON usuarios.usuario\_id = libros\_usuarios.usuario\_id AND DATE(libros\_usuarios.fecha\_creacion) = DATE(NOW()) - INTERVAL 2 DAY

INNER JOIN libros ON libros\_usuarios.libro\_id = libros.libro\_id

GROUP BY libros\_usuarios.libro\_id

ORDER BY COUNT(libros\_usuarios.libro\_id) DESC

LIMIT 1;

**Stored Procedures**

Rutinas en el motor de base de datos, no retornan valores, pueden recibir parámetros. No son agnósticos, es decir, dependen del motor de base de datos.

Procedure para prestar libros

→ **Procedure**

DELIMITER %

CREATE PROCEDURE prestamo(usuario\_id INT, libro\_id INT)

BEGIN

INSERT INTO libros\_usuarios(libro\_id, usuario\_id) VALUES (libro\_id, usuario\_id);

UPDATE libros SET stock = stock - 1 WHERE libros\_usuarios.libro\_id = libro\_id;

END%

DELIMITER ;

→**Listar procedures**

SHOW PROCEDURE STATUS WHERE db = database() AND type = 'PROCEDURE';

→ **Ejecutar stored procedures**

CALL prestamo(20,2);

**→ Eliminar procedures**

DROP PROCEDURE prestamo;

**NOTA: No es posible editar directamente un procedure, se debe eliminar y luego crear el nuevo procedure con los nuevos cambios.**

→ **Obtener valores que se ejecutan dentro del procedure**

Agregamos argumento con bandera OUT

DELIMITER %

CREATE PROCEDURE prestamo(usuario\_id INT, libro\_id INT, **OUT cantidad INT)**

BEGIN

INSERT INTO libros\_usuarios(libro\_id, usuario\_id) VALUES (libro\_id, usuario\_id);

UPDATE libros SET stock = stock - 1 WHERE libros.libro\_id = libro\_id;

**SET cantidad = (SELECT stock FROM libros WHERE libros.libro\_id = libro\_id);**

END%

DELIMITER ;

**Para obtenerla desde afuera**

**Creamos una variable con cualquier valor random**

→ SET @cantidad = -1;

→ CALL prestamo(1, 20, @cantidad);

→ SELECT @cantidad;

**Stored procedures con condicionales**

DELIMITER %

CREATE PROCEDURE prestamo(usuario\_id INT, libro\_id INT, OUT cantidad INT)

BEGIN

SET cantidad = (SELECT stock FROM libros WHERE libros.libro\_id = libro\_id);

IF cantidad > 0 THEN

INSERT INTO libros\_usuarios(libro\_id, usuario\_id) VALUES (libro\_id, usuario\_id);

UPDATE libros SET stock = stock - 1 WHERE libros.libro\_id = libro\_id;

SET cantidad = cantidad - 1;

**ELSE IF condicion 2**

**ELSE IF condicion 3**

**ELSE IF condicion 4**

.

.

.

ELSE

SELECT "No es posible realizar el prestamo" AS mensaje\_error;

END IF;

END%

DELIMITER ;

**Stored procedures con cases semejante a switch**

DELIMITER %

CREATE PROCEDURE tipo\_lector(usuario\_id INT)

BEGIN

SET @cantidad = (SELECT COUNT(\*) FROM libros\_usuarios WHERE libros\_usuarios.usuario\_id = usuario\_id);

SELECT @cantidad;

CASE

WHEN @cantidad >= 20 THEN

SELECT "Fanatico" AS mensaje;

WHEN @cantidad >= 10 AND @cantidad < 20 THEN

SELECT "Fanatico" AS mensaje;

WHEN @cantidad >= 5 AND @cantidad < 10 THEN

SELECT "Fanatico" AS mensaje;

ELSE

SELECT "Nuevo" AS mensaje;

END CASE;

END%

DELIMITER ;

**Stored procedures con ciclos**

**Usando WHILE**

DELIMITER %

CREATE PROCEDURE libros\_azar()

