

# Módulo 6: Sistemas de Gestión de Bases de datos Relacionales

Los sistemas de gestión de bases de datos actúan como intermediarios entre las aplicaciones y los datos almacenados, proporcionan un conjunto de herramientas y servicios para crear y mantener las bases de datos de manera eficiente y segura. Son paquetes de software que comúnmente suelen llamarse “motor” de base de datos. Existen diversos fabricantes de sistemas de gestión de bases de datos, y normalmente definimos a la base de datos por el nombre del sistema de gestión, así decimos una base MySQL o decimos una base Oracle, o decimos una base MsSQL cuando nos referimos a las bases de Microsoft. Todos tienen en común el lenguaje SQL.

## PARTE 1:

### Arquitectura de los (SGBDR) Sistemas de Gestión de Bases de Datos Relacionales

Enfocaremos el análisis de la arquitectura en dos aspectos importantes de estudio, el primero el modelo base de sistema de gestión, luego estudiaremos los componentes y funciones principales para la gestión.

a) **Modelo base:** Estos sistemas proporcionan muchas herramientas, tanto para que los diseñadores de software y los programadores puedan gestionar los datos almacenados con foco en la representación y procesamiento de los datos para obtener información, como para que los administradores de infraestructura, enfocados en la performance y la eficiencia gestionen los recursos dedicados garantizando la disponibilidad, resguardo y recuperación de datos.

Estos SGBDR se basan en un modelo teórico de tres capas propuesto en los años 70 por ANSI/SPARC (American National Standards Institute / Standards Planning and Requirements Committee).

Su objetivo principal es separar los distintos niveles de abstracción de una base de datos para garantizar independencia de datos y facilitar la gestión y uso de los sistemas de bases de datos en tres niveles.

➤ Nivel Externo, este nivel proporciona la vista de usuario, también es denominado “Vista” o “Vista de usuario” (Views), provee la forma en que los usuarios ven los datos, ofrece opciones de personalización de los datos, oculta la complejidad interna del sistema, implementa medidas de seguridad. Se implementa en forma de Views, donde se definen las columnas específicas, formatos de los datos y restricciones de visualización para los usuario y grupos.

➤ Nivel Conceptual, este nivel describe la lógica de la base de datos, también denominado “Nivel Lógico”, “Esquema Lógico” o “Vista Global”. En este nivel se implementa toda la lógica de la base de datos, las tablas, relaciones entre entidades, restricciones de datos y definición de las vistas. Se implementa en tablas, índices, constraints (restricciones), claves, claves foráneas, índices, triggers o disparadores, procedimientos almacenados, etc.

➤ Nivel Físico, este describe como se almacenan los físicamente los niveles anteriores en los dispositivos de almacenamiento. Define los diferentes archivos y estructuras de carpetas a utilizar en el almacenamiento, las formas de indexación, la distribución en el disco y la

optimización del rendimiento de entrada salida. Aquí se puede encontrar una gran variedad de implementaciones, que cada fabricante desarrolla y optimiza para su mercado objetivo, por ejemplo, motores usan una carpeta con permisos específicos, mientras que otros pueden tomar directamente una unidad de disco o filesystem en un formato específico y gestionar su propio sistema de ficheros.

### Independencia de Datos

Como se menciona anteriormente esta arquitectura garantiza la independencia de datos en dos aspectos: Lógico y Físico.

- Independencia Lógica: la separación permite que los cambios en el nivel conceptual no afectan a las aplicaciones en el nivel externo, por ejemplo, si se modifica la definición del modelo conceptual, incorporando tablas, o restricciones, acompañado de los ajustes necesarios de las vistas, el nivel externo se mantiene sin cambios y las aplicaciones no requieren ajustes adicionales.
- Independencia Física: Los cambios en el almacenamiento físico no afectan a esquema lógico ni a las aplicaciones. Por ejemplo, un sistema de gestión de bases de datos, puede agregar archivos para índices y datos, o modificar el tamaño de los archivos, reordenar los datos dentro de los ficheros para optimizar el espacio, a fin de distribuir carga y optimizar rendimiento, sin que ello altere la forma en que los niveles superiores funcionan.

### b) Componentes y funciones principales de los SGBDR:

Entre los Componentes se distinguen:

- I. Procesador de Consultas: Se ocupa del análisis, validación, optimización y ejecución de las consultas.
- II. Gestor de transacciones: Se ocupa de controlar la ejecución de transacciones de forma concurrente, mediante técnicas de bloqueos en diferentes niveles, detección y resolución de deadlocks (abrazos mortales) y de la recuperación ante fallos, para mantener la consistencia de los datos mediante técnicas de logs (registro de transacciones) y checkpoints (puntos de verificación). Este componente se desarrolla más adelante.
- III. Gestor de Almacenamiento: conocido también en inglés como "Storage Manager", se ocupa de dos aspectos desafiantes para el sistema: uno es el "Gestor de Archivos", que se ocupa de interactuar con el sistema operativo en todo lo relacionado al acceso de entrada, salida para almacenamiento persistente de los datos y el otro es el "Gestor de Buffers", los gestores administran un espacio propio de memoria RAM, para el procesamiento rápido de los datos, que suelen denominar memoria cache de la base, aquí se almacenan temporalmente tablas, índices, estadísticas y resultados de las consultas recientes con el propósito principal de optimizar su disponibilidad y sincronización con el almacenamiento permanente, minimizando las operaciones de E/S.
- IV. Diccionario de Datos: El diccionario de datos actúa como un repositorio central de metadatos. Una base de datos especial que contiene: definiciones, estructuras, restricciones, relaciones y permisos de todos los objetos de la base de datos, como tablas,

vistas, índices, procedimientos, usuarios, etc. Por ejemplo, en relación a las columnas de una tabla, tiene detalles del identificador o nombre de la columna, el tipo de datos, el tamaño, el formato, etc. pero no el dato en sí que se almacena en una tabla. Se utilizan dos sub lenguajes para la gestión del Diccionario de Datos, ellos son DDL (Data Definition Language) y DML (Data Manipulation Language).

Entre las funciones podemos agruparlas de la siguiente forma:

I.Gestión de Datos. Distinguimos funciones dedicadas al almacenamiento y organización de las diferentes estructuras de datos como tablas e índices que involucran: Particionamiento: dedicada a la distribución horizontal y vertical de datos, se trata de dividir tablas muy grandes en fracciones para mejorar el acceso y optimizar performance.

Acceso y Recuperación mediante la aplicación de diferentes métodos de acceso como secuencial, indexado o hash.

Cache y buffering: dedicadas a la minimización y optimización de operaciones de E/S, con técnicas basadas en estadísticas mantienen en memoria las páginas más accedidas, los índices más utilizados y otros elementos para la optimización de consultas.