BEGIN

SET @iteracion = 0;

WHILE @iteracion < 5 DO

SELECT libro\_id FROM libros ORDER BY RAND() LIMIT 1;

SET @iteracion = @iteracion + 1;

END WHILE;

END%

DELIMITER ;

**Usando REPEAT**

DELIMITER %

CREATE PROCEDURE libros\_azar()

BEGIN

SET @iteracion = 5;

REPEAT

SELECT libro\_id FROM libros ORDER BY RAND() LIMIT 1;

SET @iteracion = @iteracion + 1;

UNTIL @iteracion >= 5

END REPEAT;

END%

DELIMITER ;

**Por lo menos el ciclo se realizará una vez**

**Bloqueos**

Varios usuarios actuando sobre una misma base de datos. Bloqueos o cierres. Una porcion de la base de datos hasta que el proceso termine. Cuando sentencia es ejecuta tipo

consultar,actualizar insertar o eliminar. segundo tipo inserción, actualización o eliminación → esta no bloquea.

**Transacciones**

Tareas que impactan a más de una tabla a la vez. Mecanismo para agrupar sentencias en una sola, en caso de que nunca se cumpla una, ninguna se ejecuta. Atomicidad.

Antes, durante y después → estados

Transacciones

START TRANSACTION;

SET @libro\_id = 20, @usuario\_id = 3;

UPDATE libros SET stock = stock - 1 WHERE libro\_id = @libro\_id;

SELECT stock FROM libros WHERE libro\_id = @libro\_id;

INSERT INTO libros\_usuarios(libro\_id, usuario\_id) VALUES (@libro\_id, @usuario\_id);

SELECT \* FROM libros\_usuarios;

**COMMIT; → Finalizamos la transaccion**

Si no ejecuto commit puedo revertir los cambios

Ahora ejecutando un caso invalido

START TRANSACTION;

SET @libro\_id = 20, @usuario\_id = 100; El usuario no existe

UPDATE libros SET stock = stock - 1 WHERE libro\_id = @libro\_id;

SELECT stock FROM libros WHERE libro\_id = @libro\_id;

INSERT INTO libros\_usuarios(libro\_id, usuario\_id) VALUES (@libro\_id, @usuario\_id);

SELECT \* FROM libros\_usuarios;

ROLLBACK;

→ **La mayoria de veces utilizaremos las Transacciones en los stored procedures**

Stored procedure + transaction

DELIMITER %

CREATE PROCEDURE prestamo(usuario\_id INT, libro\_id INT)

BEGIN

DECLARE EXIT HANDLER FOR SQLEXCEPTION

BEGIN

ROLLBACK;

END;

START TRANSACTION;

INSERT INTO libros\_usuarios(libro\_id, usuario\_id) VALUES(libro\_id, usuario\_id);

UPDATE libros SET stock = stock - 1 WHERE libros.libro\_id = libro\_id;

COMMIT;

END%

DELIMITER ;

**Motores de almacenamiento**

Afortunadamente para nosotros, los administradores de base de datos, MySQL nos permite trabajar con diferentes motores de almacenamiento, entre los que destacan MyISAM e InnoDB.

¿Motor de almacenamiento?, ¿Qué es eso? 🤔, verás, un motor de almacenamiento se el encargado de almacenar, gestionar y recuperar toda la información de una tabla. Es por ello que es de suma importancia que nosotros conozcamos la existencia de estos motores, cuales son sus principales diferencias y en qué casos es bueno utilizar uno u otro, de esta forma que podamos garantizar un mejor performance en nuestras aplicaciones. 😉

Para que nosotros conozcamos que motor de almacenamiento podemos utilizar basta con ejecutar la siguiente sentencia en nuestra terminal.

→ SHOW ENGINES;

Obtendremos el siguiente listado.

InnoDB

MRG\_MYISAM

MEMORY

BLACKHOLE

MyISAM

CSV

ARCHIVE

PERFORMANCE\_SCHEMA

FEDERATED

En esta ocasión nos centraremos en explicar los dos motores de almacenamiento más populares, me refiero a MyISAM e InnoDB.

**MyISAM** es el motor por **default de MySQL**. Una de las principales ventajas de este motor es la velocidad al momento de recuperar información. **MyISAM** es una excelente opción cuando las sentencias predominantes en nuestra aplicación sean de consultas. Esta es una de las razones por las cuales MyISAM es tan popular en aplicaciones web.

**Si tu aplicación necesita realizar búsquedas full-text MyISAM es un mejor opcion.**

La principal desventajas de MyISAM recae en la ausencia de atomocidad, ya que no se comprueba la integridad referencial de los datos. Se gana tiempo en la inserción, sí, pero perdemos confiabilidad en los datos.

Por otro lado tenemos el motor de almacenamiento InnoDB. La principal ventaja de este motor recae en la seguridad de las operaciones. **InnoDB permite la ejecución de transacciones, esto nos garantiza que los datos se persisten de forma correcta y si existe algún error podamos revertir todos los cambios realizados.**

Algo interesante a mencionar sobre InnoDB es que este motor realiza un bloqueo total sobre un tabla cuando es ejecutada una se las siguientes sentencias.

* **Select**
* **Insert**
* **Update**
* **Delete**

Si deseamos trabajar con transacción y la integridad de los datos sea crucial nuestra mejor opción será InnoDB, por otro lado, sí lo que deseamos es una mayor rapidez al momento de obtener información será necesario utilizar MyISAM.

Gestión

Si nosotros así lo deseamos podemos cambiar el motor de almacenamiento. Existen dos formas de hacer esto. La primera, es modificar el archivo my.cnf.

[mysqld]

default-storage-engine = innodb

**La segunda forma es hacerlo directamente desde nuestra sección, basta con ejecutar la siguiente sentencia.**

→ SET storage\_engine=INNODB;

En ambos casos modificamos el motor de almacenamiento de MyISAM a InnoDB.

**Si nosotros deseamos conocer qué motor de almacenamiento utiliza una tabla en particular, podemos hacerlo ejecutando la siguiente sentencia.**

SHOW TABLE STATUS WHERE `Name` = 'tabla' \G;

**Si deseamos crear una tabla utilizando un motor en particular, debemos seguir la siguiente estructura.**

CREATE TABLE tabla\_innodb (id int, value int) ENGINE=INNODB;

CREATE TABLE tabla\_myisam (id int, value int) ENGINE=MYISAM;

CREATE TABLE tabla\_default (id int, value int);

**Eventos MYSQL**

eventos = Cron Jobs

Los eventos nos permitirán a nosotros cómo administradores de base de datos programar ciertas tareas las cuales queremos que se ejecuten de forma periódica o en un momento en concreto, de tal manera que podamos automatizar ciertos procesos.

Paso a paso para generar cron jobs en sql

Crear tabla

→ CREATE TABLE test(

evento VARCHAR(50),

fecha DATETIME

);

Habilitar nuestro servidor para ejecutar eventos

→ SET GLOBAL event\_scheduler = ON;

Crear evento (El nombre del evento no debe de poseer más de 64 caracteres.) Esta configuración sirve para ejecutarlo una vez después de un minuto de la hora actual

→ **CREATE EVENT** insertion\_event

**ON** **SCHEDULE AT** CURRENT\_TIMESTAMP + INTERVAL 1 MINUTE

**DO** INSERT INTO test VALUES ('Evento 1', NOW());

Si queremos que el evento se ejecute en una fecha en concreto, únicamente debemos de especificarlo en ON SCHEDULE AT. Recuerda, el formato es año-mes-día hora:minuto:segundo.