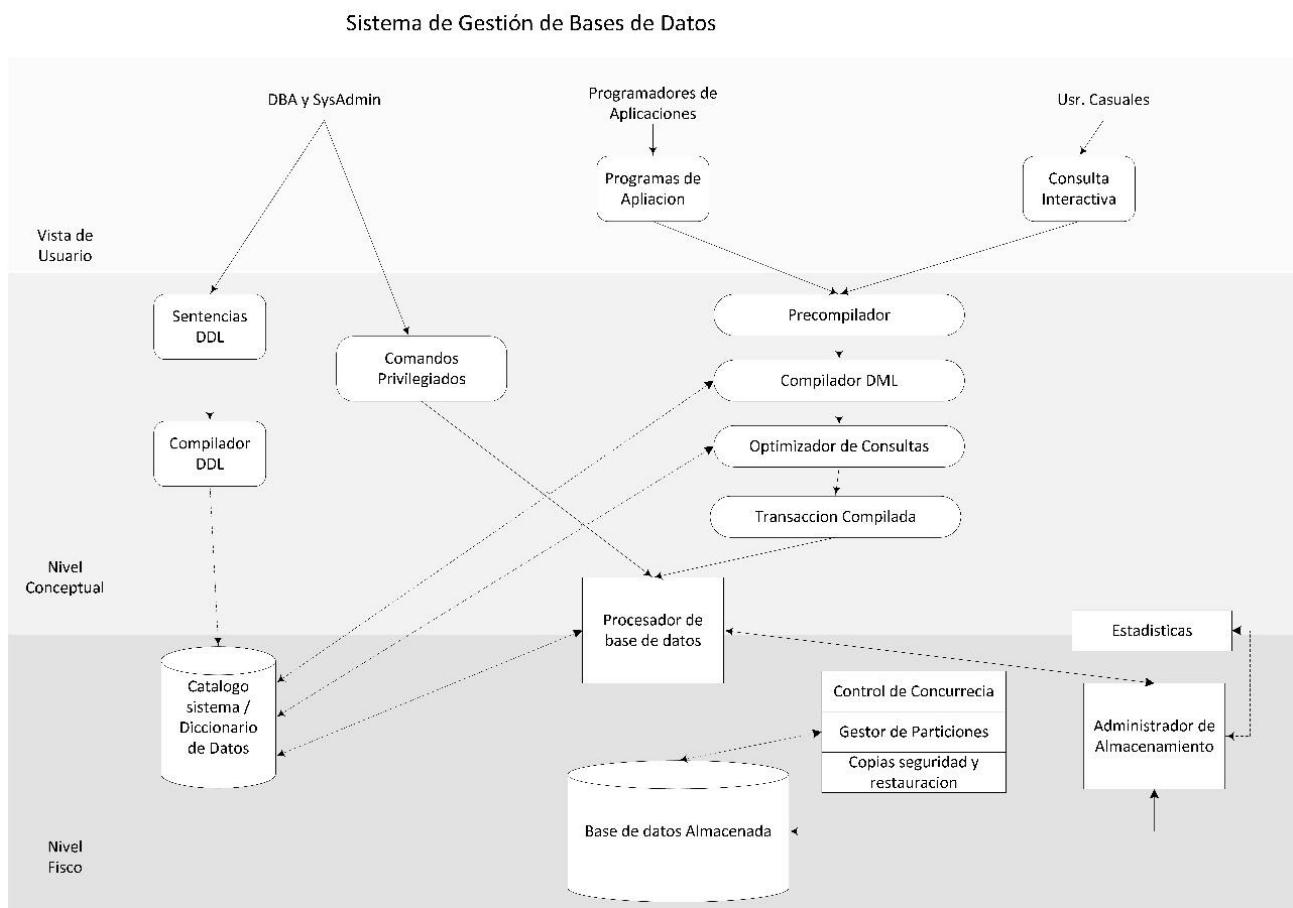
Paralelización: destinada a la ejecución paralela de consultas complejas.

II.Integridad y Consistencia, se ocupa de la verificación de las restricciones de Integridad de una tabla sobre claves primarias únicas y no nulas, de la integridad referencial sobre las claves foráneas, la integridad de tipos de datos o rangos y la integridad definida por el usuario (cuando se definen restricciones de negocio personalizadas). Estas funciones se aplican al momento de realizar inserciones, actualizaciones y borrado de datos, se suelen usar mecanismos como ejecución de triggers, procedimientos almacenados y acciones en cascada para mantener la consistencia de los datos.

III.Control de Acceso y Seguridad: se trata de la Autenticación y Autorización, gestión de usuarios, roles y permisos que estudiaremos más adelante.

IV.Administración y mantenimiento: los diferentes sistemas de gestión de bases de datos tienen funciones orientadas al monitoreo de recursos del sistema y gestión del rendimiento, que actúan mediante el análisis y optimización de consultas, generando planes de ejecución, basados en la lógica del modelo y estadísticas de uso o rendimiento. Proveen además funciones orientadas a roles específicos, como los DBA (Administradores de Bases de Datos) y SysAdmin (Administrador de Sistemas) para facilitar las tareas de mantenimiento que se ocupan de las tareas de respaldos y recuperación automatizados, reorganización y optimización de índices, actualización de estadísticas del optimizador y manejo de espacios de almacenamiento.

En la siguiente figura podemos observar los distintos componentes del modelo





## PARTE 2:

### DDL – Lenguaje de Definición de Datos:

El DDL tiene como función principal la modificación de la estructura de la base de datos y, por lo tanto, actualiza automáticamente el diccionario de datos. Es un subconjunto de sentencias SQL o denominado también un sub lenguaje que incluye una gran variedad de sentencias que podemos identificar o agrupar según se trate de crear, modificar o eliminar definiciones de diferentes elementos como esquemas, bases de datos, tablas, vistas, etc. Así podemos hallar una vista de sentencias tales como ALTER DATABASE, ALETER TABLE, ALTER VIEW, ALTER TABLESPACE, CREATE FUNCTION, DROP TABLE, RENAME TABLE, etc. Puede encontrar una referencia completa de las sentencias de MySQL para la definición de datos en el manual de referencia de Mysql : [MySQL :: MySQL 8.4 Reference Manual :: 15.1 Data Definition Statements](#)

Esto se realiza mediante instrucciones de SQL como: CREATE TABLE, ALTER TABLE y DROP TABLE

**CREATE TABLE** esta instrucción acompañada del nombre de la tabla a crear y los datos de las columnas permite agregar una tabla a la base de datos.

Ejemplo:

```
CREATE TABLE empleados (
    id INT PRIMARY KEY,
    nombre VARCHAR(100),
    salario DECIMAL(10,2)
);
```

**ALTER TABLE** esta sentencia permite que se modifiquen columnas de una tabla permitiendo agregar o quitar columnas o modificar tipos o tamaño de los datos previamente definidos.

Ejemplo:

```
ALTER TABLE empleados ADD fecha_ingreso DATE ;
```

**DROP TABLE**, permite eliminar una tabla de la base de datos.

Ejemplo:

```
DROP TABLE empleados ;
```

No menos importante, es poder ver la estructura de una tabla que necesitamos usar:

**SHOW CREATE TABLE** esta sentencia nos permite ver la estructura completa de la tabla con sus restricciones y chequeos.

```
SHOW CREATE TABLE empleados ;
```

**Veamos en detalle la sintaxis de CREATE TABLE**

```
CREATE TABLE nombre_tabla
( columna1 tipo_dato opciones,
  columna2 tipo_dato opciones,
  ...
  CONSTRAINT nombre_constraint tipo_constraint (columnas)
);
```



Analicemos cada componente:

**CREATE TABLE:** Sentencia principal para crear una tabla.

nombre\_tabla: Nombre que se le asigna a la tabla , no puede repetirse.

columna1, columna2, ... columnaN: Nombres de las columnas que formarán la tabla, no se repiten.

tipo\_dato: Tipo de dato de la columna puede ser INT, TINYINT, SMALLINT, BIGINT, VARCHAR, DATE, DECIMAL, etc.

opciones: Restricciones y propiedades adicionales para la columna (NOT NULL, DEFAULT, , PRIMARY KEY, etc.).

**CONSTRAINT:** Permite definir restricciones a nivel de tabla (claves primarias, foráneas, etc.).

nombre\_constraint: Nombre opcional para la restricción.

tipo\_constraint: Tipo de restricción (PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK).

(columnas): Columnas afectadas por la restricción.

Nota : Para ampliar detalles de los tipos de datos se recomienda visitar el manual de referencia en el siguiente enlace : <https://dev.mysql.com/doc/refman/8.4/en/data-types.html> y los detalles de almacenamiento en: <https://dev.mysql.com/doc/refman/8.4/en/storage-requirements.html>

Detalles de las opciones fundamentales:

NOT NULL: La columna no puede contener valores NULL, debe tener algún valor asignado.

DEFAULT valor: Define un valor por defecto para la columna si no se especifica al insertar datos.

AUTO\_INCREMENT (en MySQL): Asigna automáticamente un valor incremental a la columna (útil para claves primarias). En otras bases de datos (como PostgreSQL), se usa SERIAL.

UNIQUE: Asegura que todos los valores de la columna sean únicos.

CHECK (condición): Define una condición que deben cumplir los valores de la columna.

Ejemplo creación de la una tabla de clientes genérica:

```
CREATE TABLE clientes (
    cliente_id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(255) NOT NULL,
    apellido VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE,
    fecha_registro DATE DEFAULT CURRENT_DATE,
    telefono VARCHAR(20),
    direccion VARCHAR(255),
    ciudad VARCHAR(255),
    codigo_postal VARCHAR(10),
    pais VARCHAR(50) DEFAULT 'España',
    CHECK (LENGTH(codigo_postal) = 5) -- restricción CHECK
);
```

**Claves Primarias (PRIMARY KEY)** : En una tabla se pueden definir claves primarias formadas por una sola columna o por más de una columna, para ello las opciones de SQL son:

A nivel columna la sintaxis es:

```
columna tipo_dato PRIMARY KEY
```

A Nivel de Tabla (Constraint):



CONSTRAINT pk\_nombre\_tabla PRIMARY KEY (columna1, columna2, ...)

Recordemos que una clave primaria (PK) permite identificar de manera única cada fila en una tabla, una tabla solo puede tener una clave primaria y las columnas que forman la clave primaria no pueden contener valores NULL.

En el ejemplo anterior, `cliente\_id` ya es la clave primaria. Si quisieramos definir una clave primaria compuesta en una tabla productos genérica:

```
CREATE TABLE productos (
    producto_id INT,
    codigo_fabricante VARCHAR(50),
    nombre VARCHAR(255) NOT NULL,
    precio DECIMAL(10, 2),
    CONSTRAINT pk_productos PRIMARY KEY (producto_id, codigo_fabricante)
);
```

### Claves Foráneas (FOREIGN KEY)

Sintaxis a aplicar a nivel de tabla:

```
CONSTRAINT fk_nombre_restriccion FOREIGN KEY (columna_local) REFERENCES
nombre_tabla_referenciada (columna_referenciada)
    ON DELETE opción
    ON UPDATE opción
```

Recordemos que una clave foránea (FK) establece una relación entre dos tablas donde la clave foránea en la tabla actual (columna\_local) hace referencia a la clave primaria (o clave única) en otra tabla (columna\_referenciada).

ON DELETE opción: Define qué hacer cuando se elimina una fila en la tabla referenciada.

ON UPDATE opción: Define qué hacer cuando se actualiza una fila en la tabla referenciada.

Las opciones fundamentales para ON DELETE y ON UPDATE son :

CASCADE: Elimina/actualiza las filas correspondientes en la tabla actual.

SET NULL: Establece la columna de clave foránea a NULL en la tabla actual.

SET DEFAULT: Establece la columna de clave foránea al valor por defecto en la tabla actual.

RESTRICT (o 'NO ACTION'): Impide la eliminación/actualización si existen filas relacionadas en la tabla actual.

### Profundicemos :

ON DELETE es una cláusula que se define dentro de una restricción FOREIGN KEY. Esta cláusula especifica qué acción debe tomar el sistema de base de datos cuando se intenta eliminar una fila en la tabla "madre" (la tabla que contiene la clave primaria referenciada), y esa fila tiene filas relacionadas en la tabla "hija" (la tabla que contiene la clave foránea). En otras palabras, si eliminas un registro en la tabla donde la clave foránea "apunta", ¿qué hacemos con los registros relacionados en la tabla donde está la clave foránea?. Supongamos que tenemos definido lo siguiente:



```
CREATE TABLE clientes (
    cliente_id INT PRIMARY KEY,
    nombre VARCHAR(255)
); -- Tabla madre

CREATE TABLE pedidos (
    pedido_id INT PRIMARY KEY,
    cliente_id INT,
    FOREIGN KEY (cliente_id) REFERENCES clientes (cliente_id) ON
DELETE RESTRICT
); -- Tabla hija
```

**ON DELETE RESTRICT** (u **ON DELETE NO ACTION**): Impide la eliminación de la fila en la tabla madre si existen filas relacionadas en la tabla hija. Es decir, si hay registros en la tabla pedidos que apuntan a un cliente específico, no se puede borrar ese cliente.

Cuando usar: Cuando es crucial mantener la integridad referencial y evitar eliminaciones accidentales que podrían dejar datos huérfanos. Es el comportamiento por defecto en muchos SGBDs si no se especifica **ON DELETE**.

Si insertamos los datos

```
INSERT INTO clientes (cliente_id, nombre) VALUES (1, 'Juan Pérez');
INSERT INTO pedidos (pedido_id, cliente_id) VALUES (101, 1);
```

Luego intentamos eliminar el cliente con ID 1

```
DELETE FROM clientes WHERE cliente_id = 1;
```

Esto da error: No se puede eliminar el cliente porque tiene pedidos asociados.

En este caso, la instrucción **DELETE** fallará porque la restricción **ON DELETE RESTRICT** impide eliminar el cliente mientras existan pedidos asociados a él. Primero deberías eliminar los pedidos del cliente, y luego podrías eliminar al cliente.

**ON DELETE CASCADE**: Elimina automáticamente las filas relacionadas en la tabla hija cuando se elimina la fila en la tabla madre. Si se borra un cliente, se borrarán todos los pedidos relacionados.

Cuando usar: Cuando la eliminación de una fila en la tabla madre implica lógicamente la eliminación de las filas relacionadas en la tabla hija, y existe se quiere explícitamente este comportamiento. Debe usarse con cuidado porque puede producir eliminación de datos no deseada, debemos estar muy seguros en estos casos.

Supongamos que tenemos definido lo siguiente (ver modificación de **ON DELETE**):

```
CREATE TABLE clientes (
    cliente_id INT PRIMARY KEY,
    nombre VARCHAR(255)
);

CREATE TABLE pedidos (
    pedido_id INT PRIMARY KEY,
    cliente_id INT,
    FOREIGN KEY (cliente_id) REFERENCES clientes (cliente_id) ON
DELETE CASCADE
);
```



Insertamos un cliente y pedidos

```
INSERT INTO clientes (cliente_id, nombre) VALUES (1, 'María López');
INSERT INTO pedidos (pedido_id, cliente_id) VALUES (201, 1);
INSERT INTO pedidos (pedido_id, cliente_id) VALUES (202, 1);
```

Eliminamos el cliente con ID 1

```
DELETE FROM clientes WHERE cliente_id = 1; -- ¡Esto tendrá éxito!
```

Verificamos los pedidos

```
SELECT * FROM pedidos;
```

La consulta devuelve todas las columnas en NULL. No habrá ningún pedido. Los pedidos 201 y 202 también fueron eliminados automáticamente.

En este caso, la opción ON DELETE CASCADE se ejecuta y automáticamente se eliminan los pedidos asociados al cliente con ID 1.

**ON DELETE SET NULL:** Establece el valor de la columna de clave foránea a `NULL` en las filas relacionadas en la tabla hija cuando se elimina la fila en la tabla madre. Esto solo funciona si la columna de clave foránea permite valores `NULL` (no tiene la restricción `NOT NULL` en la tabla hija).

**Cuándo usar:** Cuando quieras mantener la existencia de las filas en la tabla hija, pero desvinculándolas de la fila eliminada en la tabla padre. Útil si la relación es opcional.

```
CREATE TABLE clientes (
    cliente_id INT PRIMARY KEY,
    nombre VARCHAR(255)
);

CREATE TABLE pedidos (
    pedido_id INT PRIMARY KEY,
    cliente_id INT NULL, -- La clave foránea permite valores NULL
    FOREIGN KEY (cliente_id) REFERENCES clientes (cliente_id) ON
DELETE SET NULL
);
```

Insertamos cliente y pedido

```
INSERT INTO clientes (cliente_id, nombre) VALUES (1, 'Carlos García');
INSERT INTO pedidos (pedido_id, cliente_id) VALUES (301, 1);
```

Eliminamos el cliente con ID 1

```
DELETE FROM clientes WHERE cliente_id = 1;
```

Esto no genera error y se ejecuta, al verificar los pedidos vemos el valor null en la columna cliente\_id

```
SELECT * FROM pedidos;
```

En este caso vemos el pedido con ID 301 en la tabla pedidos, pero la columna cliente\_id se establece a NULL, indicando que ya no está asociado a ningún cliente. Esto puede formar una inconsistencia referencial.

**ON DELETE SET DEFAULT:** Establece el valor de la columna de clave foránea al valor por defecto definido para esa columna en las filas relacionadas en la tabla hija cuando se elimina la fila en la tabla madre. Esto solo funciona si la columna de clave foránea tiene un valor DEFAULT definido.