→ **CREATE EVENT** insertion\_event

**ON SCHEDULE AT** '2018-12-31 12:00:00'

**DO** INSERT INTO test VALUES ('Evento 1', NOW());

Si nuestro evento es con múltiples sentencias sql

→ DELIMITER //

**CREATE EVENT** insertion\_event

**ON SCHEDULE AT** CURRENT\_TIMESTAMP + INTERVAL 1 MINUTE

**DO**

**BEGIN**

INSERT INTO test VALUES ('Evento 1', NOW());

INSERT INTO test VALUES ('Evento 2', NOW());

INSERT INTO test VALUES ('Evento 3', NOW());

**END** //

DELIMITER ;

Listar eventos

→ SHOW events\G;

Eliminar eventos

→ DROP EVENT nombre\_evento;

Es importante mencionar que una vez el evento haya **expirado**, este, será eliminado de forma automática. Si nosotros no queremos que esto ocurra debemos de apoyarnos de **ON COMPLETION**

→ **ON SCHEDULE AT** CURRENT\_TIMESTAMP + INTERVAL 1 MINUTE

**ON COMPLETION PRESERVE**

…

Normalmente los eventos trabajan en conjunto con stored procedures

**→ CREATE EVENT** nombre\_evento

**ON SCHEDULE AT** 'fecha de ejeución'

**DO CALL** store\_procedure();

**Eventos Periódicos**

Eventos que se ejecutan con cierta frecuencia establecida.

En esta ocasión, el evento se ejecuta cada minuto después de las 6:30 PM. Podemos programar la ejecución para cada segundo, minuto, hora, semana, mes o año.

→ **CREATE EVENT** insertion\_event

**ON SCHEDULE EVERY 1 MINUTE STARTS** '2018-07-07 18:30:00'

**DO** INSERT INTO test VALUES ('Evento 1', NOW());

Si nosotros queremos que el evento se ejecute entre un rango de fechas debemos de apoyarnos de ENDS.

→ **CREATE EVENT** insertion\_event

**ON SCHEDULE EVERY 1 MINUTE STARTS** '2018-07-07 18:30:00'

**ENDS** '2018-07-07 19:00:00'

**DO** INSERT INTO test VALUES ('Evento 1', NOW());

Ahora, el evento se ejecuta durante un periodo de 30 minutos (De 6:30 PM a 7:00 PM).

**Editar eventos**

Deshabilitar eventos

→ ALTER EVENT nombre\_evento

DISABLE;

Para habilitar nuevamente

→ ALTER EVENT nombre\_evento

ENABLE;

**Detener todos los eventos**

→ SET GLOBAL event\_scheduler = OFF;

**Template de un evento en mysql**

→ CREATE

[DEFINER = { user | CURRENT\_USER }]

EVENT

[IF NOT EXISTS]

event\_name

ON SCHEDULE schedule

[ON COMPLETION [NOT] PRESERVE]

[ENABLE | DISABLE | DISABLE ON SLAVE]

[COMMENT 'string']

DO event\_body;

schedule:

AT timestamp [+ INTERVAL interval] ...

| EVERY interval

[STARTS timestamp [+ INTERVAL interval] ...]

[ENDS timestamp [+ INTERVAL interval] ...]

interval:

quantity {YEAR | QUARTER | MONTH | DAY | HOUR | MINUTE |

WEEK | SECOND | YEAR\_MONTH | DAY\_HOUR | DAY\_MINUTE |

DAY\_SECOND | HOUR\_MINUTE | HOUR\_SECOND | MINUTE\_SECOND}

**Cursores MYSQL**

En base de datos un Cursor es un mecanismo el cual nos permite procesar fila por fila el resultado de una consulta.

Como sabemos SQL es un lenguaje orientado a conjuntos. Si nosotros deseamos alterar ciertos elementos en nuestra colección tendremos que hacerlo mediante condiciones. Única y exclusivamente los elementos que cumpla con dichas condiciones podrán ser alterados. Con los cursores podremos trabajar con cada uno de los elementos (filas) de nuestra consulta sin tener que obtener nuevos conjuntos. Esto nos permitirá ser mucho más flexibles al momento de manipular la información.