Cuándo usar: Es similar a SET NULL, pero en lugar de establecer la columna a NULL, se establece a un valor por defecto específico.

```
CREATE TABLE clientes (
    cliente_id INT PRIMARY KEY,
    nombre VARCHAR(255)
);

CREATE TABLE pedidos (
    pedido_id INT PRIMARY KEY,
    cliente_id INT DEFAULT 0, -- La clave foránea tiene un valor por
defecto
    FOREIGN KEY (cliente_id) REFERENCES clientes (cliente_id) ON
DELETE SET DEFAULT
);
```

Insertamos algunos datos

```
INSERT INTO clientes (cliente_id, nombre) VALUES (1, 'Ana Rodríguez');
INSERT INTO pedidos (pedido_id, cliente_id) VALUES (401, 1);
```

Eliminamos el cliente con ID 1

```
DELETE FROM clientes WHERE cliente_id = 1;
```

No se genera error y se borran el cliente .

Verificamos los pedidos

```
SELECT * FROM pedidos;
```

El pedido 401 existirá, pero con el valor de cliente\_id establecido a 0 (el valor por defecto).

En este caso, el pedido con ID 401 permanece en la tabla pedidos, pero la columna cliente\_id se establece a 0 (el valor por defecto), indicando que ya no está asociado a ningún cliente válido.

Opciones de ON UPDATE: Las opciones de ON UPDATE son similares a las analizadas anteriormente.

**ON UPDATE CASCADE:** Si se modifica la clave primaria en la tabla referenciada, la misma modificación se propaga a la tabla que hace referencia a esa clave foránea. Por ejemplo, si se cambia el ID de un cliente en la tabla de clientes, se actualizarán todos los pedidos relacionados con ese cliente en la tabla de pedidos.

**ON UPDATE RESTRICT:** Impide la modificación de la clave primaria si existen referencias a ella en la tabla hija. Si se intenta cambiar el ID de un cliente que tiene pedidos, la operación de actualización se rechazará. Es la opción más restrictiva.

**ON UPDATE NO ACTION:** Similar a RESTRICT, pero la comprobación de la restricción puede diferir hasta el momento de la confirmación de la transacción. En algunos motores de base de datos, como MySQL, NO ACTION y RESTRICT pueden ser equivalentes.

**ON UPDATE SET NULL:** Si se modifica la clave primaria en la tabla madre, las referencias a esa clave en la tabla hija se establecen a NULL. Asume que la columna de la clave foránea puede admitir valores NULL.

**ON UPDATE SET DEFAULT:** Si se modifica la clave primaria en la tabla madre, las referencias a esa clave en la tabla hija se establecen al valor por defecto de la columna de clave foránea. Esta opción no es soportada por todos los motores de base de datos.

Ejemplo en la tabla de pedidos relacionada con la tabla clientes.

```
CREATE TABLE pedidos (
    pedido_id INT AUTO_INCREMENT PRIMARY KEY,
    cliente_id INT NOT NULL,
    fecha_pedido DATE NOT NULL,
    importe DECIMAL(10, 2),
    CONSTRAINT fk_pedidos_clientes FOREIGN KEY (cliente_id) REFERENCES
    clientes (cliente_id)
    ON DELETE RESTRICT -- No permitir eliminar un cliente si tiene pedidos.
    ON UPDATE CASCADE   -- Actualizar el cliente_id en pedidos si se cambia en la tabla clientes.
);
```



### Cuándo usar RESTRICT o CASCADE:

Usar RESTRICT cuando:

- La integridad referencial es crítica y se quiere evitar la eliminación accidental de datos relacionados.
- La eliminación de una fila en la tabla madre no implica lógicamente la eliminación de las filas relacionadas en la tabla hija.
- Es necesario tener un control estricto sobre las eliminaciones y asegurarte de que todas las dependencias se resuelvan manualmente.

Ejemplo:

En una aplicación bancaria, no querrías permitir la eliminación de un cliente si tiene cuentas bancarias activas. Usarías `ON DELETE RESTRICT` en la relación entre `clientes` y `cuentas`.

Usar CASCADE con precaución cuando:

- Existe certeza de que la eliminación de una fila en la tabla madre implica lógicamente la eliminación de las filas relacionadas en la tabla hija.
- Se busca simplificar el proceso de eliminación y automatizar la limpieza de datos relacionados.

Solamente si se analizaron completamente las implicaciones de eliminar datos en cascada y se aceptaron los riesgos asociados.

Ejemplo:

En un sistema específico de gestión de tareas, si se elimina un proyecto, se desea eliminar automáticamente todas las tareas asociadas a ese proyecto. Se usa ON DELETE CASCADE en la relación entre proyectos y tareas.

### Buenas Prácticas para desarrolladores:

- Planificación: Analizar cuidadosamente sobre las relaciones entre las tablas y cómo deben comportarse las eliminaciones antes de definir las restricciones FOREIGN KEY.
- Documentación: Documentar las restricciones FOREIGN KEY y las opciones ON DELETE para que otros desarrolladores entiendan el comportamiento de la base de datos.
- Pruebas: Realizar testeos exhaustivos para asegurar de que las restricciones FOREIGN KEY y las opciones ON DELETE funcionan como se espera.

En resumen, la opción ON DELETE en una restricción FOREIGN KEY te permite controlar cómo se manejan las eliminaciones de datos relacionados en tu base de datos. RESTRICT proporciona la mayor seguridad al prevenir eliminaciones que romperían la integridad referencial, mientras que CASCADE permite la eliminación automática de datos relacionados, pero debe usarse con precaución. Las opciones SET NULL y SET DEFAULT ofrecen alternativas para mantener las filas relacionadas, pero desvinculándolas de la fila eliminada, lo que puede afectar la integridad referencial.

### Condición CHECK

En DDL la condición CHECK define una restricción de integridad que se puede aplicar a columnas o tablas para asegurar que los datos insertados cumplan ciertos criterios(validación). Si se intenta insertar o actualizar un valor que no cumpla con la condición CHECK, se producirá un error y la operación será rechazada. Por



ejemplo, consideremos que en la tabla producto el código debe cumplir con un formato de la siguiente forma: L-12345-X

Es decir que:

Comienza con una letra mayúscula inicial, seguida de un guion medio (-); luego cinco dígitos numéricos, otro guion medio (-) y finaliza con una letra mayúscula final distinta de la primera.

En MySQL, las restricciones CHECK existen desde la versión 8.0.16, pero con una limitación importante dado que no soportan expresiones regulares directamente dentro de la cláusula CHECK como lo hacen otros SGDB. Por eso, para validar un formato como L-12345-X, debemos utilizar funciones de cadena (SUBSTRING() y LIKE) en combinación.

Ejemplo:

```
CREATE TABLE productos (
    codigo_producto VARCHAR(9) PRIMARY KEY,
    CONSTRAINT ck_codigo_formato CHECK (
        LENGTH(codigo_producto) = 9 AND          -- Longitud exacta
        SUBSTRING(codigo_producto, 2, 1) = '-' AND  -- Guiones en posición 2 y 8
        SUBSTRING(codigo_producto, 8, 1) = '-' AND
        SUBSTRING(codigo_producto, 1, 1) BETWEEN 'A' AND 'Z' AND -- Primer carácter entre A y Z
        SUBSTRING(codigo_producto, 9, 1) BETWEEN 'A' AND 'Z' AND -- Última letra entre A y Z
        SUBSTRING(codigo_producto, 3, 5) REGEXP '^[0-9]{5}$' -- Los caracteres del medio deben ser todos dígitos. Si la versión no soporta REGEXP usar: CAST(SUBSTRING(codigo_producto, 3, 5) AS UNSIGNED) BETWEEN 10000 AND 99999 ),
        CONSTRAINT ck_codigo_distinto CHECK (
            SUBSTRING(codigo_producto, 1, 1) <> SUBSTRING(codigo_producto, 9, 1)
    )
);
```

Probamos insertando los siguientes valores:

```
INSERT INTO productos (codigo_producto) VALUES ('A-12345-Z');
INSERT INTO productos (codigo_producto) VALUES ('B-54321-B');
INSERT INTO productos (codigo_producto) VALUES ('1-12345-Z');
INSERT INTO productos (codigo_producto) VALUES ('A12345Z');
```

Veremos que solo el primero es correcto y que el resto nos dan mensajes de error.