Para nosotros poder hacer uso de un cursor será necesario seguir los siguientes pasos.

1. **Crear un cursor a partir de una sentencia SQL.**
2. **Apertura del cursor.**
3. **Acceso a datos.**
4. **Cierre del cursor.**

Es importante mencionar que en MySQL los cursores solo podrán ser utilizados dentro de **stored procedures.**

Permite manejar cada registro de cada tabla y generar una serie de acciones en vez de utilizar un sentencias para obtener tablas adyacentes;

Como sabemos SQL es un lenguaje orientado a conjuntos. Si nosotros deseamos alterar ciertos elementos en nuestra colección tendremos que hacerlo mediante condiciones. Única y exclusivamente los elementos que cumplan con dichas condiciones podrán ser alterados. Con los cursores podremos trabajar con cada uno de los elementos (filas) de nuestra consulta sin tener que obtener nuevos conjuntos. Esto nos permitirá ser mucho más flexibles al momento de manipular la información.

Para nosotros poder hacer uso de un cursor será necesario seguir los siguientes pasos.

Crear un cursor a partir de una sentencia SQL.

Apertura del cursor.

Acceso a datos.

Cierre del cursor.

Es importante mencionar que en MySQL los cursores solo podrán ser utilizados dentro de stored procedures.

Veamos un ejemplo.

1.-Crear un stored procedure el cual incremente en 10 el número de páginas de cada libro (tabla libros). En consola debemos visualizar el título, el anterior número de páginas y el actual número de páginas de cada libro.

Bien, este ejemplo nos permitirá comprender de una mejor manera el uso de cursores.

Veamos como pudiese quedar nuestro stored procedure.

→ DELIMITER //

DROP PROCEDURE IF EXISTS facilito\_procedure//

CREATE PROCEDURE facilito\_procedure()

BEGIN

DECLARE var\_id INTEGER;

DECLARE var\_paginas INTEGER;

DECLARE var\_titulo VARCHAR(255);

DECLARE var\_final INTEGER DEFAULT 0;

DECLARE cursor1 CURSOR FOR SELECT libro\_id, titulo, paginas FROM libros;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET var\_final = 1;

OPEN cursor1;

bucle: LOOP

FETCH cursor1 INTO var\_id, var\_titulo, var\_paginas;

IF var\_final = 1 THEN

LEAVE bucle;

END IF;

UPDATE libros SET paginas = var\_paginas + 10 WHERE libro\_id = var\_id;

SELECT

var\_titulo AS 'titulo',

var\_paginas AS 'Anterior',

paginas AS 'Incremento'

FROM libros WHERE libro\_id = var\_id;

END LOOP bucle;

CLOSE cursor1;

END//

DELIMITER ;

Dentro del stored procedure lo primero que hacemos es declarar todas las variables que usaremos. En este caso como trabajaremos con el título y el número de páginas de cada libro declaró tres nuevas variables. var\_id, var\_paginas, var\_titulo.

Para tener un control sobre la iteración de cada uno de los elementos en la consulta declaró una cuarta variable llamada var\_final, cuyo valor comienza en 0.

Una vez hemos declarado todas las variables procedemos a crear nuestro cursor. El cursor se creará a partir de una consulta SQL. En este caso la consulta no es demasiado compleja, sin embargo, si así lo deseamos podemos crear un cursos a partir de una consulta con joins, order, group etc...

Con la sentencia

DECLARE CONTINUE HANDLER FOR NOT FOUND SET var\_final = 1;

Indicamos que una vez todos los elementos (filas) dentro de nuestro cursor hayan sido iterados la variable var\_final tomará un nuevo valor, de 0 a 1.

Para comenzar la iteración de los elementos será necesario abrir el cursor.

→ OPEN cursor1;

La iteración la logramos utilizando un Loop, el cual será controlado a partir de la variable var\_final.

IF var\_final = 1 THEN

--Si ya no existen más elementos finalizamos el ciclo.

LEAVE bucle;

END IF;

Al nosotros usar la cláusula Fetch obtenemos el elemento (fila) actual de nuestra consulta y avanzamos al siguiente elemento.

En este caso como en nuestra consulta hemos obtenido 3 columnas (id, titulo, páginas) asignamos el valor de cada una de las columnas a nuestras variables. Es importante mencionar que los valores podrán ser asignados únicamente a variables del mismo tipo, no podremos asignar un varchar a una variable de tipo integer.

El orden es importante.

→ FETCH cursor1 INTO var\_id, var\_titulo, var\_paginas;

Una vez hemos realizado todas las tareas correspondientes y se han iterado todos los elementos (filas) el siguiente paso será cerrar el cursor.

close cursor1

Para ejecutar nuestro stored prcedure ejecutaremos la siguiente sentencia.

→ **call facilito\_procedure();**

**Respaldo de información**

mysqldump para respaldos de información

**Para realizar respaldo a una base de datos**

→ mysqldump base\_de\_datos > ruta/archivo\_respaldo.sql

**Para múltiples bases de datos**

→ mysqldump **--databases db1 db2 db3** > ruta/archivo\_respaldo.sql

**Asignar permisos a usuarios MYSQL**

Pasos para crear usuarios

Autenticarnos

→ mysql -u root -p -h localhost<ip>

Crear el usuario especifico

→ CREATE USER 'usuario'@'localhost' IDENTIFIED BY 'password';

Hasta este punto, nosotros ya podemos autenticarnos con el servidor utilizando el nuevo usuario, sin embargo, una vez autenticado las acciones que podemos hacer son mínimas, debido a que este usuario no posee los permisos necesarios para trabajar con las bases de datos.

Asignar permisos

**Otorgar permisos de administrador**

→ GRANT ALL PRIVILEGES ON \*.\* TO 'nombre\_usuario'@'localhost';

Los asteriscos indican que los permisos serán asignados a todas las bases de datos y a todas las tablas (primer asteriscos bases de datos, segundo asterisco tablas).

Si queremos dar ciertos permisos a una base de datos específica para todas las tablas internas

→ GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, DROP

ON codigofacilito.\*

TO 'nombre\_usuario'@'localhost';

En esta ocasión estamos indicando que el nuevo usuario podrá **consultar, crear, actualizar y eliminar registros**, así cómo podrá crear o eliminar elementos (tablas, índices, columnas, funciones, stores, etc ...)

Todos estos permisos serán válidos únicamente en la **base de datos codigofacilito** y se aplicarán a **todas las tablas.**

Si queremos asignar permisos únicamente a una tabla, **reemplazamos el asteriscos por el nombre de la tabla.**

**ejemplo codigofacilito.usuarios**

Luego refrescar los privilegios como paso final

→ FLUSH PRIVILEGES;

**Lista de permisos**

1. **CREATE** permite crear nuevas tablas o bases de datos.
2. **DROP** permite eliminar tablas o bases de datos.
3. **DELETE** permite eliminar registros de tablas.
4. **INSERT** permite insertar registros en tablas.
5. **SELECT** permite leer registros en las tablas.
6. **UPDATE** permite actualizar registros en las tablas.
7. **GRANT OPTION** permite remover permisos de usuarios.
8. **SHOW** **DATABASE** Permite listar las bases de datos existentes.