## INDICES

Crear tablas y guardar datos es solo una parte de la historia. La otra, igual de importante, es asegurarse de que la aplicación sea rápida y eficiente. Aquí es donde los índices entran en juego, y el DDL (Data Definition Language) es la herramienta para crearlos y administrarlos.

### ¿Qué son los Índices y Por Qué Son Importantes?

Consideremos que tenemos una biblioteca gigantesca sin ningún tipo de organización. Si alguien pide un libro específico, tenemos que revisar libro por libro hasta encontrarlo, lo cual resultaría muy lento.

Sin embargo, con un catálogo de tarjetas que nos dé información de la ubicación de los libros, al buscar un libro por título, autor o tema, (un índice) tendríamos la ubicación.

En las bases de datos, los índices funcionan de manera muy similar. Son estructuras especiales que se utilizan para localizar rápidamente las filas de una tabla. Sin índices, el SGBD tendría que escanear toda la tabla (lo que se conoce como "full table scan") para encontrar los datos buscados, lo cual puede ser extremadamente lento en tablas grandes.



**En resumen, los índices sirven para:**

Acelerar las consultas SELECT: Especialmente aquellas que usan WHERE, JOIN, ORDER BY y GROUP BY.

Mejorar la eficiencia de UPDATE y DELETE: Aunque puede sonar contradictorio, un UPDATE o DELETE primero tiene que encontrar la fila que va a modificar o eliminar, y los índices ayudan en esa búsqueda.

Garantizar la unicidad de los datos: Los índices únicos aseguran que no haya filas duplicadas en una o más columnas.

**Pero debemos también considerar que los índices:**

Ocupan espacio en disco: Los índices son estructuras de datos que se almacenan en la base de datos.

Ralentizan INSERT, UPDATE y DELETE: Cada vez que modificas los datos en una tabla indexada, también hay que actualizar el índice. Es como en la biblioteca del ejemplo anterior, cada vez que se agrega o mueve un libro, se debe actualizar el catálogo, caso contrario progresivamente pierde consistencia la información y su valor de uso.

Entonces la cuestión está en encontrar el equilibrio, es decir, crear solo índices donde realmente se los necesita para las consultas más frecuentes y críticas. Para evaluar la situación de una tabla en una consulta podemos usar EXPLAIN antes del SELECT, lo que nos permite ver el plan de ejecución que resulta del optimizador de consultas.

En DDL las sentencias principales son **CREATE INDEX, ALTER TABLE y DROP INDEX**.

**CREATE INDEX:** Crea índices adicionales, es el comando más directo para agregar un índice a una tabla existente.

Sintaxis básica:

```
CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX nombre_indexe  
ON nombre_tabla (columna1 [ASC | DESC], columna2 [ASC | DESC], ...);
```

**Analicemos cada componente:**

**CREATE INDEX:** La es la sentencia para crear un índice.

**[UNIQUE | FULLTEXT | SPATIAL]:** Son modificadores opcionales que definen el tipo de índice:

**UNIQUE:** Garantiza que todos los valores en la columna (o combinación de columnas) indexadas sean únicos. Si se intenta insertar un valor duplicado, genera un error. Esto es muy útil para evitar datos repetidos y para búsquedas rápidas.

**FULLTEXT:** Se utiliza para indexar columnas que contienen texto largo (como TEXT o VARCHAR) y realizar búsquedas de texto completas (MATCH...AGAINST). Esto es para búsquedas tipo "Google" dentro de tus datos.

**SPATIAL:** Para columnas que almacenan datos geográficos (coordenadas, formas, etc.) y realizar consultas espaciales. Es útil si se trabaja con mapas o ubicaciones.

**nombre\_indexe:** El nombre que se le da al índice. Es una buena práctica usar un nombre descriptivo, como idx\_tabla\_columna.

**ON nombre\_tabla:** Especifica la tabla donde se creará el índice.

**(columna1 [ASC | DESC], columna2 [ASC | DESC], ...):** Las columnas (o combinación de columnas) sobre las que se creará el índice. Se puede especificar si el orden es ascendente (ASC) o descendente (DESC).



Ejemplos:

- Índice simple para búsquedas por email

Supongamos que tenemos una tabla usuarios y a menudo buscamos usuarios por su correo electrónico, antes de crear el índice, debemos ver el plan de ejecución de una consulta que usa el email, para verificar si ya es eficiente o si necesita un índice.

```
EXPLAIN SELECT * FROM usuarios WHERE email = 'juan.perez@example.com';
```

Si es necesario creamos el índice en la columna 'email'

```
CREATE INDEX idx_usuarios_email ON usuarios (email);
```

Luego de crear el índice, volvemos a ejecutar EXPLAIN para ver la diferencia.

```
EXPLAIN SELECT * FROM usuarios WHERE email = 'juan.perez@example.com';
```

- Índice único para un username

Para asegurarnos de que cada usuario tenga un nombre de usuario único:

```
CREATE UNIQUE INDEX idx_usuarios_username ON usuarios (username);
```

- Índice compuesto (múltiples columnas)

Si se busca productos por categoría y precio:

```
CREATE INDEX idx_productos_categoria_precio ON productos (categoria, precio);
```

Un índice compuesto es útil cuando las consultas filtran u ordenan por las columnas en el mismo orden en que fueron definidas en el índice. Aquí también es importante utilizar EXPLAIN para ver diferentes alternativas.

```
SELECT * FROM productos WHERE categoria = 'Electrónica' AND precio < 500;
```

Al colocar las condiciones directamente en el orden declarado en el índice se asegura su utilización efectiva.

```
SELECT * FROM productos WHERE categoria = 'Electrónica';
```

Esto también utilizará el índice, pero solo la primera parte.

En cambio:

```
SELECT * FROM productos WHERE precio < 500;
```

Probablemente NO utilizará el índice porque no puede saltarse la primera columna del índice compuesto. Podemos ver esto como una guía de teléfonos: se puede buscar por apellido y luego por nombre, pero no solo por nombre, dado que los nombres están ordenados dentro de cada apellido, es necesario recorrer toda la guía para obtener todos los nombres posibles y descartar duplicados.

**ALTER TABLE:** Agregando Índices como Restricciones (Constraints)

Aunque CREATE INDEX es útil, a menudo los índices se crean como parte de la definición de la tabla o se añaden a través de ALTER TABLE, especialmente cuando se definen claves primarias o foráneas, o restricciones de unicidad.

Sintaxis general para agregar un índice modificando la tabla:

```
ALTER TABLE nombre_tabla ADD INDEX nombre_indice (columna1, ...);
```

Para agregar un índice de valores únicos:

```
ALTER TABLE nombre_tabla ADD UNIQUE INDEX nombre_indice (columna1, ...);
```

Para definir PRIMARY KEY (que es un índice UNIQUE por defecto):

```
ALTER TABLE nombre_tabla ADD PRIMARY KEY (columna1, ...);
```

O directamente como vimos en la creación de la tabla:

```
CREATE TABLE mi_tabla (
    id INT PRIMARY KEY,
    nombre VARCHAR(255)
);
```

Al definir FOREIGN KEY (automáticamente crea un índice en la columna de la clave foránea):

```
ALTER TABLE nombre_tabla
ADD CONSTRAINT fk_nombre_clave_foranea
FOREIGN KEY (columna_local) REFERENCES tabla_referenciada
(columna_referenciada);
```

Ejemplos:

Añadir un índice simple a la tabla 'pedidos' en la columna 'fecha\_pedido'

```
ALTER TABLE pedidos ADD INDEX idx_pedidos_fecha_pedido (fecha_pedido);
```

Añadir un índice único a la combinación de 'dni' y 'pais' para una tabla 'clientes'

```
ALTER TABLE clientes ADD UNIQUE INDEX idx_clientes_dni_pais (dni, pais);
```

¿Cuál usar, CREATE INDEX o ALTER TABLE ADD INDEX?

En la práctica, para agregar índices a columnas existentes, ambos son funcionalmente muy similares. ALTER TABLE es más general para modificar la estructura de la tabla, mientras que CREATE INDEX es específicamente para índices. Muchos desarrolladores usan ALTER TABLE por consistencia al modificar tablas.

**DROP INDEX:** Eliminando Índices



Si un índice ya no es necesario, por ejemplo, porque la consulta que lo usaba ya no se ejecuta con frecuencia, o porque se reemplazó con uno mejor, es bueno eliminarlo para liberar espacio y reducir la sobrecarga en las operaciones INSERT/UPDATE/DELETE.

Sintaxis:

```
DROP INDEX nombre_index ON nombre_tabla;
```

Ejemplo:

Eliminar el índice que creamos en la tabla 'usuarios' para 'email'

```
DROP INDEX idx_usuarios_email ON usuarios;
```

No se puede eliminar el índice de una PRIMARY KEY directamente con DROP INDEX, se debe usar ALTER TABLE DROP PRIMARY KEY.

Ejemplo:

Eliminar la clave primaria de una tabla, cuidado con el uso de esta sentencia, generalmente requiere un análisis profundo de la integridad y consistencia de la base de datos, será parte de una restructuración de claves primarias, o de una modificación de entidades.

```
ALTER TABLE clientes DROP PRIMARY KEY;
```

Verificación y visualización de índices:

La sentencia **SHOW INDEX FROM nombretabla** nos permite ver la definición de los índices de una tabla. Para poder optimizar nuestras consultas. Si no conocemos los índices disponibles difícilmente podamos crear consultas óptimas.

```
SHOW INDEX FROM clientes;
```

### Buenas Prácticas en relación a los índices

- Consultar los índices disponibles en las tablas para generar las consultas aprovechando los existentes.
- Analizar las consultas más lentas: Usar EXPLAIN para ver cómo MySQL está ejecutando tus consultas. Si ves "Using filesort", "Using temporary", o un type de ALL (full table scan), es una señal de que necesitas un índice.

```
EXPLAIN SELECT * FROM productos WHERE precio > 1000 ORDER BY fecha_lanzamiento DESC;
```

La salida de EXPLAIN tendrá pistas sobre dónde enfocar los esfuerzos de indexación.

Indexar las columnas en tus cláusulas WHERE, JOIN, ORDER BY y GROUP BY, son las principales candidatas a ser indexadas.

- Considera índices compuestos: Si tus consultas a menudo filtran u ordenan por varias columnas juntas, un índice compuesto puede ser muy beneficioso. Recuerda el "orden de las columnas" en el índice compuesto.

- No indexar en exceso: Demasiados índices pueden ser contraproducentes. Ralentizan las operaciones de escritura y consumen mucho espacio. Sé selectivo.
- Evitar indexar columnas con muy pocos valores únicos: Si una columna solo tiene dos valores (por ejemplo, activo SI/NO), un índice en esa columna probablemente no te dará mucho beneficio, ya que se tendría que escanear la mitad de la tabla.
- Tipos de datos correctos: Usar el tipo de dato más pequeño y apropiado para tus columnas también contribuye a un mejor rendimiento del índice y de la tabla en general. Por ejemplo, INT en lugar de BIGINT si tus números no serán tan grandes.
- Mantenimiento de la base de datos: A medida que los datos cambian, los índices pueden fragmentarse. Los SGBD generalmente manejan esto bien, pero en algunos casos (tablas con muchas eliminaciones o actualizaciones), puedes considerar OPTIMIZE TABLE para reorganizar el espacio de disco y mejorar el rendimiento.

### Conclusión

Entender y aplicar los principios de indexación es una de las habilidades más valiosas de un programador para optimizar el rendimiento de las aplicaciones. Comenzando de apoco, experimentando con EXPLAIN, y luego probando variantes de las consultas se puede desarrollar la habilidad para saber cuándo y dónde crear índices.



### PARTE 3:

#### DML – Lenguaje de Manipulación de Datos:

El DML (Data Manipulation Language) tiene como función principal la modificación de los datos en las estructuras definidas, no modifica el diccionario de datos, pero puede generar estadísticas y esta relacionado con las funciones de partición control de integridad y consistencia. Es un subconjunto de sentencias SQL o destinado a la selección, modificación y eliminación de datos. Podemos encontrar una referencia completa de las sentencias de MySQL para DML en el manual de referencia : [MySQL 8.4 Reference Manual/SQL Statements/Data Manipulation Statements](#)

Así podemos encontrar sentencias que nos permitan insertar datos en una fila, actualizar una columna o eliminar una fila de una tabla que ya se han visto en el módulo anterior.

A modo de identificación las sentencias comunes en este sub lenguaje son SELECT, INSERT, UPDATE, DELETE y REPLACE.

#### Ejemplos de sentencias DML:

-Para inserta el registro en la tabla empleados con los valores detallados.

```
INSERT INTO empleados (id, nombre, salario, departamento)
VALUES (1, 'Laura García', 4500.00, 'Ventas');
```

-Para copiar empleados del departamento de ventas en la tabla historial\_empleados con la fecha de salida actual. Esto combina INSERT en una tabla con un SELECT de otra.

```
INSERT INTO historial_empleados (id, nombre, salario, fecha_salida)
SELECT id, nombre, salario, CURDATE() FROM empleados WHERE departamento
= 'Ventas';
```

-Actualización o modificación: el ejemplo incrementa 10% el salario de los empleados del departamento Ventas

```
UPDATE empleados
    SET salario = salario * 1.10
    WHERE departamento = 'Ventas';
```

-Borrado: el ejemplo siguiente elimina el o los registros con ID = 3 de la tabla empleados

```
DELETE FROM empleados WHERE id = 3;
```

-Reemplazo, en el siguiente ejemplo si existe un empleado con ID 1, lo reemplaza con los nuevos datos; si no, lo inserta.

```
REPLACE INTO empleados (id, nombre, salario, departamento)
VALUES (1, 'Laura García', 4800.00, 'Marketing');
```

En general las sentencias de DML serán generalmente utilizadas por los programadores para la gestión de datos desde las aplicaciones, en tanto que las de DDL son más comunes en los roles de diseño y DBA, aunque los programadores pueden incluirlas en scripts de instalación o incorporación de nuevas funciones en sistemas

existentes. También es importante considerar que algunas tecnologías de desarrollo de aplicaciones incorporan herramientas para el modelado de datos y ejecutan este tipo de instrucciones, sobre distintos SGBDR, sin que el desarrollador se preocupe, en primera instancia, de las sentencias particulares a ejecutar, no por ello debería ignorarlas o desconocer su uso, debido a que en ocasiones las restricciones complejas provocan errores en la modelización automatizada.

Existen también funciones de agrupación para las estructuras de selección que permiten el uso de funciones de resumen de datos.

El siguiente ejemplo muestra el salario promedio de los empleados por departamento.

```
SELECT departamento, AVG(salario) AS salario_promedio  
FROM empleados  
GROUP BY departamento;
```

Para obtener la suma de salarios por departamento se reemplaza AVG por SUM. Otros resultados deseados podrían ser el salario mayor o menor utilizando MAX y MIN respectivamente.

Finalmente se pueden agregar funciones de ordenamiento de los datos combinadas o no con las de agrupación

```
SELECT departamento, AVG(salario) AS salario_promedio FROM empleados  
GROUP BY departamento  
ORDER BY departamento;
```

```
SELECT departamento, SUM(salario) AS salario_promedio FROM empleados  
GROUP BY departamento  
ORDER BY SUM(salario), departamento;
```

#### PARTE 4:

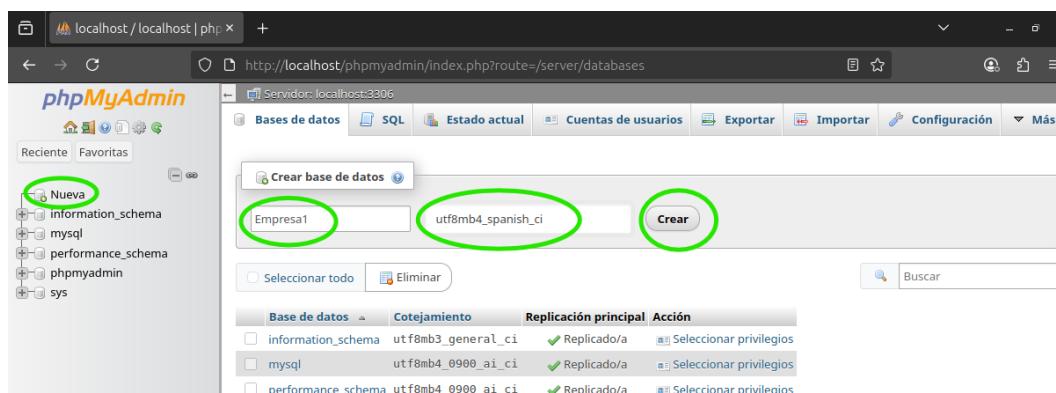
##### Gestión mediante phpMyAdmin

La mayoría de los SGBDR poseen una interface gráfica de gestión que permite la definición y manipulación de datos de manera intuitiva, que evalúa las restricciones y permite la realización de la gestión en un entorno amigable. En este caso utilizaremos la aplicación phpmyAdmin (PMA) que como Mysql Workbench permite gestionar las bases de datos, y es parte del paquete XAMPP.