**Sentencias a tener en cuenta**

**Listado de todos los usuarios.**

SELECT User FROM mysql.user;

**Eliminar un usuario.**

DROP USER 'usuario'@'localhost';

**Remover permisos en concreto (Ejemplo update y delete).**

REVOKE UPDATE, DELETE ON \*.\* FROM 'usuario'@'localhost';

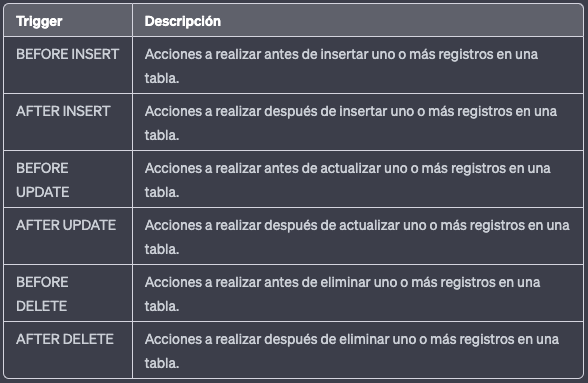
**Remover todos los privilegios.**

REVOKE ALL PRIVILEGES ON \*.\* FROM 'usuario'@'localhost';

**Triggers Mysql**

Acción automática que se ejecuta cuando se registra un evento de INSERT, UPDATE o DELETE.

Un trigger solo existe con una tabla. Los triggers pueden dispararse antes de que pasen ciertas acciones, tales como:

****

Es una combinación uno a muchos, es decir, para una tabla pueden haber n cantidad de triggers. Un trigger le pertenece a única y exclusiva tabla.

**Ventajas de los triggers**

1. Con los triggers seremos capaces de validar todos aquellos valores los cuales no pudieron ser validados mediante un constraints, asegurando así la integridad de los datos.
2. Los triggers nos permitirán ejecutar reglas de negocios.
3. Utilizando la combinación de eventos nosotros podemos realizar acciones sumamente complejas.
4. Los trigger nos permitirán llevar un control de los cambios realizados en una tabla. Para esto nos debemos de apoyar de una segunda tabla (Comúnmente una tabla log).

**Desventajas de los triggers**

1. Los triggers al ejecutarse de forma automática puede dificultar llevar un control sobre qué sentencias SQL fueron ejecutadas.
2. Los triggers incrementan la sobrecarga del servidor. Un mal uso de triggers puede tornarse en respuestas lentas por parte del servidor.

**Crear trigger**

DELIMITER %

CREATE TRIGGER after\_insert\_update\_quantity\_books

AFTER INSERT ON libros

FOR EACH ROW

BEGIN

UPDATE autores SET cantidad\_libros = cantidad\_libros + 1 WHERE autor\_id = NEW.autor\_id;

END;

%

DELIMITER ;

NEW hace referencia la nuevo registro insertado en la tabla de libros y que adicionalmente tiene el atributo de autor\_id

Luego agregamos un nuevo libro para poder evidenciar el trigger impactando la tabla de autores

INSERT INTO libros(autor\_id, titulo, fecha\_publicacion) VALUES(1, 'Area 81', '2011-07-01');

**Ahora ejecutando el evento DELETE**

DELIMITER %

CREATE TRIGGER after\_delete\_update\_quantity\_books

AFTER DELETE ON libros

FOR EACH ROW

BEGIN

UPDATE autores SET cantidad\_libros = cantidad\_libros - 1 WHERE autor\_id = OLD.autor\_id;

END;

%

DELIMITER ;

**Evento UPDATE**

DELIMITER %

CREATE TRIGGER after\_update\_actualizar\_libros

AFTER UPDATE ON libros

FOR EACH ROW

BEGIN

IF(NEW.autor\_id != OLD.autor\_id) THEN

UPDATE autores SET cantidad\_libros = cantidad\_libros + 1 WHERE autor\_id = NEW.autor\_id;

UPDATE autores SET cantidad\_libros = cantidad\_libros - 1 WHERE autor\_id = OLD.autor\_id;

END IF;

END;

%

DELIMITER ;

**Para listar los triggers**

SHOW TRIGGERS\G ;

**Para eliminar los triggers asociados a la vbase de datos**

DROP TRIGGER IF EXISTS <database>.<trigger>

DROP TRIGGER IF EXISTS libreria\_cf.after\_insert\_update\_quantity\_books;