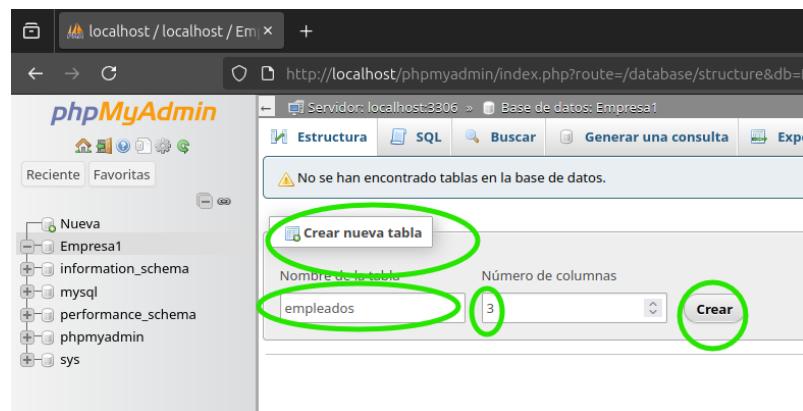
En entornos de servidores en la nube frecuentemente no contamos con una interface gráfica (o no deseamos usar recursos para instalarla), que nos permita usar una aplicación como workbench en modo local, esta aplicación PMA, proporciona una interfaz WEB, lo que nos permite gestionar desde un navegador la base de datos sin necesidad de levantar el modo grafico del servidor, ni tampoco instalar una aplicación cliente. Es útil en entornos cuando tenemos servidores, sin interfaz gráfica, ya que nos permite la gestión de la base en un entorno amigable, sin necesidad de usar recursos del servidor para el modo gráfico.

En primer término, debemos instalar y configurar el acceso al sistema de gestión de base de datos, una vez que competemos el login encontramos la pantalla principal de PMA, donde comenzaremos creando una nueva base de datos.

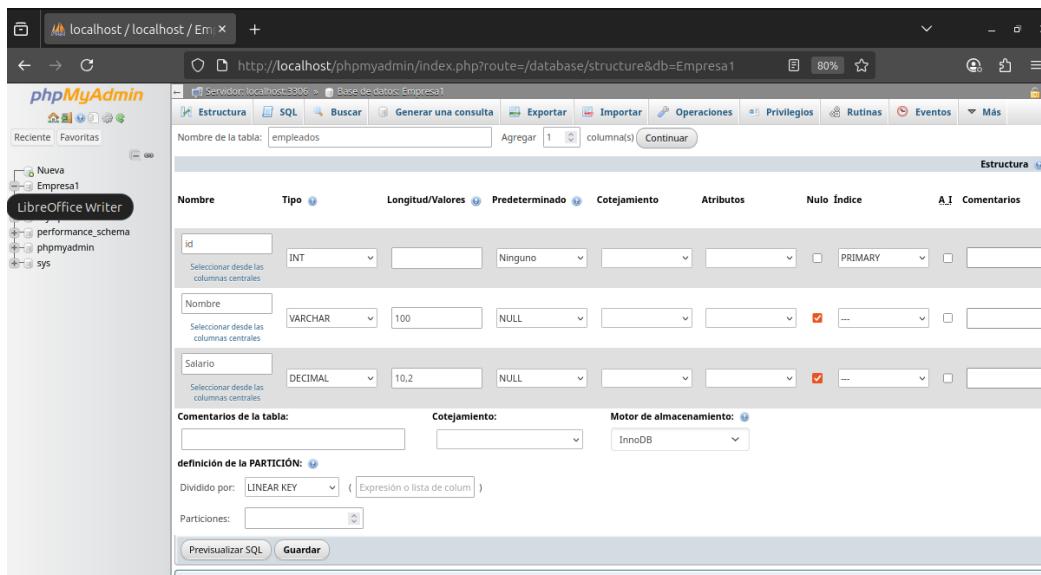
Nota: La aplicación PhpMyAdmin puede instalarse de por separado, aunque es recomendable la instalación de XAMPP que está disponible para Linux Windows y MAC.



Una vez creada la base de datos la aplicación nos propone crear una tabla, procedemos a definir la tabla empleados con 3 columnas:

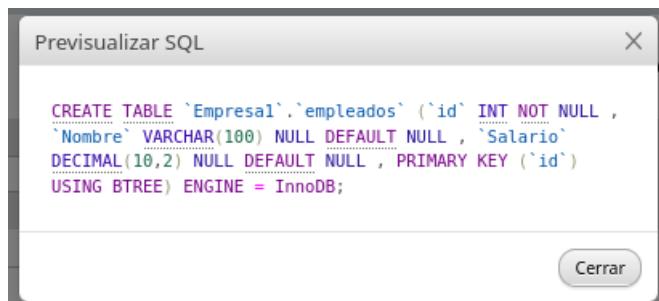


Y definimos los detalles de cada columna.

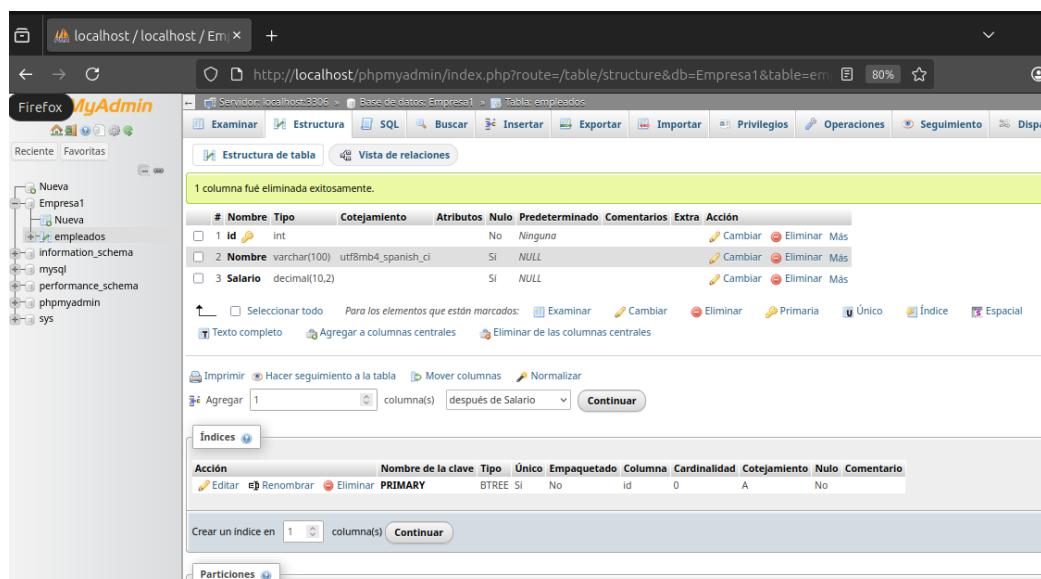


The screenshot shows the 'Estructura' (Structure) tab in phpMyAdmin for the 'empleados' table. The table has three columns: 'id' (INT, Primary Key), 'Nombre' (VARCHAR(100)), and 'Salario' (DECIMAL(10,2)). The 'Motor de almacenamiento' (Storage Engine) is set to InnoDB. A 'Previsualizar SQL' (Preview SQL) button is visible at the bottom.

Nos ofrece Pre visualizar SQL donde podemos observar la sentencia DDL a ejecutar.



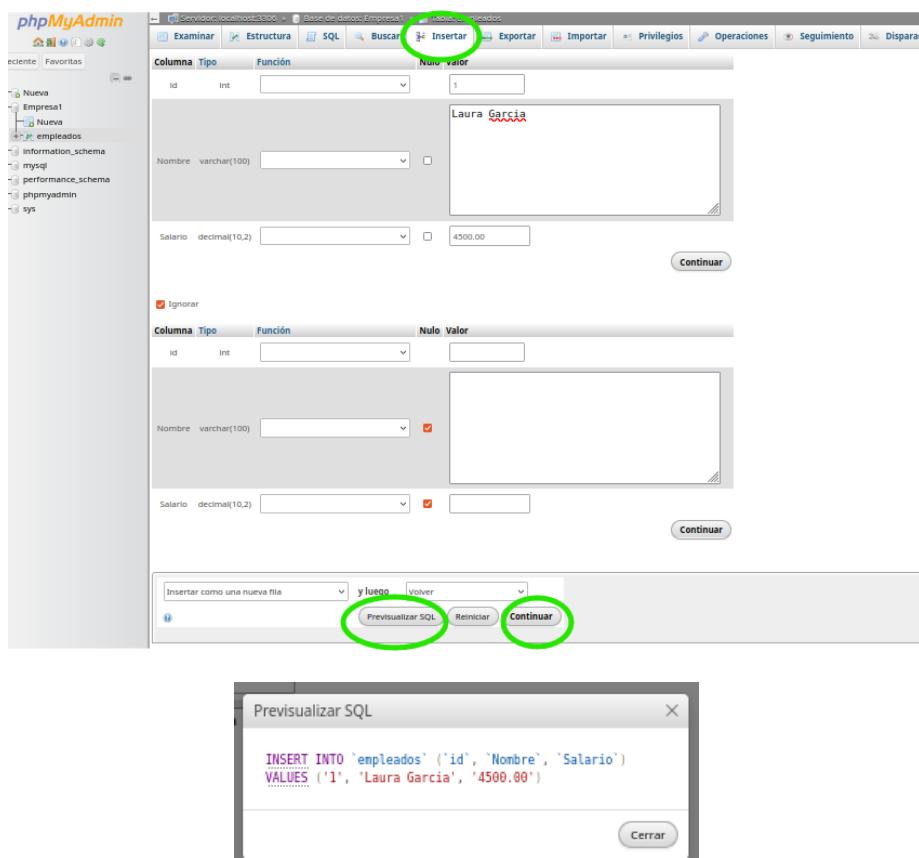
Damos Guardar, se crea la tabla, nos muestra su estructura y nos ofrece la posibilidad de cambiar las definiciones realizadas:



The screenshot shows the 'Estructura de tabla' (Table Structure) page for the 'empleados' table. It displays the table structure with three columns: 'id', 'Nombre', and 'Salario'. The 'id' column is defined as an INT type with a PRIMARY KEY constraint. The 'Nombre' column is a VARCHAR(100) type. The 'Salario' column is a DECIMAL(10,2) type. There is also a section for indices, showing a PRIMARY index on the 'id' column.

Una vez que habiendo usado DDL se cuenta con la definición de datos completa. Podemos ejecutar sentencias de DML para trabajar con los datos.

La interface de PMA nos permite manipular los datos manualmente en la tabla escribiendo los valores en cada columna seleccionando la solapa Insertar podemos agregar datos manualmente y visualizar la sentencia SQL:



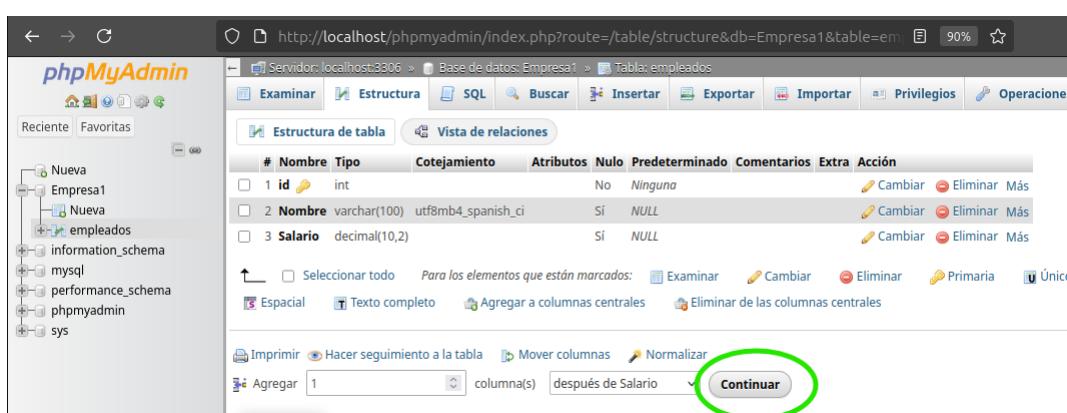
The screenshot shows the phpMyAdmin interface for the 'empleados' table. In the 'Insertar' tab, the 'Nombre' field contains 'Laura Garcia'. At the bottom, the 'Continuar' button and the 'Previsualizar SQL' button are highlighted with green circles.

**Previsualizar SQL**

```
INSERT INTO `empleados` (`id`, `Nombre`, `Salario`)
VALUES ('1', 'Laura Garcia', '4500.00')
```

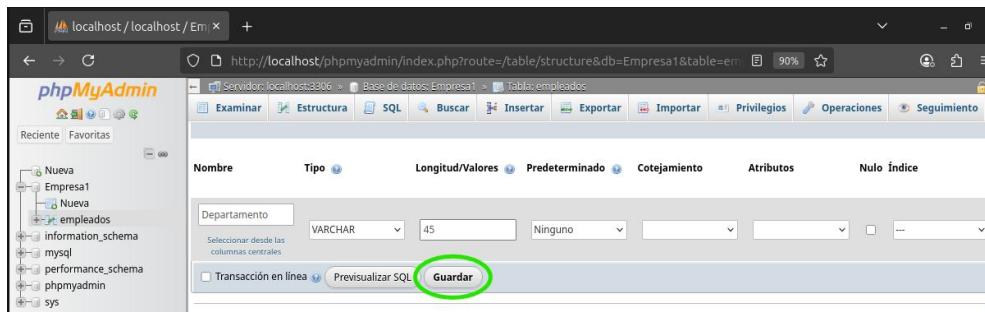
Al confirmar realiza la actualización de los datos en la tabla.

A continuación, agregamos a la tabla la columna departamento, vamos a la solapa estructura en la sección agregar seleccionamos continuar:

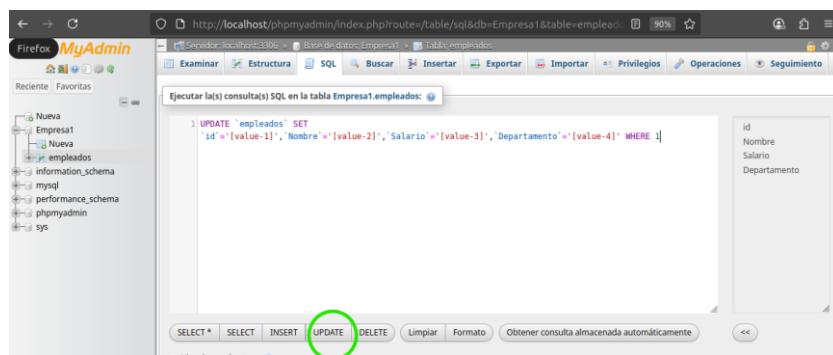


The screenshot shows the 'Estructura de tabla' tab for the 'empleados' table. A new column 'departamento' is being added after 'Salario'. The 'Continuar' button at the bottom is highlighted with a green circle.

Definimos los detalles de la columna y damos guardar



En la solapa SQL encontraremos varias opciones para realizar operaciones de DML habituales como SELECT, INSERT, UPDATE y DELETE, que nos asisten en la construcción de las sentencias.



Aquí podemos aplicar todas las sentencias estudiadas anteriormente, ver los EXPLAIN de un SELECT o ver la estructura de una tabla y los índices.