

SQL práctico para MySQL y MariaDB

SQL práctico para MySQL y MariaDB

**Guía completa con
ejemplos prácticos y
ejercicios resueltos**

Autor: Jose Vicente Carratala

"Este libro está dedicado a todos los que aman aprender."

Table of Contents

- 1. Prólogo
- 2. Cómo instalar un servidor local
 - 2.1. Instalación de un entorno de desarrollo local
 - 2.1.1. Introducción
 - 2.1.2. Instalación en Windows
 - 2.1.3. Instalación en macOS
 - 2.1.4. Instalación en Linux (Debian/Ubuntu)
- 3. Fundamentos de SQL
 - 3.1. DDL
 - 3.2. DML
 - 3.3. DQL
 - 3.4. DCL
 - 3.5. Operaciones
 - 3.6. Clausulas
 - 3.7. Funciones
 - 3.7.1. Agregado
 - 3.7.2. Matemáticas
 - 3.7.3. Cadenas
 - 3.7.4. Fechas
 - 3.7.5. Condicional
 - 3.8. Claves foráneas
 - 3.9. Peticiones JOIN
 - 3.10. Vistas SQL
 - 3.11. Exportación e importación
 - 3.11.1. Formatos de exportación
 - 3.11.2. MySQL DUMP
- 4. Tipos de datos
 - 4.1. Numéricos
 - 4.1.1. Enteros
 - 4.1.2. Decimales
 - 4.1.3. Otros
 - 4.2. Fechas
 - 4.3. Cadena
 - 4.4. JSON
- 5. Motores de almacenamiento
 - 5.1. InnoDB
 - 5.2. MyISAM
 - 5.3. Merge MyISAM
 - 5.4. CSV
 - 5.5. Memory
 - 5.6. Aria
 - 5.7. Sequence
- 6. Operaciones
 - 6.1. Importar provincias
 - 6.2. Códigos postales
 - 6.3. Creación de impuestos

- 6.4. Modificación y ampliación
- 6.5. Borrar
- 7. Base de datos de empresa
- 8. Procedimientos almacenados
- 9. Eventos
- 10. TCL
- 11. Disparadores
- 12. SQL
- 13. Joins
- 14. Insertar y entrar
- 15. Subconsultas
 - 15.1. Preparacion
 - 15.2. Where
 - 15.3. From
 - 15.4. Select
 - 15.5. Subconsulta relacion
- 16. Epílogo

1. Prólogo

La gestión eficiente de los datos es crucial para cualquier empresa moderna. SQL, el lenguaje de consulta estructurado, es la herramienta esencial que permite interactuar, manipular y administrar bases de datos de manera efectiva. Desde grandes corporaciones hasta pequeños negocios, SQL se ha consolidado como un estándar indispensable para almacenar, recuperar y analizar información clave.

Este libro surge con la intención de ofrecer al lector una guía práctica, clara y concisa sobre el uso de SQL, partiendo desde los conceptos fundamentales hasta abarcar temas avanzados como procedimientos almacenados, vistas, eventos, transacciones, disparadores y consultas complejas. Cada capítulo está cuidadosamente estructurado para facilitar la comprensión a través de ejemplos concretos, acompañados de ejercicios que permitirán afianzar los conocimientos adquiridos y aplicarlos en situaciones reales.

Sea que estés dando tus primeros pasos en el mundo del desarrollo de bases de datos o que busques profundizar y perfeccionar tus habilidades, este libro es un compañero valioso que te guiará paso a paso en el aprendizaje de SQL. Con cada consulta y operación que realices, estarás fortaleciendo tu capacidad para tomar decisiones estratégicas basadas en datos.

Bienvenido a este recorrido por el apasionante mundo de SQL. Espero que disfrutes el aprendizaje y que encuentres en estas páginas las herramientas que necesitas para llevar tus proyectos de datos al siguiente nivel.

2. Cómo instalar un servidor local

Aquí tienes un texto introductorio seguido de instrucciones detalladas para instalar un servidor local en Windows, macOS y Linux, con el objetivo de realizar las prácticas del libro.

2.1. Instalación de un entorno de desarrollo local

2.1.1. Introducción

Para poder realizar las prácticas del libro de forma efectiva y sin depender de un servidor externo, vamos a instalar un entorno de desarrollo local. Esto te permitirá crear bases de datos, ejecutar consultas SQL y gestionar usuarios de forma totalmente autónoma en tu propio ordenador.

Utilizaremos **MySQL**, uno de los sistemas de gestión de bases de datos más populares y compatibles con los ejemplos presentados en el libro. A continuación, encontrarás instrucciones específicas para cada sistema operativo: Windows, macOS y Linux.

2.1.2. Instalación en Windows

1. Descargar MySQL Installer

- Visita <https://dev.mysql.com/downloads/installer/>. - Descarga la versión "MySQL Installer for Windows" (preferentemente la que incluye todos los productos: mysql-installer-community).

1. Instalar MySQL

- Ejecuta el instalador. - Elige la opción “Developer Default” o “Server only”. - Durante el proceso, configura la contraseña para el usuario root. - Acepta los valores por defecto para el puerto y opciones de red.

1. Verificar la instalación

- Abre MySQL Command Line Client desde el menú Inicio. - Escribe la contraseña que configuraste para root. - Prueba con el comando:

Consulta SQL:

```
SHOW DATABASES;
```

1. (Opcional) Instalar MySQL Workbench

- Es una interfaz gráfica para trabajar más cómodamente con tus bases de datos.

2.1.3. Instalación en macOS

1. Instalar Homebrew (si no lo tienes)

Abre la Terminal y ejecuta:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install
```

1. Instalar MySQL con Homebrew

```
brew install mysql
```

1. Iniciar el servicio de MySQL

```
brew services start mysql
```

1. Acceder al cliente de MySQL

```
mysql -u root
```

1. (Opcional) Cambiar la contraseña del usuario root

Consulta SQL:

```
ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'tucco';
```

1. (Opcional) Instalar MySQL Workbench

Descárgalo desde <https://dev.mysql.com/downloads/workbench/>.

2.1.4. Instalación en Linux (Debian/Ubuntu)

1. Actualizar los repositorios

```
sudo apt update
```

1. Instalar el servidor de MySQL

```
sudo apt install mysql-server
```

1. Iniciar sesión en MySQL

```
sudo mysql
```

1. (Opcional) Configurar una contraseña para el usuario root

Consulta SQL:

```
ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'tucontraseña';
```

1. (Opcional) Instalar MySQL Workbench

```
sudo apt install mysql-workbench
```

Con este entorno local ya estarás listo para seguir los ejemplos y ejercicios del libro. A lo largo del texto encontrarás instrucciones para crear bases de datos, tablas, insertar datos, hacer consultas complejas y realizar operaciones más avanzadas con SQL. ¡Vamos allá!

3. Fundamentos de SQL

3.1. DDL

El lenguaje DDL (Data Definition Language) es una de las partes fundamentales de SQL, y su propósito es definir y estructurar los elementos que componen una base de datos. A diferencia de otros subconjuntos de SQL que se centran en la manipulación o consulta de datos, el DDL se enfoca en establecer el esqueleto sobre el que se construyen las aplicaciones de bases de datos.

Con DDL podemos crear nuevas bases de datos, diseñar las tablas que las conforman, establecer relaciones entre datos, y también modificar o eliminar estructuras existentes. Es decir, define cómo se almacenarán los datos, pero no manipula los datos en sí.

Entre las instrucciones más comunes de DDL encontramos:

- CREATE: para crear bases de datos, tablas, índices, vistas, etc.
- ALTER: para modificar la estructura de una tabla existente (añadir, eliminar o cambiar columnas, por ejemplo).
- DROP: para eliminar objetos de la base de datos como tablas o bases de datos completas.
- TRUNCATE: para vaciar completamente una tabla sin eliminar su estructura.
- RENAME: para cambiar el nombre de una tabla.
- COMMENT: para documentar la estructura mediante comentarios asociados a tablas o columnas.

Estas instrucciones son fundamentales en la fase inicial de diseño y despliegue de cualquier sistema que utilice bases de datos, y conocerlas a fondo es esencial para cualquier profesional que trabaje con SQL.

Creación de una base de datos

La creación de una base de datos es el primer paso en cualquier proyecto que requiera almacenar y gestionar información de forma estructurada. En SQL, este proceso se realiza mediante la sentencia CREATE DATABASE. Con ella, indicamos al sistema gestor que queremos definir un nuevo espacio donde podremos posteriormente crear tablas, insertar datos y establecer relaciones.

La base de datos es el contenedor principal de toda la estructura lógica de almacenamiento. Es fundamental definirla desde el inicio con un nombre representativo, claro y conciso, que nos ayude a identificar su propósito dentro del sistema.

Consulta SQL:

```
CREATE DATABASE empresa;
```

Resultado de la consulta en el terminal:

```
Query OK, 1 row affected (0,00 sec)
```

Eliminación de una base de datos

La eliminación de una base de datos es una operación crítica que borra de forma permanente toda la información contenida en ella. Para llevar a cabo esta acción en SQL, se utiliza la sentencia DROP DATABASE.

Esta instrucción elimina no solo las tablas, sino también todos los datos, vistas, procedimientos almacenados y cualquier otro objeto definido dentro de la base de datos. Por este motivo, debe utilizarse con extrema precaución, especialmente en entornos de producción, ya que no se puede deshacer.

Es una herramienta útil cuando ya no se necesita una base de datos determinada, o cuando se está realizando una limpieza o reestructuración del sistema.

Consulta SQL:

```
DROP DATABASE empresa;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,01 sec)
```

Nota: La eliminación de la base de datos es opcional: se muestra como información complementaria

Uso de la base de datos

Una vez creada una base de datos, es necesario indicarle al sistema gestor sobre cuál de ellas queremos trabajar. Para ello se utiliza la sentencia USE, seguida del nombre de la base de datos que deseamos activar.

Esta instrucción establece el contexto para las operaciones que se realicen a continuación. A partir de su ejecución, todas las sentencias SQL que no especifiquen explícitamente otra base de datos se aplicarán sobre la que ha sido seleccionada.

Es especialmente útil cuando se está trabajando con múltiples bases de datos dentro del mismo servidor, ya que permite cambiar rápidamente de una a otra sin necesidad de cerrar sesión o reiniciar la conexión.

Consulta SQL:

```
USE empresa;
```

Resultado de la consulta en el terminal:

Database changed

Creación de nuevas tablas

Una vez que disponemos de una base de datos sobre la que trabajar, el siguiente paso consiste en definir las estructuras que almacenarán la información. Esto se hace mediante la creación de tablas. Una tabla es una entidad que organiza los datos en filas y columnas, y cada columna representa un campo con un tipo de dato específico.

En este caso, se crea una tabla llamada usuarios dentro de la base de datos empresa. Esta tabla está diseñada para almacenar la información básica de los usuarios del sistema. Cada fila representará un usuario distinto y cada columna contendrá un dato específico como el nombre, los apellidos o el correo electrónico.

El campo Identificador es un entero que se incrementa automáticamente con cada nuevo registro y actúa como clave primaria, lo que garantiza que cada usuario pueda ser identificado de forma única.

La elección de tipos de datos como VARCHAR permite almacenar cadenas de longitud variable, y la restricción NOT NULL asegura que todos los campos deben contener información obligatoriamente. El uso del motor de almacenamiento InnoDB permite disponer de características avanzadas como integridad referencial y transacciones.

Además, se incluye un comentario en la tabla para documentar su propósito, lo cual es una buena práctica cuando se diseñan bases de datos que deben ser mantenidas por distintos equipos o a lo largo del tiempo.

Consulta SQL:

```
CREATE TABLE `empresa`.`usuarios`  
(  
    `Identificador` INT(10) NOT NULL AUTO_INCREMENT ,  
    `nombre` VARCHAR(50) NOT NULL ,  
    `contraseña` VARCHAR(50) NOT NULL ,  
    `nombrepropio` VARCHAR(50) NOT NULL ,  
    `apellidos` VARCHAR(100) NOT NULL ,  
    `email` VARCHAR(100) NOT NULL ,  
    `telefono` VARCHAR(50) NOT NULL ,  
    PRIMARY KEY (`Identificador`)  
)  
ENGINE = InnoDB  
COMMENT = 'En esta tabla guardaremos los usuarios';
```

Resultado de la consulta en el terminal:

Query OK, 0 rows affected, 1 warning (0,03 sec)

Eliminación de tablas

Eliminar una tabla de la base de datos implica borrar por completo su estructura y todo el contenido que haya almacenado. Para ello se utiliza la sentencia `DROP TABLE`, seguida del nombre de la tabla que se desea eliminar.

Esta operación es irreversible: al ejecutarla, se eliminan todos los datos, índices, restricciones y definiciones asociadas a la tabla. Por ese motivo, debe utilizarse con precaución, especialmente si la tabla contiene información importante o forma parte de relaciones con otras tablas.

La eliminación de tablas suele realizarse durante fases de desarrollo, pruebas o reestructuración del modelo de datos.

Consulta SQL:

```
DROP TABLE usuarios;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,01 sec)
```

Vaciado de tablas

La sentencia `TRUNCATE TABLE` se utiliza para eliminar rápidamente todos los registros de una tabla, sin borrar su estructura. A diferencia de `DROP TABLE`, que elimina por completo la tabla, `TRUNCATE` conserva su definición, permitiendo seguir utilizándola inmediatamente después.

Este comando es útil cuando se desea vaciar una tabla de forma eficiente, especialmente si contiene un gran volumen de datos. Internamente, suele ser más rápido que un `DELETE` sin cláusula `WHERE`, ya que no genera tantos registros de transacción ni activa disparadores (triggers) en la mayoría de los gestores de bases de datos.

Tras ejecutar `TRUNCATE`, la tabla queda vacía, como recién creada, y si contiene un campo con `AUTO_INCREMENT`, este contador normalmente se reinicia.

Consulta SQL:

```
TRUNCATE TABLE usuarios;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,03 sec)
```

Renombrado de tablas

La sentencia `RENAME TABLE` permite cambiar el nombre de una tabla existente dentro de

una base de datos. Esta operación es útil cuando se desea actualizar la nomenclatura para hacerla más representativa, o bien por motivos de organización o estandarización del esquema.

En este caso, la tabla `usuarios` pasa a llamarse `misusuarios`, conservando intactos tanto sus datos como su estructura. Es importante tener en cuenta que, al renombrar una tabla, cualquier consulta, vista o procedimiento que hiciera referencia a su nombre anterior deberá actualizarse para seguir funcionando correctamente.

Consulta SQL:

```
RENAME TABLE  
`empresa`.`usuarios`  
TO  
`empresa`.`misusuarios`;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,02 sec)
```

Devolviendo su nombre original a la tabla

Renombrar una tabla es una operación reversible, y puede realizarse cuantas veces sea necesario para ajustar la estructura del modelo a nuevas necesidades o convenciones. En este caso, se devuelve el nombre original `usuarios` a la tabla que previamente se había renombrado como `misusuarios`.

Este tipo de acciones es común durante el desarrollo o mantenimiento de una base de datos, cuando se prueban cambios en la estructura y se necesita alternar entre distintos nombres de forma temporal. Como siempre, es importante asegurarse de que cualquier referencia al nombre anterior se actualice si se mantiene en el sistema.

Consulta SQL:

```
RENAME TABLE  
`empresa`.`misusuarios`  
TO  
`empresa`.`usuarios`;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,01 sec)
```

Alteración de tablas

Con el tiempo, es habitual que una tabla necesite adaptarse a nuevas necesidades de almacenamiento. Para ello, SQL ofrece la sentencia `ALTER TABLE`, que permite modificar la estructura de una tabla existente sin perder los datos ya almacenados.

En este caso, se añade una nueva columna llamada `fotografia` a la tabla `usuarios`. Esta columna se usará para almacenar una cadena de texto (por ejemplo, el nombre de un archivo de imagen o una URL), y se ha definido como obligatoria con `NOT NULL`.

La cláusula `AFTER` permite indicar la posición exacta en la que se insertará la nueva columna dentro de la tabla, en este caso, inmediatamente después de `telefono`. Aunque el orden de las columnas no afecta funcionalmente a la base de datos, puede ser útil para mantener la tabla organizada de forma lógica o coherente con la interfaz de usuario.

Consulta SQL:

```
ALTER TABLE
`usuarios`
ADD
`fotografia` VARCHAR(100) NOT NULL
AFTER `telefono`;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,03 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Creación de la tabla de clientes

La tabla `clientes` está diseñada para almacenar toda la información relevante sobre los clientes de una empresa. Cada registro representa un cliente distinto e incluye tanto datos generales como identificativos y de contacto.

La estructura define un campo `Identificador` que funciona como clave primaria mediante `AUTO_INCREMENT`, asegurando que cada cliente tenga un identificador único. El campo `nombre` permite registrar la razón social o denominación del cliente, mientras que `idfiscal` almacena su identificación fiscal.

Además, se incluye la dirección y el código postal del cliente, junto con los datos de la persona de contacto asociada a la empresa: nombre y correo electrónico. Finalmente, el campo `imagen` está destinado a almacenar una referencia visual, como un logotipo o fotografía, que puede utilizarse en interfaces gráficas o documentos.

Esta tabla utiliza el motor de almacenamiento `InnoDB`, que permite mantener la integridad de los datos y soportar relaciones entre tablas, lo cual será especialmente útil al trabajar con claves foráneas en fases posteriores del desarrollo.

Consulta SQL:

```
CREATE TABLE clientes
(
    Identificador INT(10) NOT NULL AUTO_INCREMENT ,
    nombre VARCHAR(150) NOT NULL ,
    idfiscal VARCHAR(50) NOT NULL ,
    direccion VARCHAR(250) NOT NULL ,
    codigopostal VARCHAR(20) NOT NULL ,
    nombrepersonacontacto VARCHAR(250) NOT NULL ,
    emailpersonacontacto VARCHAR(100) NOT NULL ,
    imagen VARCHAR(100) NOT NULL ,
    PRIMARY KEY (Identificador)
)
ENGINE = InnoDB
COMMENT = 'En esta tabla guardaremos los clientes';
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 1 warning (0,02 sec)
```

3.2. DML

Una vez creada la estructura de una base de datos mediante DDL, el siguiente paso es comenzar a trabajar con los datos que esa estructura contendrá. El lenguaje de manipulación de datos permite insertar nuevos registros en las tablas, actualizar la información existente y eliminar aquellos datos que ya no son necesarios. Estas acciones son fundamentales para el mantenimiento y evolución de cualquier sistema basado en bases de datos.

A través de instrucciones como INSERT, UPDATE y DELETE, es posible gestionar de forma dinámica el contenido de las tablas. Por ejemplo, se puede registrar la información de un nuevo cliente, actualizar su dirección o eliminar su ficha si ya no es relevante. Estas operaciones deben realizarse respetando las reglas y restricciones definidas previamente en el diseño de la base de datos, como claves primarias, claves foráneas o validaciones de tipo de datos.

El uso correcto del DML garantiza la integridad de los datos y permite construir aplicaciones fiables que respondan a las necesidades reales de gestión de información. Además, muchas de estas operaciones pueden combinarse con condiciones para realizar acciones selectivas, evitando así afectar datos que no deben ser modificados o eliminados. **Inserción de registros**

Una vez definida la estructura de una tabla, el siguiente paso es comenzar a poblarla con datos. Para ello se utiliza la sentencia INSERT INTO, que permite añadir un nuevo registro especificando los valores para cada uno de los campos.

En este ejemplo, se inserta un cliente completo en la tabla clientes, proporcionando información para todos los campos definidos. El primer valor es NULL, lo que indica que

no se especifica manualmente el valor del campo Identificador, ya que éste se generará automáticamente gracias a la propiedad AUTO_INCREMENT.

El resto de los valores incluyen el nombre de la empresa, su identificación fiscal, la dirección, el código postal, el nombre de la persona de contacto, su correo electrónico y el nombre de un archivo de imagen asociado al cliente. Esta inserción representa un caso típico de uso en una aplicación de gestión empresarial, donde se requiere registrar clientes con toda su información desde el principio.

Consulta SQL:

```
INSERT INTO clientes VALUES(
    NULL,
    'Nombre de la empresa 1',
    'C000001',
    'Dirección de la empresa 1',
    '46001',
    'Juan Lopez Martinez',
    'juanlopez@empresa1.com',
    'logempresa1.jpg'
);
```

Resultado de la consulta en el terminal:

```
Query OK, 1 row affected (0,00 sec)
```

Inserción de registros especificando los campos

Es posible insertar datos en una tabla sin necesidad de completar todos los campos, siempre que aquellos que se omitan permitan valores nulos o tengan definidos valores por defecto. Para ello, se especifica explícitamente el listado de columnas en las que se desea insertar información.

En este caso, se añaden los datos básicos de un cliente en la tabla clientes: el nombre de la empresa, su identificación fiscal, dirección y código postal. Se omiten los campos relacionados con la persona de contacto y la imagen, lo cual es válido siempre que esos campos estén configurados en la tabla para aceptar valores nulos o hayan sido definidos como opcionales.

Esta técnica es útil cuando se dispone de información parcial al momento del registro, permitiendo completar los datos más adelante sin bloquear el flujo de trabajo.

Consulta SQL:

```
INSERT INTO clientes
(
    nombre,
    idfiscal,
    direccion,
    codigopostal
)
VALUES
(
    'Nombre de la empresa 2',
    'C000002',
    'Dirección de la empresa 2',
    '46002'
);
```

Resultado de la consulta en el terminal:

```
ERROR 1364 (HY000): Field 'nombrepersonacontacto' doesn't have a default value
```

Error en la inserción

En esta sentencia se intenta insertar un nuevo registro en la tabla clientes, pero no se está indicando el listado de columnas ni se están proporcionando todos los valores necesarios. Esto puede generar un error si la tabla requiere más campos o si el orden de los valores no coincide con la estructura de la tabla.

Cuando se omite la lista de columnas, SQL espera que se proporcionen valores para **todos** los campos definidos en la tabla, en el **mismo orden** en el que fueron creados. En este caso, como la tabla clientes contiene ocho columnas, proporcionar solo cuatro generará un fallo al no cumplir con las restricciones de integridad ni con los requisitos de obligatoriedad (NOT NULL) de los campos restantes.

Este ejemplo destaca la importancia de utilizar siempre la sintaxis completa de `INSERT INTO ... (columnas) VALUES (...)` cuando no se van a insertar todos los datos o cuando no se conoce el orden exacto de las columnas.

Consulta SQL:

```
INSERT INTO clientes
VALUES
(
    'Nombre de la empresa 2',
    'C000002',
    'Dirección de la empresa 2',
    '46002'
);
```

Resultado de la consulta en el terminal:

```
ERROR 1136 (21S01): Column count doesn't match value count at row 1
```

Actualización de los datos

La sentencia UPDATE permite modificar los valores de uno o varios campos en los registros existentes de una tabla. En este ejemplo, se actualizan los datos de contacto de los clientes, concretamente el nombre y el correo electrónico de la persona de contacto.

Sin embargo, al no incluir una cláusula WHERE, esta actualización afectará **a todos los registros** de la tabla clientes. Es decir, todos los clientes almacenados pasarán a tener como persona de contacto a "Jorge Martinez García" y su respectivo correo electrónico.

Este tipo de operaciones sin filtro pueden ser peligrosas si se ejecutan por error, ya que pueden sobrescribir información valiosa de forma masiva. Por ello, es fundamental asegurarse de utilizar WHERE cuando se desee actualizar un registro específico o un conjunto limitado de ellos.

Consulta SQL:

```
UPDATE clientes
SET
nombrepersonacontacto = 'Jorge Martinez García',
emailpersonacontacto = 'jorgemartinez@empresa2.com';
```

Resultado de la consulta en el terminal:

Actualización haciendo uso de WHERE

En este caso se realiza una actualización más precisa utilizando la cláusula WHERE, que permite restringir la operación a un único registro de la tabla. Concretamente, se modifican los datos de contacto del cliente cuyo Identificador es igual a 1.

Esta práctica es la forma habitual de trabajar con actualizaciones en SQL, ya que evita cambios masivos no deseados y permite mantener la integridad de la información. Al identificar de forma única el registro que se quiere modificar, se garantiza que solo se alteran los datos del cliente deseado.

Consulta SQL:

```
UPDATE clientes
SET
nombrepersonacontacto = 'Jose Lopez',
emailpersonacontacto = 'joselopez@empresa1.com'
WHERE
Identificador = 1;
```

Resultado de la consulta en el terminal:

```
Query OK, 1 row affected (0,01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Consulta de eliminación La sentencia `DELETE FROM` permite eliminar registros de una tabla. Cuando se utiliza sin una cláusula `WHERE`, como en este caso, se borran **todos los registros** existentes en la tabla `clientes`, dejando la tabla vacía pero conservando su estructura.

Este tipo de operación puede ser útil en entornos de desarrollo o pruebas, cuando se desea reiniciar el contenido de la tabla. Sin embargo, debe utilizarse con precaución en producción, ya que los datos eliminados no se pueden recuperar fácilmente si no se ha hecho una copia previa.

Consulta SQL:

```
DELETE FROM clientes;
```

Resultado de la consulta en el terminal:

```
Query OK, 1 row affected (0,00 sec)
```

Consulta de eliminación haciendo uso de WHERE

En esta sentencia se elimina un único registro de la tabla `clientes`, identificado de forma específica mediante la cláusula `WHERE`. Solo el cliente cuyo `Identificador` sea igual a 4 será eliminado, mientras que el resto de los datos permanecerán intactos.

Este enfoque es el más seguro y recomendable para eliminar datos, ya que permite un control preciso sobre qué registros se afectan. El uso de la clave primaria como filtro garantiza que se eliminará exactamente un único cliente, evitando errores de eliminación masiva.

Consulta SQL:

```
DELETE FROM clientes
WHERE Identificador = 4;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,00 sec)
```

3.3. DQL

Mientras que DDL define la estructura de la base de datos y DML permite modificar su contenido, el lenguaje de consulta de datos se centra en obtener información. DQL permite realizar consultas sobre las tablas con el fin de recuperar datos específicos, combinarlos, filtrarlos, ordenarlos o incluso generar resultados calculados a partir de los datos existentes.

La instrucción principal de DQL es **SELECT**, que permite extraer información según múltiples criterios. Se puede utilizar para ver todos los registros de una tabla o solo aquellos que cumplan ciertas condiciones. También permite proyectar columnas concretas, asignar alias a los campos, agrupar resultados, realizar cálculos, aplicar funciones agregadas y mucho más.

Este lenguaje es especialmente potente cuando se combina con otras herramientas de SQL, como cláusulas WHERE, operadores lógicos, funciones y uniones de tablas. Gracias a DQL, las bases de datos se convierten en verdaderas fuentes de conocimiento y análisis, capaces de proporcionar respuestas rápidas y precisas a preguntas complejas.

Sentencia de selección

La instrucción **SELECT * FROM** permite consultar y visualizar el contenido completo de una tabla. En este caso, se obtendrán todos los registros y todas las columnas de la tabla **clientes**.

El asterisco (*) actúa como comodín e indica que se desean recuperar **todas las columnas** sin necesidad de enumerarlas una por una. Es una forma rápida y práctica de explorar los datos, especialmente útil en fases de desarrollo, depuración o revisión general.

No obstante, en proyectos más avanzados o en entornos de producción, se recomienda especificar solo las columnas necesarias para optimizar el rendimiento y reducir la carga de transferencia de datos.

Consulta SQL:

```
SELECT * FROM clientes;
```

Resultado de la consulta en el terminal:

```
+-----+-----+
| Identificador | nombre           | idfiscal | direccion
+-----+-----+
| 2             | Nombre de la empresa 1 | C000001   | Dirección de la empresa
+-----+-----+
1 row in set (0,00 sec)
```

Selección de registros especificando las columnas

En esta consulta se recuperan únicamente algunas columnas específicas de la tabla clientes: el nombre de la empresa, su identificación fiscal, la dirección y el código postal.

A diferencia de `SELECT *`, esta forma de consultar datos es más eficiente y clara, ya que permite obtener solo la información relevante para un propósito concreto. Además, facilita la lectura del resultado y reduce el consumo de recursos, especialmente cuando las tablas contienen muchas columnas o gran volumen de datos.

Consulta SQL:

```
SELECT
    nombre,
    idfiscal,
    direccion,
    codigopostal
FROM clientes;
```

Resultado de la consulta en el terminal:

```
+-----+-----+-----+
| nombre           | idfiscal | direccion           | codigopostal
+-----+-----+-----+
| Nombre de la empresa 1 | C000001   | Dirección de la empresa 1 | 46001
+-----+-----+-----+
1 row in set (0,00 sec)
```

Alias de columna

Esta consulta realiza una selección de columnas específicas de la tabla clientes, y además utiliza la cláusula AS para asignar nombres más descriptivos o personalizados a cada columna del resultado.

Esta técnica, conocida como **alias de columna**, mejora la legibilidad de los datos al presentar encabezados más claros o adaptados al contexto en el que se muestran (por ejemplo, en informes, paneles de usuario o exportaciones). Aunque los alias no modifican los nombres reales de las columnas en la tabla, sí permiten mostrar la información de forma más comprensible para el usuario final.

Consulta SQL:

```
SELECT
    nombre AS 'Nombre del cliente',
    idfiscal AS 'Identificación fiscal',
    direccion AS 'Dirección del cliente',
    codigopostal AS 'Código postal'
FROM clientes;
```

Resultado de la consulta en el terminal:

```
+-----+-----+
| Nombre del cliente | Identificación fiscal | Dirección del cliente
+-----+-----+
| Nombre de la empresa 1 | C000001 | Dirección de la empresa 1
+-----+-----+
1 row in set (0,00 sec)
```

3.4. DCL

El lenguaje de control de datos se encarga de gestionar los permisos y la seguridad dentro de un sistema de bases de datos. A través de DCL es posible definir qué usuarios tienen acceso a la base de datos, qué acciones pueden realizar y sobre qué objetos tienen autorización. Esta capa es esencial para garantizar la confidencialidad, integridad y disponibilidad de la información.

Las instrucciones más representativas de DCL son GRANT y REVOKE. Con GRANT se otorgan privilegios a uno o varios usuarios, como la posibilidad de leer datos (SELECT), modificar registros (UPDATE), o incluso administrar la base de datos completa. Por el contrario, REVOKE permite retirar esos privilegios cuando ya no se desea que el usuario tenga acceso a ciertas funcionalidades.

Además, mediante CREATE USER es posible dar de alta nuevos usuarios en el sistema y asociarles credenciales de acceso. En sistemas multiusuario, estas instrucciones cobran especial importancia, ya que permiten aplicar políticas de control de acceso, limitar el uso de recursos o establecer cuotas de operaciones para garantizar un funcionamiento eficiente y seguro del sistema.

Creación de un nuevo usuario en el sistema

La sentencia CREATE USER se utiliza para crear una nueva cuenta de usuario en el sistema gestor de bases de datos. En este caso, se está definiendo un usuario llamado josevicente que podrá conectarse únicamente desde el host local (localhost).

La cláusula IDENTIFIED VIA mysql_native_password especifica el método de autenticación que se utilizará, en este caso el método tradicional de MySQL basado en contraseña. El valor indicado en USING representa el hash o valor cifrado de la contraseña, aunque aquí aparece reemplazado por asteriscos a modo de ejemplo.

Crear usuarios personalizados es una práctica fundamental para mantener la seguridad

y el control de acceso a la base de datos, permitiendo asignar diferentes privilegios según el perfil y las funciones de cada persona o aplicación.

Consulta SQL:

```
CREATE USER 'josevicente'@'localhost'  
IDENTIFIED WITH mysql_native_password  
BY 'josevicente';
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,00 sec)
```

Asignación de permisos para el usuario recién creado

Una vez creado un usuario en el sistema, es necesario asignarle permisos para definir qué acciones puede realizar dentro del gestor de bases de datos. Para ello se utiliza la sentencia GRANT, que otorga uno o varios privilegios a un usuario determinado.

En este caso, al usuario josevicente se le conceden los permisos SELECT (lectura de datos) y SHOW VIEW (visualización de vistas), aplicables a todas las bases de datos y todas las tablas del sistema (ON *.*). Esto significa que podrá consultar información, pero no modificarla.

Además, se establecen límites operativos con cláusulas como MAX_QUERIES_PER_HOUR y MAX_CONNECTIONS_PER_HOUR, que restringen la cantidad de consultas, conexiones y actualizaciones que el usuario puede realizar por hora. Estas restricciones permiten controlar el uso de recursos del servidor y evitar abusos o sobrecargas accidentales.

Consulta SQL:

```
GRANT SELECT, SHOW VIEW ON *.* TO 'josevicente'@'localhost';
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,01 sec)
```

Alteración de permisos para el usuario creado

Consulta SQL:

```
ALTER USER 'josevicente'@'localhost'  
WITH  
    MAX_QUERIES_PER_HOUR 100  
    MAX_CONNECTIONS_PER_HOUR 100  
    MAX_UPDATES_PER_HOUR 100  
    MAX_USER_CONNECTIONS 100;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,01 sec)
```

Permisos para que un usuario pueda acceder a una base de datos Esta instrucción otorga al usuario josevicente todos los privilegios disponibles sobre todas las tablas y objetos de la base de datos empresa. Al utilizar GRANT ALL PRIVILEGES, se le concede acceso total para realizar operaciones como crear, modificar, eliminar tablas, insertar y actualizar datos, gestionar índices, vistas, y más.

El ámbito de aplicación está limitado a la base de datos empresa, lo cual es indicado mediante empresa.*. Esto significa que el usuario tendrá control total dentro de esa base de datos, pero no fuera de ella.

Esta asignación de privilegios es adecuada para usuarios administradores o desarrolladores que necesitan trabajar con todas las funciones disponibles en una base de datos concreta. Como buena práctica, debe usarse con moderación, reservándola solo para roles que realmente requieran acceso completo.

Consulta SQL:

```
GRANT ALL PRIVILEGES ON
`empresa`.*
TO
'josevicente'@'localhost';
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,01 sec)
```

Otorgado de todos los permisos Esta sentencia otorga **todos los privilegios** sobre la base de datos empresa al usuario josevicente, que se conecta desde localhost. Es decir, este usuario podrá realizar cualquier tipo de operación dentro de esa base de datos: crear y eliminar tablas, insertar, modificar y borrar datos, crear vistas, ejecutar procedimientos almacenados, entre otras acciones.

El uso de empresa.* indica que los permisos aplican a todos los objetos (tablas, vistas, procedimientos, etc.) contenidos dentro de la base de datos empresa.

Este tipo de concesión es apropiado para usuarios que necesitan control total sobre una base de datos específica, como administradores o desarrolladores principales. Sin embargo, por razones de seguridad, se recomienda no asignar privilegios globales sin necesidad, y limitar siempre el alcance al mínimo imprescindible.

-- Creación de una contraseña cifrada

La función PASSWORD() en MySQL se utiliza para generar una versión cifrada (hash) de una cadena de texto, generalmente una contraseña. En este caso, la función devuelve el hash correspondiente a la cadena 'josevicente'.

Aunque esta función puede parecer útil para definir contraseñas manualmente, es importante tener en cuenta que `PASSWORD()` está obsoleta y no debe utilizarse para gestionar contraseñas de usuarios, ya que su comportamiento ha cambiado entre versiones de MySQL y no garantiza compatibilidad ni seguridad.

En su lugar, se recomienda utilizar mecanismos actuales de autenticación gestionados por el propio sistema gestor, como `mysql_native_password` o `caching_sha2_password`, los cuales se emplean automáticamente al crear usuarios con contraseñas seguras mediante `CREATE USER` y `IDENTIFIED BY`.

Consulta SQL:

```
SELECT SHA2('josevicente', 256);
```

Resultado de la consulta en el terminal:

```
+-----+  
| SHA2('josevicente', 256) |  
+-----+  
| 83247e9d6794e956782aa9399810c2ac7999b9ddef058e852ea5c09fddb7f0c3 |  
+-----+  
1 row in set (0,00 sec)
```

Creación de un usuario con una contraseña cifrada

En esta sentencia se crea un nuevo usuario llamado `josevicente2`, que solo podrá conectarse desde `localhost`, y se le asigna una contraseña ya cifrada utilizando el plugin de autenticación `mysql_native_password`.

El valor especificado en la cláusula `USING` es un **hash de contraseña** previamente generado, normalmente con la función `PASSWORD()` u otro mecanismo de cifrado compatible con MySQL. Esto permite crear usuarios sin exponer la contraseña en texto plano dentro del script, una práctica recomendable en contextos donde se requiere mayor seguridad.

Este enfoque es útil cuando se desea automatizar la creación de usuarios en diferentes entornos, garantizando que todos utilicen la misma contraseña sin necesidad de compartirla en claro.

Consulta SQL:

```
CREATE USER 'josevicente2'@'localhost'  
IDENTIFIED WITH mysql_native_password  
AS '*078129177D940A1383CB641D1FE9CE2E859A6EDB';
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,01 sec)
```

3.5. Operaciones

En el contexto de SQL, las operaciones representan las consultas y manipulaciones que realizamos sobre los datos para extraer valor de ellos. Estas operaciones incluyen tanto acciones básicas como la selección de registros mediante condiciones, como acciones más avanzadas que permiten aplicar filtros, realizar cálculos, evaluar rangos de valores o utilizar patrones de búsqueda.

Las operaciones pueden combinarse con cláusulas lógicas (AND, OR, NOT), operadores relacionales (=, !=, <, >, BETWEEN, IN, etc.) y funciones de transformación para adaptar la consulta a las necesidades concretas del usuario. También es posible aplicar alias para personalizar los resultados, calcular valores derivados (como impuestos o descuentos), y trabajar con expresiones para construir nuevas columnas dinámicamente.

Estas herramientas permiten no solo acceder a los datos, sino transformarlos, categorizarlos, y presentarlos de forma útil para su interpretación, convirtiendo al lenguaje SQL en un medio no solo para almacenar información, sino también para analizarla y tomar decisiones.

Creación de una tabla de productos

La tabla `productos` está diseñada para almacenar la información detallada de los artículos que una empresa ofrece, ya sean bienes físicos o productos digitales. Cada fila de la tabla representa un producto individual, con un conjunto de atributos que permiten describirlo de forma completa.

El campo `Identificador` funciona como clave primaria y se genera automáticamente gracias a la opción `AUTO_INCREMENT`, garantizando un valor único para cada producto. Los campos `nombre`, `descripcion` y `precio` permiten registrar el nombre comercial, una descripción extensa y el precio del producto, respectivamente.

El campo `categoria` utiliza un tipo de dato `ENUM`, que restringe su contenido a dos opciones: '`fisico`' o '`virtual`'. Esto facilita la clasificación de productos según su naturaleza. El campo `peso` resulta útil para productos físicos, especialmente en operaciones logísticas o de envío, mientras que `imagen` almacena una referencia visual del producto, como un nombre de archivo o ruta de imagen.

Se utiliza el motor de almacenamiento `InnoDB`, que proporciona soporte para transacciones y claves foráneas, y se incluye un comentario explicativo para documentar la finalidad de la tabla. Esta estructura es ideal para sistemas de gestión de inventarios, catálogos de productos o tiendas en línea.

Consulta SQL:

```
CREATE TABLE productos
(
    Identificador INT(10) NOT NULL AUTO_INCREMENT ,
    nombre VARCHAR(150) NOT NULL ,
    descripcion TEXT NOT NULL ,
    precio DECIMAL(10,2) NOT NULL ,
    categoria ENUM('fisico','virtual') NOT NULL ,
    peso DECIMAL(10,2) NOT NULL ,
    imagen VARCHAR(100) NOT NULL ,
    PRIMARY KEY (Identificador)
)
ENGINE = InnoDB
COMMENT = 'En esta tabla guardaremos los productos';
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 1 warning (0,02 sec)
```

Estas instrucciones insertan múltiples registros en la tabla `productos`, representando un catálogo mixto de artículos físicos y virtuales. Cada sentencia `INSERT INTO` añade un nuevo producto con toda su información: nombre, descripción, precio, categoría, peso y nombre de archivo de la imagen asociada.

Los productos físicos como el ratón, teclado y monitor incluyen un valor de peso distinto de cero, útil para cálculos logísticos. Por su parte, los cursos —como el de SQL, PHP y Python— se registran como productos virtuales, con un peso de cero, lo que refleja su naturaleza digital.

Al utilizar `NULL` en el primer campo (`Identificador`), se permite que MySQL genere automáticamente el valor correspondiente, gracias a la propiedad `AUTO_INCREMENT`. Esta forma de inserción resulta muy habitual cuando se desea poblar una base de datos con datos reales o de prueba que sirvan para desarrollar, probar o presentar funcionalidades del sistema.

Consulta SQL:

```
INSERT INTO productos VALUES(
    NULL,
    'Ratón',
    'Ratón de ordenador',
    15.50,
    'fisico',
    0.1,
    'raton.jpg'
);
INSERT INTO productos VALUES(
    NULL,
    'Teclado',
    'Teclado de ordenador',
```

```

20.50,
'fisico',
0.2,
'teclado.jpg'
);
INSERT INTO productos VALUES(
    NULL,
'Monitor',
'Monitor de ordenador',
200.50,
'fisico',
2,
'monitor.jpg'
);
INSERT INTO productos VALUES(
    NULL,
'Curso de SQL',
'Curso de SQL paso a paso',
300.50,
'vertual',
0,
'cursosql.jpg'
);
INSERT INTO productos VALUES(
    NULL,
'Curso de PHP',
'Curso de PHP paso a paso',
200.50,
'vertual',
0,
'cursophp.jpg'
);
INSERT INTO productos VALUES(
    NULL,
'Curso de Python',
'Curso de Python paso a paso',
400.50,
'vertual',
0,
'cursopython.jpg'
);

```

Resultado de la consulta en el terminal:

```
Query OK, 1 row affected (0,01 sec)
```

Esta consulta selecciona todos los productos cuyo precio sea mayor a 100. Al utilizar `SELECT *`, se recuperan todas las columnas de la tabla `productos` para aquellos registros que cumplan con la condición especificada en la cláusula `WHERE`.

Este tipo de filtro es útil cuando se desea analizar o destacar productos de mayor valor dentro del catálogo, por ejemplo, para identificar artículos premium, generar informes de ventas o aplicar promociones específicas. La condición `precio > 100` actúa como un

criterio de segmentación económica dentro de los datos almacenados.

Consulta SQL:

```
SELECT *
FROM productos
WHERE precio > 100;
```

Resultado de la consulta en el terminal:

Identificador	nombre	descripcion	precio	cat
3	Monitor	Monitor de ordenador	200.50	fis
4	Curso de SQL	Curso de SQL paso a paso	300.50	vir
5	Curso de PHP	Curso de PHP paso a paso	200.50	vir
6	Curso de Python	Curso de Python paso a paso	400.50	vir

4 rows in set (0,00 sec)

Esta consulta recupera todos los productos cuyo precio sea inferior a 100. Al emplear `SELECT *`, se obtienen todas las columnas de cada producto que cumpla con la condición establecida.

Este tipo de consulta es útil para identificar artículos económicos o accesibles dentro del catálogo, por ejemplo, con fines promocionales, análisis de precios, o para mostrar productos destacados en un rango de bajo coste. La cláusula `WHERE precio < 100` permite segmentar los datos según su valor económico.

Consulta SQL:

```
SELECT *
FROM productos
WHERE precio < 100;
```

Resultado de la consulta en el terminal:

Identificador	nombre	descripcion	precio	categoria	peso
1	Ratón	Ratón de ordenador	15.50	fisico	0.10
2	Teclado	Teclado de ordenador	20.50	fisico	0.20

2 rows in set (0,00 sec)

Esta consulta selecciona todos los productos cuyo precio sea exactamente igual a 15.50. Al utilizar `SELECT *`, se extraen todas las columnas de los registros que cumplen con esa condición.

Este tipo de filtro es útil cuando se quiere localizar un producto específico por su

precio, verificar si existe un artículo con un valor determinado o realizar análisis puntuales sobre precios fijos. La igualdad exacta implica que el valor debe coincidir exactamente, incluyendo los decimales.

Consulta SQL:

```
SELECT *
FROM productos
WHERE precio = 15.50;
```

Resultado de la consulta en el terminal:

Esta consulta recupera todos los productos cuyo precio **sea diferente** de 15.50. La condición `precio != 15.50` excluye únicamente aquellos productos que tienen ese valor exacto, devolviendo el resto.

Este tipo de operación es útil cuando se desea analizar todos los productos excepto uno con un precio específico, o cuando se quiere aplicar un tratamiento diferenciado a ciertos valores excluyendo casos particulares. Es una forma de filtrar resultados por exclusión.

Consulta SQL:

```
SELECT *
FROM productos
WHERE precio != 15.50;
```

Resultado de la consulta en el terminal:

Identificador	nombre	descripcion	precio	categoria	peso	...
1	Ratón	Ratón de ordenador	15.50	fisico	0.10	...

1 row in set (0,00 sec)

Esta consulta selecciona todos los productos cuyo precio sea mayor a 100 **y** cuya categoría sea 'fisico'. La combinación de condiciones mediante el operador lógico AND implica que **ambos criterios deben cumplirse simultáneamente** para que un producto sea incluido en el resultado.

Este tipo de consulta permite realizar segmentaciones más específicas, como identificar productos físicos de alto valor. Es especialmente útil en análisis de inventario, estrategias de venta o planificación logística, donde interesa enfocar la atención en un subconjunto concreto de productos que cumplan con múltiples condiciones.

Consulta SQL:

```
SELECT *
FROM productos
WHERE
precio > 100
AND
categoria = 'fisico';
```

Resultado de la consulta en el terminal:

Identificador	nombre	descripcion	precio	categoria	peso
3	Monitor	Monitor de ordenador	200.50	fisico	2.00

1 row in set (0,00 sec)

Esta consulta obtiene todos los productos que cumplen al menos una de estas dos condiciones: que su precio sea mayor a 100 o que su categoría sea 'fisico'.

El operador lógico OR permite que se incluyan en el resultado tanto los productos caros (independientemente de su categoría) como aquellos que sean físicos (sin importar su precio). De esta manera, se amplía el conjunto de resultados, ya que basta con que se cumpla una de las condiciones para que un producto sea seleccionado.

Este tipo de consulta es útil cuando se quiere analizar un conjunto más diverso de productos que cumplan diferentes criterios de inclusión.

Consulta SQL:

```
SELECT *
FROM productos
WHERE
precio > 100
OR
categoria = 'fisico';
```

Resultado de la consulta en el terminal:

Identificador	nombre	descripcion	precio	cat
1 Ratón	Ratón de ordenador	15.50	fis	
2 Teclado	Teclado de ordenador	20.50	fis	
3 Monitor	Monitor de ordenador	200.50	fis	
4 Curso de SQL	Curso de SQL paso a paso	300.50	vir	
5 Curso de PHP	Curso de PHP paso a paso	200.50	vir	
6 Curso de Python	Curso de Python paso a paso	400.50	vir	

6 rows in set (0,00 sec)

Esta consulta selecciona todos los productos cuyo precio sea **mayor o igual** a 200.50. El operador `>=` incluye tanto los productos que superan ese valor como aquellos que lo igualan exactamente.

Este tipo de condición es útil cuando se desea establecer un umbral mínimo de precio, por ejemplo, para identificar productos de alta gama, aplicar descuentos exclusivos o analizar el comportamiento de los artículos más costosos del catálogo.

Consulta SQL:

```
SELECT *
FROM productos
WHERE precio >= 200.50;
```

Resultado de la consulta en el terminal:

Identificador	nombre	descripcion	precio	cat
3 Monitor	Monitor de ordenador	200.50	fis	
4 Curso de SQL	Curso de SQL paso a paso	300.50	vir	
5 Curso de PHP	Curso de PHP paso a paso	200.50	vir	
6 Curso de Python	Curso de Python paso a paso	400.50	vir	

4 rows in set (0,00 sec)

Esta consulta recupera todos los productos cuyo precio sea **menor o igual** a 200.50. La condición `<=` permite incluir tanto los productos con precios inferiores como aquellos que coinciden exactamente con ese valor.

Este tipo de filtro es útil para listar artículos dentro de un rango accesible o moderado de precio, analizar el comportamiento de los productos más económicos o preparar catálogos enfocados en promociones o presupuestos limitados.

Consulta SQL:

```
SELECT *
FROM productos
WHERE precio <= 200.50;
```

Resultado de la consulta en el terminal:

Identificador	nombre	descripcion	precio	categoria
1	Ratón	Ratón de ordenador	15.50	fisico
2	Teclado	Teclado de ordenador	20.50	fisico
3	Monitor	Monitor de ordenador	200.50	fisico
5	Curso de PHP	Curso de PHP paso a paso	200.50	virtual

4 rows in set (0,00 sec)

Esta consulta selecciona todos los productos cuyo precio esté **entre 20 y 100**, excluyendo ambos extremos. La combinación de condiciones con AND obliga a que el precio sea **mayor que 20 y menor que 100** al mismo tiempo.

Este tipo de filtro es ideal para identificar productos dentro de un rango de precios específico, como una franja de productos de precio medio. Es especialmente útil para crear secciones en tiendas online, aplicar descuentos selectivos o realizar análisis por tramos de valor.

Consulta SQL:

```
SELECT *
FROM productos
WHERE
precio > 20
AND
precio < 100;
```

Resultado de la consulta en el terminal:

Identificador	nombre	descripcion	precio	categoria	peso
2	Teclado	Teclado de ordenador	20.50	fisico	0.20

1 row in set (0,00 sec)

Esta consulta selecciona todos los productos cuyo precio se encuentra **entre 10 y 100**, incluyendo ambos valores extremos. El uso de la cláusula BETWEEN simplifica la expresión al agrupar dos condiciones en una sola.

Este tipo de consulta es muy útil cuando se desea trabajar con intervalos de precios

definidos, permitiendo localizar fácilmente productos dentro de un rango económico concreto. Su sintaxis clara y directa mejora la legibilidad y es especialmente práctica en informes, filtros dinámicos o interfaces de usuario que permiten búsquedas por rangos.

Consulta SQL:

```
SELECT *
FROM productos
WHERE
precio BETWEEN 10 AND 100;
```

Resultado de la consulta en el terminal:

```
+-----+-----+-----+-----+-----+
| Identificador | nombre | descripcion | precio | categoria | peso |
+-----+-----+-----+-----+-----+
|           1 | Ratón   | Ratón de ordenador | 15.50 | fisico    | 0.10 |
|           2 | Teclado | Teclado de ordenador | 20.50 | fisico    | 0.20 |
+-----+-----+-----+-----+-----+
2 rows in set (0,00 sec)
```

Esta consulta selecciona todos los productos cuyo nombre sea exactamente "Ratón" o "Teclado". Utiliza el operador lógico OR para combinar ambas condiciones, de modo que basta con que se cumpla una de ellas para que el producto sea incluido en el resultado.

Este tipo de consulta es útil cuando se desea obtener información sobre productos concretos dentro del catálogo, por ejemplo, para generar comparativas, mostrar detalles específicos o aplicar promociones individuales a ciertos artículos.

Consulta SQL:

```
SELECT *
FROM productos
WHERE
nombre = "Ratón"
OR
nombre = "Teclado";
```

Resultado de la consulta en el terminal:

```
+-----+-----+-----+-----+-----+
| Identificador | nombre | descripcion | precio | categoria | peso |
+-----+-----+-----+-----+-----+
|           1 | Ratón   | Ratón de ordenador | 15.50 | fisico    | 0.10 |
|           2 | Teclado | Teclado de ordenador | 20.50 | fisico    | 0.20 |
+-----+-----+-----+-----+-----+
2 rows in set (0,00 sec)
```

Esta consulta obtiene todos los productos cuyo nombre coincide con alguno de los valores especificados en la lista entre paréntesis. La cláusula IN permite simplificar la escritura cuando se desea comparar un campo con múltiples valores posibles.

En este caso, se recuperan los productos llamados 'Ratón' o 'Teclado', de forma equivalente a usar múltiples condiciones con OR, pero con una sintaxis más limpia y escalable. Esta forma resulta especialmente útil cuando se trabaja con listas largas de posibles valores.

Consulta SQL:

```
SELECT *
FROM productos
WHERE
nombre IN ('Ratón','Teclado');
```

Resultado de la consulta en el terminal:

```
+-----+-----+-----+-----+-----+
| Identificador | nombre | descripcion | precio | categoria | peso |
+-----+-----+-----+-----+-----+
| 1 | Ratón | Ratón de ordenador | 15.50 | fisico | 0.10 |
| 2 | Teclado | Teclado de ordenador | 20.50 | fisico | 0.20 |
+-----+-----+-----+-----+-----+
2 rows in set (0,00 sec)
```

Esta consulta selecciona todos los productos cuyo nombre **empiece por** la palabra 'Curso'. La cláusula LIKE junto con el comodín % permite realizar búsquedas por patrón, donde % representa cualquier secuencia de caracteres.

En este caso, se obtendrán productos como 'Curso de SQL', 'Curso de PHP', 'Curso de Python', etc. Este tipo de filtro es muy útil para clasificar productos por tipo, identificar categorías implícitas en nombres o implementar funciones de búsqueda parcial en aplicaciones.

Consulta SQL:

```
SELECT *
FROM productos
WHERE
nombre LIKE 'Curso%';
```

Resultado de la consulta en el terminal:

```
+-----+-----+-----+-----+
| Identificador | nombre           | descripcion          | precio | cat
+-----+-----+-----+-----+
|           4 | Curso de SQL      | Curso de SQL paso a paso | 300.50 | vir
|           5 | Curso de PHP       | Curso de PHP paso a paso | 200.50 | vir
|           6 | Curso de Python    | Curso de Python paso a paso | 400.50 | vir
+-----+-----+-----+-----+
3 rows in set (0,00 sec)
```

Esta consulta selecciona todos los productos cuyo nombre **no comienza** por la palabra 'Curso'. La cláusula NOT LIKE invierte el patrón de coincidencia, excluyendo aquellos registros que sí lo cumplan.

Es útil para filtrar productos que **no pertenezcan** a una categoría determinada basada en el nombre, como en este caso los artículos que no son cursos. Este tipo de consulta permite separar tipos de productos, limpiar resultados o realizar análisis por exclusión.

Consulta SQL:

```
SELECT *
FROM productos
WHERE
nombre NOT LIKE 'Curso%';
```

Resultado de la consulta en el terminal:

```
+-----+-----+-----+-----+
| Identificador | nombre   | descripcion          | precio | categoria | peso |
+-----+-----+-----+-----+
|           1 | Ratón    | Ratón de ordenador | 15.50  | fisico    | 0.10 |
|           2 | Teclado  | Teclado de ordenador | 20.50  | fisico    | 0.20 |
|           3 | Monitor  | Monitor de ordenador | 200.50 | fisico    | 2.00 |
+-----+-----+-----+-----+
3 rows in set (0,00 sec)
```

Esta consulta selecciona varios campos de la tabla productos y además realiza un cálculo sobre el precio para obtener el importe del impuesto aplicando un **IVA del 21%**. Cada columna se presenta con un **alias** personalizado mediante la cláusula AS, lo que mejora la claridad del resultado.

La expresión precio*0.21 calcula el valor del impuesto para cada producto, asumiendo que todos están sujetos al mismo tipo impositivo. Este tipo de consulta es muy útil para generar listados comerciales, facturas o informes financieros que requieren mostrar el desglose entre precio base e impuestos.

Consulta SQL:

```
SELECT
nombre AS 'Nombre del producto',
descripcion AS 'Descripción',
precio AS 'Precio',
precio*0.21 AS 'Impuesto IVA 21%'
FROM productos;
```

Resultado de la consulta en el terminal:

```
+-----+-----+-----+
| Nombre del producto | Descripción           | Precio | Impuesto IVA 21%
+-----+-----+-----+
| Ratón               | Ratón de ordenador    | 15.50  | 3.255
| Teclado             | Teclado de ordenador  | 20.50  | 4.305
| Monitor             | Monitor de ordenador  | 200.50 | 42.105
| Curso de SQL         | Curso de SQL paso a paso | 300.50 | 63.105
| Curso de PHP         | Curso de PHP paso a paso | 200.50 | 42.105
| Curso de Python      | Curso de Python paso a paso | 400.50 | 84.105
+-----+-----+-----+
6 rows in set (0,00 sec)
```

Esta consulta no solo muestra el nombre, la descripción y el precio de cada producto, sino que también calcula el **importe del IVA al 21%** y el **precio total con impuesto incluido**.

La columna `precio*0.21` representa el valor del impuesto, mientras que `precio + precio*0.21` muestra el resultado final que el cliente pagaría. Los alias asignados mediante AS permiten que los encabezados del resultado sean claros y fáciles de interpretar.

Este tipo de consulta es especialmente útil en contextos de facturación, catálogos de precios o sistemas de punto de venta donde se necesita mostrar el desglose completo del precio para cada producto.

Consulta SQL:

```
SELECT
nombre AS 'Nombre del producto',
descripcion AS 'Descripción',
precio AS 'Precio',
precio*0.21 AS 'Impuesto IVA 21%',
precio + precio*0.21 AS 'Total'
FROM productos;
```

Resultado de la consulta en el terminal:

```
+-----+-----+-----+-----+
| Nombre del producto | Descripción           | Precio | Impuesto IVA 21%
+-----+-----+-----+-----+
| Ratón               | Ratón de ordenador      | 15.50  | 3.255
| Teclado             | Teclado de ordenador     | 20.50  | 4.305
| Monitor             | Monitor de ordenador     | 200.50 | 42.105
| Curso de SQL         | Curso de SQL paso a paso | 300.50 | 63.105
| Curso de PHP         | Curso de PHP paso a paso | 200.50 | 42.105
| Curso de Python       | Curso de Python paso a paso | 400.50 | 84.105
+-----+-----+-----+-----+
6 rows in set (0,00 sec)
```

Esta consulta presenta una vista detallada de los productos, incluyendo su nombre, descripción y precio base, junto con un cálculo automático del **IVA al 21%** y el **precio total con impuestos incluidos**.

El uso de expresiones aritméticas directamente en la consulta permite calcular dinámicamente los valores sin necesidad de almacenarlos en la base de datos. Además, los alias asignados a cada columna mejoran la comprensión de los resultados, haciéndolos más legibles y adecuados para su presentación en informes, listados comerciales o tickets de venta.

Consulta SQL:

```
SELECT
nombre AS 'Nombre del producto',
descripción AS 'Descripción',
precio AS 'Precio',
precio*0.21 AS 'Impuesto IVA 21%',
precio + precio*0.21 AS 'Total'
FROM productos;
```

Resultado de la consulta en el terminal:

```
+-----+-----+-----+-----+
| Nombre del producto | Descripción           | Precio | Impuesto IVA 21%
+-----+-----+-----+-----+
| Ratón               | Ratón de ordenador      | 15.50  | 3.255
| Teclado             | Teclado de ordenador     | 20.50  | 4.305
| Monitor             | Monitor de ordenador     | 200.50 | 42.105
| Curso de SQL         | Curso de SQL paso a paso | 300.50 | 63.105
| Curso de PHP         | Curso de PHP paso a paso | 200.50 | 42.105
| Curso de Python       | Curso de Python paso a paso | 400.50 | 84.105
+-----+-----+-----+-----+
6 rows in set (0,00 sec)
```

Esta consulta amplía el desglose económico de cada producto mostrando, además del precio base y el IVA, el **valor equivalente a un 10% de descuento** sobre el precio con

impuestos incluidos.

El cálculo `(precio + precio*0.21)/10` permite obtener rápidamente cuánto se ahorraría el cliente si se aplicara un descuento del 10% sobre el total. Esta operación puede servir para mostrar precios promocionales o preparar estrategias de marketing, facilitando la comparación entre el precio original y posibles rebajas. Los alias hacen que los resultados sean claros y fáciles de entender para cualquier usuario.

Consulta SQL:

```
SELECT
    nombre AS 'Nombre del producto',
    descripcion AS 'Descripción',
    precio AS 'Precio',
    precio*0.21 AS 'Impuesto IVA 21%',
    precio + precio*0.21 AS 'Total',
    (precio + precio*0.21)/10 AS 'Descuento 10%'
FROM productos;
```

Resultado de la consulta en el terminal:

Nombre del producto	Descripción	Precio	Impuesto IVA 21%
Ratón	Ratón de ordenador	15.50	3.255
Teclado	Teclado de ordenador	20.50	4.305
Monitor	Monitor de ordenador	200.50	42.105
Curso de SQL	Curso de SQL paso a paso	300.50	63.105
Curso de PHP	Curso de PHP paso a paso	200.50	42.105
Curso de Python	Curso de Python paso a paso	400.50	84.105

Esta consulta ofrece un desglose completo del precio de cada producto, incluyendo:

- El **precio base** (`precio`)
- El **importe del IVA al 21%** (`precio * 0.21`)
- El **precio total con IVA incluido**
- El **valor del 10% de descuento** sobre ese total
- Y finalmente, el **precio final con el descuento ya aplicado**

Todo ello se presenta con nombres claros mediante alias, lo que permite interpretar los resultados con facilidad. Esta estructura es muy útil en contextos comerciales y promociones, ya que muestra de forma transparente cómo se calcula el precio final que pagará el cliente tras aplicar impuestos y descuentos.

Consulta SQL:

```
SELECT
nombre AS 'Nombre del producto',
descripcion AS 'Descripción',
precio AS 'Precio',
precio*0.21 AS 'Impuesto IVA 21%',
precio + precio*0.21 AS 'Total',
(precio + precio*0.21)/10 AS 'Descuento 10%',
(precio + precio*0.21) - (precio + precio*0.21)/10 AS 'Precio con el descuento'
FROM productos;
```

Resultado de la consulta en el terminal:

Nombre del producto	Descripción	Precio	Impuesto IVA 21%
Ratón	Ratón de ordenador	15.50	3.255
Teclado	Teclado de ordenador	20.50	4.305
Monitor	Monitor de ordenador	200.50	42.105
Curso de SQL	Curso de SQL paso a paso	300.50	63.105
Curso de PHP	Curso de PHP paso a paso	200.50	42.105
Curso de Python	Curso de Python paso a paso	400.50	84.105

6 rows in set (0,00 sec)

Esta consulta añade una columna adicional que construye un **código identificador del producto** combinando su nombre y su categoría. Esta concatenación se realiza con el operador `||`, que une los valores de texto para formar una cadena única, útil como referencia interna o externa.

Además de este "código de producto", la consulta presenta un desglose completo del precio: base, impuesto, total, descuento del 10% y precio final con descuento. Este enfoque es muy útil para generar informes comerciales personalizados, catálogos automatizados o presentaciones de producto donde se necesiten códigos de identificación y cálculos dinámicos en una misma consulta.

Consulta SQL:

```
SELECT
nombre || '' || categoria AS 'Código producto',
nombre AS 'Nombre del producto',
descripcion AS 'Descripción',
precio AS 'Precio',
precio*0.21 AS 'Impuesto IVA 21%',
precio + precio*0.21 AS 'Total',
(precio + precio*0.21)/10 AS 'Descuento 10%',
(precio + precio*0.21) - (precio + precio*0.21)/10 AS 'Precio con el descuento'
FROM productos;
```

Resultado de la consulta en el terminal:

Código producto	Nombre del producto	Descripción	Precio
1	Ratón	Ratón de ordenador	15.5
1	Teclado	Teclado de ordenador	20.5
1	Monitor	Monitor de ordenador	200.5
1	Curso de SQL	Curso de SQL paso a paso	300.5
1	Curso de PHP	Curso de PHP paso a paso	200.5
1	Curso de Python	Curso de Python paso a paso	400.5

6 rows in set, 8 warnings (0,00 sec)

3.6. Cláusulas

Las cláusulas en SQL son elementos fundamentales que permiten estructurar una consulta y definir cómo se comporta. Actúan como bloques que filtran, organizan, agrupan o limitan los datos devueltos por una instrucción SELECT. Cada cláusula cumple un propósito específico y puede combinarse con otras para formar consultas complejas y precisas.

Entre las cláusulas más comunes se encuentran WHERE, que filtra registros según condiciones; ORDER BY, que ordena los resultados por una o varias columnas; GROUP BY, que agrupa registros con valores comunes para aplicar funciones agregadas; y HAVING, que actúa como un filtro adicional sobre los grupos resultantes.

Estas cláusulas permiten que las consultas no solo recuperen datos, sino que lo hagan de forma lógica, estructurada y ajustada a criterios concretos. Gracias a ellas, SQL se convierte en una herramienta potente para explorar grandes volúmenes de información y obtener exactamente lo que se necesita.

--- En SQL, las **cláusulas** son componentes fundamentales que permiten **estructurar y afinar las consultas**, definiendo exactamente qué datos se desean recuperar, en qué orden y bajo qué condiciones. Aunque una consulta puede ser tan simple como SELECT

* FROM tabla;; las cláusulas permiten convertir esa instrucción básica en una herramienta poderosa y precisa para extraer información relevante.

A lo largo de este apartado, exploraremos algunas de las cláusulas más comunes y útiles en SQL:

- ORDER BY, para ordenar los resultados por uno o más campos, ya sea ascendente o descendente.
- GROUP BY, para agrupar registros que comparten un mismo valor y poder aplicar funciones agregadas sobre ellos.
- HAVING, para filtrar los resultados de una agrupación, permitiendo aplicar condiciones después del GROUP BY.
- WHERE, para filtrar filas específicas antes de agrupar o mostrar resultados.

Estas cláusulas se utilizan con frecuencia en combinación unas con otras, y son esenciales para el desarrollo de consultas avanzadas, la generación de informes y la toma de decisiones basada en datos.

En los próximos ejercicios, aprenderás a utilizar estas cláusulas de forma práctica y progresiva. --- Esta consulta utiliza la cláusula SELECT para recuperar únicamente la columna nombre de la tabla productos. Gracias a ella, podemos **extraer campos específicos** de una tabla sin necesidad de obtener todos los datos.

Es un ejemplo básico del uso de cláusulas en SQL: en este caso, SELECT y FROM trabajan juntas para definir **qué datos** se quieren obtener y **de dónde**. Esta estructura es la base de todas las consultas de selección y puede ampliarse fácilmente con otras cláusulas como WHERE, ORDER BY o GROUP BY para obtener resultados más refinados.

Consulta SQL:

```
SELECT  
nombre  
FROM productos;
```

Resultado de la consulta en el terminal:

```
+-----+  
| nombre |  
+-----+  
| Ratón  |  
| Teclado|  
| Monitor|  
| Curso de SQL|  
| Curso de PHP|  
| Curso de Python|  
+-----+  
6 rows in set (0,00 sec)
```

Esta consulta recupera todos los registros de la tabla productos y los ordena de forma

ascendente según el valor de la columna precio. La cláusula ORDER BY permite establecer el criterio de ordenación de los resultados, y el modificador ASC (de ascending) indica que el orden debe ir de menor a mayor.

El uso de ORDER BY es esencial cuando se desea presentar los datos de forma ordenada, ya sea para facilitar la lectura, comparar precios o preparar listados jerarquizados. Si no se especifica el orden, los resultados aparecerán según el orden físico en que están almacenados en la base de datos, lo cual no siempre es útil o intuitivo.

Consulta SQL:

```
SELECT
*
FROM productos
ORDER BY precio ASC;
```

Resultado de la consulta en el terminal:

Identificador	nombre	descripcion	precio	cat
1	Ratón	Ratón de ordenador	15.50	fis
2	Teclado	Teclado de ordenador	20.50	fis
3	Monitor	Monitor de ordenador	200.50	fis
5	Curso de PHP	Curso de PHP paso a paso	200.50	vir
4	Curso de SQL	Curso de SQL paso a paso	300.50	vir
6	Curso de Python	Curso de Python paso a paso	400.50	vir

6 rows in set (0,00 sec)

Esta consulta selecciona todos los registros de la tabla productos y los ordena según el valor de la columna precio. Aunque no se especifica explícitamente, al omitir la dirección del orden (ASC o DESC), SQL asume por defecto un **orden ascendente**.

Esta forma simplificada de ORDER BY es completamente válida y produce el mismo resultado que usar ORDER BY precio ASC;. Es útil cuando se busca listar los productos desde el más barato hasta el más caro, facilitando análisis de precios, decisiones comerciales o visualizaciones ordenadas para el usuario.

Consulta SQL:

```
SELECT
*
FROM productos
ORDER BY precio;
```

Resultado de la consulta en el terminal:

```
+-----+-----+-----+-----+
| Identificador | nombre      | descripcion          | precio | cat
+-----+-----+-----+-----+
|       1 | Ratón       | Ratón de ordenador   | 15.50  | fis
|       2 | Teclado     | Teclado de ordenador | 20.50  | fis
|       3 | Monitor     | Monitor de ordenador| 200.50 | fis
|       5 | Curso de PHP | Curso de PHP paso a paso | 200.50 | vir
|       4 | Curso de SQL | Curso de SQL paso a paso | 300.50 | vir
|       6 | Curso de Python | Curso de Python paso a paso | 400.50 | vir
+-----+-----+-----+-----+
6 rows in set (0,00 sec)
```

Esta consulta selecciona todos los registros de la tabla `productos` y los ordena por la columna `precio` en **orden descendente**, es decir, del más caro al más barato. La cláusula `ORDER BY` junto con el modificador `DESC` (descending) permite invertir el orden natural de los datos.

Este tipo de ordenación es útil cuando se quiere destacar los productos de mayor valor, analizar los artículos premium, o simplemente mostrar primero los productos más caros en un catálogo o informe. Es una herramienta clave para visualizar jerarquías dentro de los datos.

Consulta SQL:

```
SELECT
*
FROM productos
ORDER BY precio DESC;
```

Resultado de la consulta en el terminal:

```
+-----+-----+-----+-----+
| Identificador | nombre      | descripcion          | precio | cat
+-----+-----+-----+-----+
|       6 | Curso de Python | Curso de Python paso a paso | 400.50 | vir
|       4 | Curso de SQL    | Curso de SQL paso a paso | 300.50 | vir
|       3 | Monitor       | Monitor de ordenador   | 200.50 | fis
|       5 | Curso de PHP    | Curso de PHP paso a paso | 200.50 | vir
|       2 | Teclado        | Teclado de ordenador   | 20.50  | fis
|       1 | Ratón          | Ratón de ordenador     | 15.50  | fis
+-----+-----+-----+-----+
6 rows in set (0,00 sec)
```

Esta consulta selecciona la columna `categoria` de la tabla `productos` y agrupa los resultados por los distintos valores que puede tomar esa columna. La cláusula `GROUP BY` permite **agrupar registros que comparten un mismo valor** en uno o más campos,

devolviendo una sola fila por cada grupo distinto encontrado.

Aunque aquí no se aplica ninguna función agregada, esta forma de uso permite identificar todas las categorías existentes en la tabla sin repeticiones. Es útil como base para estadísticas, filtros dinámicos, generación de menús o simplemente para conocer las distintas clasificaciones presentes en los datos.

Consulta SQL:

```
SELECT  
categoria  
FROM productos  
GROUP BY categoria;
```

Resultado de la consulta en el terminal:

```
+-----+  
| categoria |  
+-----+  
| fisico   |  
| virtual  |  
+-----+  
2 rows in set (0,00 sec)
```

Esta consulta recupera todos los productos cuyo precio sea superior a 50. La cláusula WHERE actúa como un **filtro** que permite seleccionar únicamente aquellos registros que cumplen con una condición determinada, en este caso, un precio mayor que 50.

El uso de WHERE es esencial para extraer subconjuntos específicos de datos dentro de una tabla, lo que permite enfocarse en los productos relevantes para un análisis, informe o acción concreta. Esta cláusula suele combinarse con otras como ORDER BY o GROUP BY para obtener resultados aún más precisos.

Consulta SQL:

```
SELECT  
*  
FROM productos  
WHERE precio > 50;
```

Resultado de la consulta en el terminal:

Identificador	nombre	descripcion	precio	categoria
3	Monitor	Monitor de ordenador	200.50	fisico
4	Curso de SQL	Curso de SQL paso a paso	300.50	virtual
5	Curso de PHP	Curso de PHP paso a paso	200.50	virtual
6	Curso de Python	Curso de Python paso a paso	400.50	virtual

4 rows in set (0,00 sec)

Esta consulta agrupa los productos por categoría, calcula el **precio máximo** en cada grupo y luego filtra esos grupos utilizando la cláusula HAVING para mostrar solo aquellos cuya categoría incluye productos con un precio superior a 200.

A diferencia de WHERE, que filtra filas antes de la agrupación, HAVING permite **filtrar sobre el resultado de funciones agregadas** como MAX(), SUM(), AVG(), etc. En este caso, se utiliza un alias (precio_maximo) para facilitar la lectura y aplicar la condición.

Este tipo de consulta es ideal para análisis por categorías, como identificar los grupos de productos más costosos o detectar segmentos con artículos de alto valor.

Consulta SQL:

```
SELECT categoria, MAX(precio) AS precio_maximo
FROM productos
GROUP BY categoria
HAVING precio_maximo > 200;
```

Resultado de la consulta en el terminal:

categoria	precio_maximo
fisico	200.50
virtual	400.50

2 rows in set (0,00 sec)

3.7. Funciones

Las funciones en SQL son herramientas que permiten realizar cálculos, transformaciones y operaciones sobre los datos almacenados en las tablas. Pueden aplicarse tanto en consultas como en instrucciones de modificación, y devuelven un valor a partir de uno o más argumentos.

Existen distintos tipos de funciones, según su propósito:

- **Funciones de agregado:** resumen conjuntos de datos, como COUNT(), SUM(), AVG(), MIN() o MAX(), y suelen usarse junto a GROUP BY para obtener resultados agrupados.
- **Funciones matemáticas:** permiten realizar cálculos numéricos, como ROUND(), FLOOR(), CEIL(), ABS(), SQRT(), entre otras.
- **Funciones de cadena:** manipulan textos y permiten unir, cortar, convertir o reemplazar cadenas, como CONCAT(), SUBSTRING(), UPPER(), LOWER(), LENGTH() o REPLACE().
- **Funciones de fecha y hora:** extraen o formatean información temporal, como NOW(), CURDATE(), DATE_FORMAT() o DATEDIFF().
- **Funciones condicionales:** permiten tomar decisiones dentro de una consulta, como IF() o CASE.

El uso de funciones eleva la capacidad expresiva de SQL, facilitando la generación de informes, estadísticas, análisis dinámicos y transformaciones de datos directamente desde el lenguaje de consulta.

3.7.1. Agregado

Las funciones de agregado permiten realizar operaciones sobre un conjunto de registros y devolver un único valor como resultado. Son especialmente útiles cuando se desea obtener resúmenes o estadísticas de los datos almacenados en una tabla. Estas funciones no analizan fila por fila, sino que operan sobre bloques de datos que pueden estar agrupados mediante la cláusula GROUP BY.

Las funciones más comunes de este tipo incluyen:

- COUNT(): cuenta la cantidad de registros o valores.
- SUM(): suma todos los valores de una columna numérica.
- AVG(): calcula la media aritmética de los valores.
- MIN(): devuelve el valor mínimo de un conjunto.
- MAX(): devuelve el valor máximo de un conjunto.

Estas funciones se utilizan con frecuencia para generar informes y resúmenes, como calcular el total de ventas, el número de pedidos realizados, el precio promedio de los productos o el valor mínimo y máximo de una columna determinada. Son esenciales para transformar grandes cantidades de datos en información comprensible y útil.

Esta consulta agrupa los productos por su categoría y cuenta cuántos productos hay en cada una. La función agregada COUNT(categoría) devuelve el número de registros dentro de cada grupo, mientras que la cláusula GROUP BY organiza los datos según los valores de categoría.

Los alias asignados mediante AS permiten mostrar encabezados más legibles en el resultado: 'Categoria' y 'Número'. Esta consulta es útil para obtener estadísticas rápidas sobre la distribución de productos, como cuántos son físicos y cuántos virtuales, lo cual es muy valioso para informes de inventario o planificación comercial.

Consulta SQL:

```
SELECT
categoria AS 'Categoria',
COUNT(categoria) AS 'Número'
FROM productos
GROUP BY categoria;
```

Resultado de la consulta en el terminal:

```
+-----+-----+
| Categoria | Número |
+-----+-----+
| fisico   |      3 |
| virtual  |      3 |
+-----+-----+
2 rows in set (0,00 sec)
```

Esta consulta calcula la **suma total de los precios** de todos los productos agrupados por categoría. La función agregada **SUM(precio)** permite obtener el valor acumulado de los precios dentro de cada grupo definido por categoría.

Los resultados muestran, por cada categoría, la cantidad total correspondiente a la suma de los precios de sus productos. Esta información es especialmente útil para **evaluar el peso económico** de cada grupo de productos, por ejemplo, para conocer cuánto valen en total los productos físicos frente a los virtuales dentro del catálogo.

Consulta SQL:

```
SELECT
SUM(precio) AS 'Cantidad',
categoria AS 'Categoria'
FROM productos
GROUP BY categoria;
```

Resultado de la consulta en el terminal:

```
+-----+-----+
| Cantidad | Categoria |
+-----+-----+
|  236.50 | fisico   |
|  901.50 | virtual  |
+-----+-----+
2 rows in set (0,00 sec)
```

Esta consulta calcula el **precio promedio** de los productos dentro de cada categoría. La

función AVG(precio) devuelve el valor medio de todos los precios agrupados por categoría.

El resultado permite comparar el **precio promedio entre productos físicos y virtuales**, lo cual es útil para análisis de posicionamiento, estudios de mercado o decisiones sobre estrategias de precios. Gracias al uso de alias ('Cantidad' y 'Categoria'), los datos se presentan de forma clara y comprensible.

Consulta SQL:

```
SELECT
AVG(precio) AS 'Cantidad',
categoria AS 'Categoria'
FROM productos
GROUP BY categoria;
```

Resultado de la consulta en el terminal:

```
+-----+-----+
| Cantidad | Categoria |
+-----+-----+
| 78.833333 | fisico   |
| 300.500000 | virtual  |
+-----+-----+
2 rows in set (0,00 sec)
```

Esta consulta obtiene el **precio mínimo** entre los productos de cada categoría. La función MIN(precio) devuelve el valor más bajo dentro de cada grupo definido por categoría, permitiendo identificar el producto más barato en cada segmento.

Este tipo de análisis es útil para detectar puntos de entrada de precios, comparar niveles económicos por categoría o destacar los productos más asequibles en cada grupo. Los alias mejoran la legibilidad de los resultados, haciendo que puedan presentarse directamente en informes o aplicaciones.

Consulta SQL:

```
SELECT
MIN(precio) AS 'Cantidad',
categoria AS 'Categoria'
FROM productos
GROUP BY categoria;
```

Resultado de la consulta en el terminal:

```
+-----+-----+
| Cantidad | Categoria |
+-----+-----+
| 15.50 | fisico   |
| 200.50 | virtual  |
+-----+-----+
2 rows in set (0,00 sec)
```

Esta consulta calcula el **precio máximo** dentro de cada categoría de productos. Utiliza la función MAX(precio) para identificar el artículo más caro en cada grupo, agrupando los resultados por el campo categoria.

Este tipo de análisis es útil para conocer los límites superiores de precios dentro del catálogo, comparar el producto más costoso entre diferentes categorías o establecer rangos de precios en informes comerciales. Los alias asignados facilitan la interpretación directa de los resultados.

Consulta SQL:

```
SELECT
MAX(precio) AS 'Cantidad',
categoria AS 'Categoria'
FROM productos
GROUP BY categoria;
```

Resultado de la consulta en el terminal:

```
+-----+-----+
| Cantidad | Categoria |
+-----+-----+
| 200.50 | fisico   |
| 400.50 | virtual  |
+-----+-----+
2 rows in set (0,00 sec)
```

3.7.2. Matemáticas

Las funciones matemáticas en SQL permiten realizar cálculos numéricos directamente sobre los datos de las tablas. Estas funciones son útiles cuando se necesita transformar, redondear o analizar valores numéricos como precios, cantidades, medidas o totales.

Algunas de las funciones matemáticas más utilizadas son:

- ROUND(valor, decimales): redondea un número al número de decimales especificado.

- FLOOR(valor): redondea hacia abajo al entero más cercano.
- CEIL(valor) o CEILING(valor): redondea hacia arriba al entero más cercano.
- ABS(valor): devuelve el valor absoluto, eliminando el signo negativo si lo hubiera.
- SQRT(valor): calcula la raíz cuadrada del valor.
- RAND(): genera un número decimal aleatorio entre 0 y 1.

Estas funciones permiten aplicar lógica matemática directamente en las consultas, sin necesidad de cálculos adicionales desde el lado de la aplicación. Son ideales para aplicar reglas de negocio, generar valores derivados, calcular descuentos, márgenes, impuestos o cualquier operación matemática necesaria sobre los datos.

Esta consulta calcula el **promedio de los precios** por categoría y redondea el resultado al número entero más cercano usando la función ROUND(). La agrupación por categoría permite obtener un valor representativo para cada tipo de producto (físico o virtual, en este caso).

Este enfoque es útil cuando se desea simplificar la presentación de datos, omitiendo los decimales para facilitar la lectura en informes, gráficas o paneles visuales. Al mostrar valores enteros, se obtiene una visión general más clara y resumida del comportamiento de los precios en cada grupo.

Consulta SQL:

```
SELECT
ROUND(AVG(precio)) AS 'Cantidad',
categoria AS 'Categoria'
FROM productos
GROUP BY categoria;
```

Resultado de la consulta en el terminal:

```
SELECT
ROUND(AVG(precio)) AS 'Cantidad',
categoria AS 'Categoria'
FROM productos
GROUP BY categoria;
```

Esta consulta calcula el **promedio de los precios** por categoría y aplica la función FLOOR() para **redondear hacia abajo** el resultado al número entero más próximo. Es decir, elimina la parte decimal sin importar su valor.

El uso de FLOOR() es útil cuando se necesita trabajar con valores enteros más conservadores, por ejemplo, en presupuestos, cálculos de mínimos o cuando se desea evitar sobreestimar promedios en informes de costos. La agrupación por categoría permite obtener este valor para cada grupo de productos de forma independiente.

Consulta SQL:

```
SELECT
FLOOR(AVG(precio)) AS 'Cantidad',
categoria AS 'Categoria'
FROM productos
GROUP BY categoria;
```

Resultado de la consulta en el terminal:

```
+-----+-----+
| Cantidad | Categoria |
+-----+-----+
|      78 | fisico    |
|     300 | virtual   |
+-----+-----+
2 rows in set (0,00 sec)
```

Esta consulta calcula el **promedio del precio** por categoría y utiliza la función CEIL() para **redondear hacia arriba** el resultado al número entero más próximo. A diferencia de FLOOR(), que recorta la parte decimal, CEIL() siempre eleva el valor al siguiente entero si hay decimales.

Este enfoque puede ser útil en escenarios donde conviene **sobreestimar ligeramente** los precios promedio, como en la planificación de márgenes de ganancia o en proyecciones económicas conservadoras. Agrupar por categoría permite comparar fácilmente estos promedios redondeados entre diferentes tipos de productos.

Consulta SQL:

```
SELECT
CEIL(AVG(precio)) AS 'Cantidad',
categoria AS 'Categoria'
FROM productos
GROUP BY categoria;
```

Resultado de la consulta en el terminal:

```
+-----+-----+
| Cantidad | Categoria |
+-----+-----+
|      79 | fisico    |
|     301 | virtual   |
+-----+-----+
2 rows in set (0,00 sec)
```

Esta consulta calcula el **promedio del precio** por categoría y lo redondea a **dos decimales** utilizando la función ROUND() con un segundo parámetro igual a 2.

Esta precisión es especialmente útil cuando se trabaja con valores monetarios, ya que permite mostrar resultados consistentes con el formato habitual de precios. Agrupar por categoría facilita la comparación entre distintos tipos de productos y permite obtener un promedio más exacto y presentable para informes financieros o comerciales.

Consulta SQL:

```
SELECT
ROUND(AVG(precio),2) AS 'Cantidad',
categoria AS 'Categoria'
FROM productos
GROUP BY categoria;
```

Resultado de la consulta en el terminal:

```
+-----+-----+
| Cantidad | Categoria |
+-----+-----+
|    78.83 | fisico   |
|   300.50 | virtual  |
+-----+-----+
2 rows in set (0,00 sec)
```

Esta consulta calcula el **promedio del precio** por categoría y aplica la función ABS() para devolver su **valor absoluto**, es decir, sin signo negativo.

Aunque en contextos normales los precios no deberían ser negativos, esta función puede resultar útil si se trabaja con datos que podrían haber sido ingresados incorrectamente, o en casos donde se analicen diferencias o márgenes que pueden dar lugar a valores negativos. El uso de ABS() garantiza que el resultado se exprese siempre como un número positivo, facilitando la interpretación y evitando confusiones visuales en los informes.

Consulta SQL:

```
SELECT
ABS(AVG(precio)) AS 'Cantidad',
categoria AS 'Categoria'
FROM productos
GROUP BY categoria;
```

Resultado de la consulta en el terminal:

```
+-----+-----+
| Cantidad | Categoría |
+-----+-----+
| 78.833333 | físico   |
| 300.500000 | virtual  |
+-----+-----+
2 rows in set (0,00 sec)
```

Esta consulta selecciona el nombre y el precio de cada producto, junto con el resultado de aplicar la función SQRT(precio), que calcula la **raíz cuadrada** del valor del precio.

Aunque no es una operación común en análisis comerciales, SQRT() puede ser útil en contextos específicos como cálculos estadísticos, modelado matemático o algoritmos que requieren transformaciones numéricas. Incluir este tipo de funciones en consultas SQL permite realizar análisis más complejos directamente desde la base de datos, sin necesidad de procesar los datos externamente.

Consulta SQL:

```
SELECT
nombre,
precio,
SQRT(precio)
FROM productos;
```

Resultado de la consulta en el terminal:

```
+-----+-----+-----+
| nombre      | precio | SQRT(precio) |
+-----+-----+-----+
| Ratón       | 15.50  | 3.9370039370059056 |
| Teclado     | 20.50  | 4.527692569068709 |
| Monitor     | 200.50 | 14.159802258506296 |
| Curso de SQL | 300.50 | 17.334935823359714 |
| Curso de PHP  | 200.50 | 14.159802258506296 |
| Curso de Python | 400.50 | 20.0124960961895 |
+-----+-----+-----+
6 rows in set (0,00 sec)
```

Esta consulta muestra el nombre y el precio de cada producto, junto con un número aleatorio generado por la función RAND().

Cada vez que se ejecuta, RAND() produce un valor decimal aleatorio entre 0 y 1, diferente para cada fila del resultado. Esta función puede utilizarse para **asignar valores de prueba**, realizar **selecciones aleatorias**, simular sorteos o introducir variabilidad en pruebas de visualización o comportamiento de interfaces.

Consulta SQL:

```
SELECT
nombre,
precio,
RAND()
FROM productos;
```

Resultado de la consulta en el terminal:

```
+-----+-----+
| nombre | precio | RAND() |
+-----+-----+
| Ratón | 15.50 | 0.18317819869441743 |
| Teclado | 20.50 | 0.9189799622809328 |
| Monitor | 200.50 | 0.04536524605505657 |
| Curso de SQL | 300.50 | 0.4698863741661295 |
| Curso de PHP | 200.50 | 0.2133361633991228 |
| Curso de Python | 400.50 | 0.6570217252308705 |
+-----+-----+
6 rows in set (0,00 sec)
```

3.7.3. Cadenas

Las funciones de cadenas permiten manipular y transformar valores de texto en SQL. Son especialmente útiles cuando se trabaja con nombres, descripciones, identificadores o cualquier otro dato de tipo VARCHAR o TEXT, ya que permiten extraer partes de texto, combinarlos, convertir su formato o realizar búsquedas y sustituciones.

Algunas de las funciones de cadenas más comunes son:

- **CONCAT(campo1, campo2, ...)**: une varias cadenas en una sola.
- **SUBSTRING(cadena, inicio, longitud)**: extrae una subcadena desde una posición específica.
- **LOWER(cadena)**: convierte todos los caracteres a minúsculas.
- **UPPER(cadena)**: convierte todos los caracteres a mayúsculas.
- **LENGTH(cadena)**: devuelve la longitud de la cadena en caracteres.
- **REPLACE(cadena, texto_a_reemplazar, texto_nuevo)**: sustituye una parte del texto por otra.

Estas funciones resultan muy prácticas para generar campos personalizados, preparar datos para exportación, limpiar entradas de usuarios, crear identificadores automáticos, o construir búsquedas más flexibles en campos de texto. Permiten trabajar con la información de forma precisa y adaptada a distintas necesidades.

Esta consulta utiliza la función **CONCAT()** para **unir el contenido** de las columnas **idfiscal** y **nombre**, separándolos con un guion y un espacio (" - "). El resultado es una

cadena combinada que puede servir como **identificador descriptivo** del cliente.

Este tipo de formato es útil en interfaces de usuario, listados desplegables, facturas u otros contextos donde se desea mostrar de forma clara y concisa la relación entre el identificador fiscal y el nombre de la empresa. CONCAT() es una de las funciones más utilizadas para construir textos personalizados directamente desde la base de datos.

Consulta SQL:

```
SELECT  
CONCAT(idfiscal," - ",nombre)  
FROM clientes;
```

Resultado de la consulta en el terminal:

```
+-----+  
| CONCAT(idfiscal," - ",nombre) |  
+-----+  
| C000001 - Nombre de la empresa 1 |  
+-----+  
1 row in set (0,00 sec)
```

Esta consulta utiliza la función SUBSTRING() para extraer una parte del valor contenido en la columna idfiscal de cada cliente. En concreto, toma **6 caracteres** empezando desde la **segunda posición**.

Este tipo de operación es útil cuando se necesita **aislar una sección específica** de un campo de texto, como un código, sufijo o parte identificativa. Puede aplicarse en validaciones, generación de claves, anonimización parcial de datos o simplemente para mostrar una versión abreviada de un valor.

Consulta SQL:

```
SELECT  
SUBSTRING(idfiscal,2,6)  
FROM clientes;
```

Resultado de la consulta en el terminal:

```
+-----+  
| SUBSTRING(idfiscal,2,6) |  
+-----+  
| 000001 |  
+-----+  
1 row in set (0,00 sec)
```

Esta consulta transforma a **minúsculas** todos los valores de la columna nombre en la tabla productos, utilizando la función LOWER().

Este tipo de operación es útil cuando se quiere **normalizar el texto** para comparaciones, búsquedas insensibles a mayúsculas/minúsculas, o para mantener un estilo uniforme en visualizaciones, informes o interfaces. También resulta práctico como paso previo en tareas de limpieza y procesamiento de datos.

Consulta SQL:

```
SELECT
  LOWER(nombre)
FROM productos;
```

Resultado de la consulta en el terminal:

```
+-----+
| LOWER(nombre) |
+-----+
| ratón          |
| teclado        |
| monitor        |
| curso de sql   |
| curso de php   |
| curso de python|
+-----+
6 rows in set (0,00 sec)
```

Esta consulta convierte a **mayúsculas** todos los valores de la columna nombre de la tabla productos, utilizando la función UPPER().

Es útil cuando se desea estandarizar el formato del texto, facilitar búsquedas sin distinción de mayúsculas o simplemente destacar la información en la salida. Esta transformación también es común en informes, etiquetas o sistemas donde se prefiere un estilo uniforme en letras capitales.

Consulta SQL:

```
SELECT
  UPPER(nombre)
FROM productos;
```

Resultado de la consulta en el terminal:

```
+-----+  
| UPPER(nombre) |  
+-----+  
| RATÓN      |  
| TECLADO    |  
| MONITOR    |  
| CURSO DE SQL|  
| CURSO DE PHP|  
| CURSO DE PYTHON|  
+-----+  
6 rows in set (0,00 sec)
```

Esta consulta convierte a **mayúsculas** todos los valores de la columna nombre de la tabla productos, utilizando la función UPPER().

Es útil cuando se desea estandarizar el formato del texto, facilitar búsquedas sin distinción de mayúsculas o simplemente destacar la información en la salida. Esta transformación también es común en informes, etiquetas o sistemas donde se prefiere un estilo uniforme en letras capitales.

Consulta SQL:

```
SELECT  
LENGTH(nombre) AS 'longitud',nombre  
FROM productos;
```

Resultado de la consulta en el terminal:

```
+-----+-----+  
| longitud | nombre        |  
+-----+-----+  
|       6  | Ratón        |  
|       7  | Teclado     |  
|       7  | Monitor      |  
|      12  | Curso de SQL |  
|      12  | Curso de PHP |  
|      15  | Curso de Python|  
+-----+-----+  
6 rows in set (0,00 sec)
```

Esta consulta utiliza la función REPLACE() para **buscar y sustituir texto** dentro de la columna nombre de la tabla productos. En concreto, reemplaza la palabra "Curso" por "Super curso" en todos los registros donde aparezca.

El resultado es una nueva versión del nombre del producto, modificada dinámicamente. Este tipo de operación es útil para realizar **ajustes de formato**, aplicar

cambios de estilo o realizar transformaciones temporales para mostrar los datos con un enfoque comercial o promocional, sin alterar los valores almacenados en la base de datos.

Consulta SQL:

```
SELECT
REPLACE(nombre,"Curso","Super curso")
FROM productos;
```

Resultado de la consulta en el terminal:

```
+-----+
| REPLACE(nombre,"Curso","Super curso") |
+-----+
| Ratón
| Teclado
| Monitor
| Super curso de SQL
| Super curso de PHP
| Super curso de Python
+-----+
6 rows in set (0,00 sec)
```

3.7.4. Fechas

Las funciones de fechas permiten trabajar con valores temporales en SQL, como fechas y horas. Estas funciones son esenciales cuando se necesita registrar, comparar, calcular o formatear información relacionada con el tiempo: registros de creación, vencimientos, períodos de actividad, antigüedad, entre otros.

Entre las funciones de fechas más utilizadas se encuentran:

- **NOW()**: devuelve la fecha y hora actuales del sistema.
- **CURDATE()**: devuelve solo la fecha actual.
- **CURTIME()**: devuelve solo la hora actual.
- **DATE_FORMAT(fecha, formato)**: permite dar formato a una fecha según una plantilla específica.
- **DATEDIFF(fecha1, fecha2)**: calcula la diferencia en días entre dos fechas.

Estas funciones resultan muy útiles para informes periódicos, cálculos de antigüedad, control de plazos, seguimiento de actividades y generación de marcas temporales. También permiten adaptar el formato de las fechas para mostrarlas de forma más comprensible para el usuario final.

La función **NOW()** devuelve la **fecha y hora actual** del sistema en el que se ejecuta la base de datos, en formato YYYY-MM-DD HH:MM:SS.

Es especialmente útil cuando se desea registrar el momento exacto de una acción (como una inserción, actualización o auditoría), mostrar la hora actual en una aplicación, o comparar datos temporales. Esta función forma parte del conjunto de utilidades de fecha y hora que ofrece SQL para gestionar información temporal de forma precisa y flexible.

Consulta SQL:

```
SELECT NOW();
```

Resultado de la consulta en el terminal:

```
+-----+  
| NOW() |  
+-----+  
| 2025-04-07 18:42:39 |  
+-----+  
1 row in set (0,00 sec)
```

La función CURDATE() devuelve la **fecha actual** del sistema, sin incluir la hora, en formato YYYY-MM-DD.

Es útil cuando se necesita trabajar únicamente con la parte de la fecha, por ejemplo, para registrar el día de una operación, comparar fechas sin tener en cuenta la hora, o filtrar registros correspondientes a un día concreto. Al centrarse solo en la fecha, CURDATE() resulta ideal para tareas de calendario, informes diarios o validaciones por fecha.

Consulta SQL:

```
SELECT CURDATE();
```

Resultado de la consulta en el terminal:

```
+-----+  
| CURDATE() |  
+-----+  
| 2025-04-07 |  
+-----+  
1 row in set (0,00 sec)
```

La función CURTIME() devuelve la **hora actual del sistema**, sin incluir la fecha, en formato HH:MM:SS.

Es útil cuando se necesita trabajar exclusivamente con la parte horaria de una marca de

tiempo, por ejemplo, para registrar la hora de entrada o salida en un sistema, medir tiempos de respuesta, o programar eventos basados en la hora del día. Su precisión facilita operaciones temporales sin necesidad de manipular fechas completas.

Consulta SQL:

```
SELECT CURTIME();
```

Resultado de la consulta en el terminal:

```
+-----+
| CURTIME() |
+-----+
| 18:51:20 |
+-----+
1 row in set (0,00 sec)
```

Esta consulta utiliza la función DATE_FORMAT() para **dar formato personalizado** a la fecha y hora actual devuelta por NOW(). El formato especificado en el segundo parámetro produce una salida como la siguiente:

```
2025/04/07 -- 14::23::45
```

En este formato:

- %Y representa el año con cuatro cifras
- %m el mes con dos dígitos
- %d el día
- %H la hora en formato 24h
- %i los minutos
- %s los segundos

El uso de caracteres como -- y :: permite **personalizar completamente la visualización** de la fecha y la hora. Esta técnica es útil para generar marcas de tiempo legibles, exportaciones, nombres de archivo o reportes con una estructura específica.

Consulta SQL:

```
SELECT DATE_FORMAT(NOW(), '%Y/%m/%d -- %H::%i::%s');
```

Resultado de la consulta en el terminal:

```
+-----+  
| DATE_FORMAT(NOW(),'%Y/%m/%d -- %H::%i::%s') |  
+-----+  
| 2025/04/07 -- 18::51::33 |  
+-----+  
1 row in set (0,00 sec)
```

Esta consulta calcula la **diferencia en años aproximados** entre la fecha actual (CURDATE()) y una fecha pasada específica: '1978-04-14'.

La función DATEDIFF() devuelve el número de **días** entre dos fechas, y al dividir ese valor entre 365, se obtiene una estimación del tiempo transcurrido en **años**. El resultado será un número decimal que representa la antigüedad, edad o duración aproximada entre ambas fechas.

Este tipo de cálculo es útil para obtener edades, antigüedades, años de servicio u otras métricas temporales en sistemas que trabajan con fechas personales, laborales o históricas.

Consulta SQL:

```
SELECT DATEDIFF(CURDATE(),'1978-04-14')/365;
```

Resultado de la consulta en el terminal:

```
+-----+  
| DATEDIFF(CURDATE(),'1978-04-14')/365 |  
+-----+  
| 47.0137 |  
+-----+  
1 row in set (0,00 sec)
```

3.7.5. Condicional

Las funciones condicionales en SQL permiten tomar decisiones dentro de una consulta, evaluando condiciones y devolviendo diferentes resultados según se cumplan o no. Son especialmente útiles cuando se desea clasificar datos, aplicar lógicas de negocio o personalizar los valores mostrados en función de ciertos criterios.

Las más representativas son:

- IF(condición, valor_si_verdadero, valor_si_falso): evalúa una condición y devuelve un valor u otro según el resultado.
- CASE: permite evaluar múltiples condiciones de forma secuencial, similar a una estructura switch en otros lenguajes.

Estas funciones permiten, por ejemplo, etiquetar productos como "caros" o "baratos" según su precio, indicar si un envío debe ir en "paquete" o "caja" según su peso, o incluso calcular recargos o descuentos personalizados. Gracias a ellas, las consultas se vuelven más dinámicas y adaptables al contexto de los datos. Esta consulta añade una columna calculada que determina el **tipo de embalaje** necesario para cada producto, en función de su peso. La función IF() evalúa si el campo peso es menor o igual a 1:

- Si la condición se cumple, muestra 'Paquete'
- Si no, muestra 'Caja'

Este tipo de lógica condicional es muy útil para **automatizar decisiones dentro de las consultas**, como clasificaciones, etiquetas o recomendaciones. En este caso, permite deducir el tipo de envío más adecuado directamente desde los datos almacenados, sin necesidad de procesar la información fuera de la base de datos.

Consulta SQL:

```
SELECT
 *,
IF(peso <= 1,'Paquete','Caja') AS 'Embalaje de envío'
FROM productos;
```

Resultado de la consulta en el terminal:

Identificador	nombre	descripción	precio	cat
1 Ratón	Ratón de ordenador	15.50	fis	
2 Teclado	Teclado de ordenador	20.50	fis	
3 Monitor	Monitor de ordenador	200.50	fis	
4 Curso de SQL	Curso de SQL paso a paso	300.50	vir	
5 Curso de PHP	Curso de PHP paso a paso	200.50	vir	
6 Curso de Python	Curso de Python paso a paso	400.50	vir	

6 rows in set (0,00 sec)

Esta consulta utiliza una estructura IF anidada para determinar de forma más detallada el tipo de embalaje según el peso del producto:

- Si el peso es menor o igual a 1:
- Y si además es mayor que 0 → se muestra 'Paquete' - Si no (es decir, si el peso es 0) → se muestra 'No aplicable'
- Si el peso es mayor que 1 → se muestra 'Caja'

Este enfoque permite establecer **tres posibles salidas** según el valor del peso, lo cual es útil cuando se necesita contemplar **casos especiales**, como productos virtuales con peso cero. Así, se puede asignar de forma lógica el tipo de envío o indicar que el envío

no aplica.

Consulta SQL:

```
SELECT
*,
IF(peso <= 1,
    IF(peso > 0,'Paquete','No aplicable')
    , 'Caja') AS 'Embalaje de envío'
FROM productos;
```

Resultado de la consulta en el terminal:

Identificador	nombre	descripcion	precio	cat
1	Ratón	Ratón de ordenador	15.50	fis
2	Teclado	Teclado de ordenador	20.50	fis
3	Monitor	Monitor de ordenador	200.50	fis
4	Curso de SQL	Curso de SQL paso a paso	300.50	vir
5	Curso de PHP	Curso de PHP paso a paso	200.50	vir
6	Curso de Python	Curso de Python paso a paso	400.50	vir

6 rows in set (0,00 sec)

Esta consulta utiliza la estructura CASE para clasificar los productos según su precio. En concreto:

- Si el precio es menor o igual a 40, el producto se etiqueta como 'Barato'.
- En cualquier otro caso, se etiqueta como 'Caro'.

La cláusula CASE permite construir **condiciones más complejas y legibles** que un simple IF, especialmente cuando hay múltiples escenarios. Es ideal para asignar categorías, etiquetas, estados o cualquier tipo de clasificación basada en valores específicos de los datos. En este caso, se crea una columna adicional que facilita el análisis y la segmentación del catálogo de productos.

Consulta SQL:

```
SELECT
*,
CASE
    WHEN precio <= 40 THEN 'Barato'
    ELSE 'Caro'
END AS 'Tipo de producto'
FROM productos;
```

Resultado de la consulta en el terminal:

Identificador	nombre	descripcion	precio	cat
1 Ratón	Ratón de ordenador	15.50	fis	
2 Teclado	Teclado de ordenador	20.50	fis	
3 Monitor	Monitor de ordenador	200.50	fis	
4 Curso de SQL	Curso de SQL paso a paso	300.50	vir	
5 Curso de PHP	Curso de PHP paso a paso	200.50	vir	
6 Curso de Python	Curso de Python paso a paso	400.50	vir	

6 rows in set (0,00 sec)

3.8. Claves foráneas

Las claves foráneas son un tipo de restricción en SQL que permiten establecer relaciones entre tablas. Sirven para mantener la integridad referencial, es decir, aseguran que los datos relacionados entre diferentes tablas estén siempre conectados de forma válida y coherente.

Una clave foránea (FOREIGN KEY) es un campo (o conjunto de campos) en una tabla que hace referencia a la clave primaria (PRIMARY KEY) de otra tabla. Esto garantiza que el valor que se inserte en la columna foránea exista previamente en la tabla referenciada.

Por ejemplo, si una tabla pedidos tiene un campo clientes_id que apunta a la tabla clientes, la clave foránea impide que se cree un pedido para un cliente que no existe, o que se elimine un cliente si todavía tiene pedidos activos (a menos que se especifique una acción especial, como ON DELETE CASCADE).

Este mecanismo es fundamental en bases de datos relacionales, ya que permite modelar relaciones como uno-a-muchos o muchos-a-muchos de forma estructurada, evitando errores y manteniendo la coherencia de la información.

Esta instrucción crea una tabla llamada pedidos, diseñada para almacenar información sobre los pedidos realizados por los clientes. Incluye los siguientes campos:

- Identificador: clave primaria con incremento automático, que garantiza que cada pedido tenga un identificador único.
- fecha: almacena la fecha en la que se realiza el pedido.
- clientes_nombre: campo que representa la relación con la tabla clientes.

La clave foránea (FOREIGN KEY) establece una **relación entre el pedido y un cliente existente**, garantizando la integridad referencial: no se puede registrar un pedido si no existe un cliente con el identificador correspondiente.

El uso del motor InnoDB permite aprovechar funcionalidades como transacciones y claves foráneas, esenciales para mantener la coherencia en sistemas relacionales. Esta tabla forma la base para construir una estructura más compleja de gestión de pedidos.

Consulta SQL:

```
CREATE TABLE pedidos
(
    Identificador INT(10) NOT NULL AUTO_INCREMENT ,
    fecha DATE NOT NULL ,
    clientes_nombre INT(10) NOT NULL ,
    FOREIGN KEY (clientes_nombre) REFERENCES clientes(Identificador),
    PRIMARY KEY (`Identificador`)
) ENGINE = InnoDB;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 2 warnings (0,01 sec)
```

Consulta SQL:

```
INSERT INTO pedidos
VALUES(
    NULL,
    '2023-12-12',
    2
);
```

Importante: El ID 2 en tu caso puede ser otro diferente

Resultado de la consulta en el terminal:

```
Query OK, 1 row affected (0,00 sec)
```

Consulta SQL:

```
CREATE TABLE lineaspedido
(
    Identificador INT(10) NOT NULL AUTO_INCREMENT ,
    pedidos_fecha INT(10) NOT NULL ,
    productos_nombre INT(10) NOT NULL ,
    cantidad INT(10) NOT NULL ,
    FOREIGN KEY (pedidos_fecha) REFERENCES pedidos(Identificador),
    FOREIGN KEY (productos_nombre) REFERENCES productos(Identificador),
    PRIMARY KEY (`Identificador`)
) ENGINE = InnoDB;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 4 warnings (0,03 sec)
```

Consulta SQL:

```
INSERT INTO `lineaspedido`(`Identificador`, `pedidos_fecha`, `productos_nombre  
INSERT INTO `lineaspedido`(`Identificador`, `pedidos_fecha`, `productos_nombre  
INSERT INTO `lineaspedido`(`Identificador`, `pedidos_fecha`, `productos_nombre
```

Resultado de la consulta en el terminal:

```
mysql> INSERT INTO `lineaspedido`(`Identificador`, `pedidos_fecha`, `producto  
Query OK, 1 row affected (0,01 sec)
```

```
mysql> INSERT INTO `lineaspedido`(`Identificador`, `pedidos_fecha`, `producto  
Query OK, 1 row affected (0,00 sec)
```

```
mysql> INSERT INTO `lineaspedido`(`Identificador`, `pedidos_fecha`, `producto  
Query OK, 1 row affected (0,00 sec)
```

Importante: El ID 2 en tu caso puede ser otro diferente

3.9. Peticiones JOIN

Las peticiones JOIN permiten combinar datos de varias tablas en una sola consulta, conectando registros relacionados mediante claves comunes. Son fundamentales en bases de datos relacionales, donde la información se distribuye entre múltiples tablas para evitar redundancia y mejorar la organización.

Los tipos de JOIN más comunes son:

- INNER JOIN: devuelve solo los registros que tienen coincidencias en ambas tablas.
- LEFT JOIN: devuelve todos los registros de la tabla izquierda, y los coincidentes de la tabla derecha (o NULL si no hay coincidencia).
- RIGHT JOIN: similar al anterior, pero conserva todos los registros de la tabla derecha.
- FULL JOIN (no soportado por todos los motores): combina los resultados de LEFT y RIGHT JOIN.
- CROSS JOIN: devuelve el producto cartesiano de ambas tablas, es decir, todas las combinaciones posibles.
- SELF JOIN: une una tabla consigo misma, útil para comparar o relacionar registros dentro de la misma tabla.

Estas operaciones permiten consultar información distribuida, como mostrar los pedidos con sus clientes, las líneas de pedido con sus productos, o listar alumnos y los

cursos en los que están matriculados. Gracias a los JOIN, SQL se convierte en un lenguaje relacional en toda su extensión.

Consulta SQL:

```
SELECT * FROM pedidos
LEFT JOIN clientes
ON pedidos.clientes_nombre = clientes.Identificador;
```

Resultado de la consulta en el terminal:

Identificador	fecha	clientes_nombre	Identificador	nombre
2	2023-12-12		2	Nombre de la

1 row in set (0,00 sec)

Consulta SQL:

```
SELECT
pedidos.Identificador AS 'Número de pedido',
pedidos.fecha AS 'Fecha del pedido',
clientes.nombre AS 'Nombre del cliente'
FROM pedidos
LEFT JOIN clientes
ON pedidos.clientes_nombre = clientes.Identificador;
```

Resultado de la consulta en el terminal:

Número de pedido	Fecha del pedido	Nombre del cliente
2	2023-12-12	Nombre de la empresa 1

1 row in set (0,01 sec)

Consulta SQL:

```
SELECT
pedidos.Identificador AS 'Número de pedido',
pedidos.fecha AS 'Fecha del pedido',
clientes.nombre AS 'Nombre del cliente',
productos.nombre AS 'Producto',
productos.precio AS 'Precio',
lineaspedido.cantidad AS 'Cantidad',
productos.precio*lineaspedido.cantidad AS 'Subtotal'
FROM pedidos
LEFT JOIN clientes
ON pedidos.clientes_nombre = clientes.Identificador
LEFT JOIN lineaspedido
ON lineaspedido.pedidos_fecha = pedidos.Identificador
LEFT JOIN productos
ON lineaspedido.productos_nombre = productos.Identificador;
```

Resultado de la consulta en el terminal:

Número de pedido	Fecha del pedido	Nombre del cliente	Producto	Precio
2	2023-12-12	Nombre de la empresa 1	Ratón	20.00
2	2023-12-12	Nombre de la empresa 1	Teclado	40.00
2	2023-12-12	Nombre de la empresa 1	Monitor	200.00

3 rows in set (0,00 sec)

Consulta SQL:

```
INSERT INTO `pedidos` (`Identificador`, `fecha`, `clientes_nombre`) VALUES (NULL, '2023-12-12', 'Nombre de la empresa 1');
```

Resultado de la consulta en el terminal:

```
INSERT INTO `pedidos` (`Identificador`, `fecha`, `clientes_nombre`) VALUES (NULL, '2023-12-12', 'Nombre de la empresa 1');
```

Consulta SQL:

```
INSERT INTO `lineaspedido` (`Identificador`, `pedidos_fecha`, `productos_nombre`, `cantidad`, `precio`)
VALUES (NULL, '2023-12-12', 'Nombre de la empresa 1', 1, 20.00);
INSERT INTO `lineaspedido` (`Identificador`, `pedidos_fecha`, `productos_nombre`, `cantidad`, `precio`)
VALUES (NULL, '2023-12-12', 'Nombre de la empresa 1', 1, 40.00);
INSERT INTO `lineaspedido` (`Identificador`, `pedidos_fecha`, `productos_nombre`, `cantidad`, `precio`)
VALUES (NULL, '2023-12-12', 'Nombre de la empresa 1', 1, 200.00);
```

Resultado de la consulta en el terminal:

```
mysql> INSERT INTO `lineaspedido`(`Identificador`, `pedidos_fecha`, `producto
Query OK, 1 row affected (0,01 sec)
```

```
mysql> INSERT INTO `lineaspedido`(`Identificador`, `pedidos_fecha`, `producto
Query OK, 1 row affected (0,00 sec)
```

```
mysql> INSERT INTO `lineaspedido`(`Identificador`, `pedidos_fecha`, `producto
Query OK, 1 row affected (0,00 sec)
```

Consulta SQL:

```
SELECT
pedidos.Identificador AS 'Número de pedido',
pedidos.fecha AS 'Fecha del pedido',
clientes.nombre AS 'Nombre del cliente',
productos.nombre AS 'Producto',
productos.precio AS 'Precio',
lineaspedido.cantidad AS 'Cantidad',
productos.precio*lineaspedido.cantidad AS 'Subtotal'
FROM pedidos
LEFT JOIN clientes
ON pedidos.clientes_nombre = clientes.Identificador
LEFT JOIN lineaspedido
ON lineaspedido.pedidos_fecha = pedidos.Identificador
LEFT JOIN productos
ON lineaspedido.productos_nombre = productos.Identificador;
```

Resultado de la consulta en el terminal:

Número de pedido	Fecha del pedido	Nombre del cliente	Producto
2	2023-12-12	Nombre de la empresa 1	Ratón
2	2023-12-12	Nombre de la empresa 1	Teclado
2	2023-12-12	Nombre de la empresa 1	Monitor
4	2023-12-22	Nombre de la empresa 1	Curso de SQL
4	2023-12-22	Nombre de la empresa 1	Curso de PHP
4	2023-12-22	Nombre de la empresa 1	Curso de Python

6 rows in set (0,00 sec)

Consulta SQL:

```
SELECT
    pedidos.Identificador AS 'Número de pedido',
    pedidos.fecha AS 'Fecha del pedido',
    clientes.nombre AS 'Nombre del cliente',
    SUM(productos.precio * lineaspedido.cantidad) AS 'Total'
FROM pedidos
LEFT JOIN clientes
    ON pedidos.clientes_nombre = clientes.Identificador
LEFT JOIN lineaspedido
    ON lineaspedido.pedidos_fecha = pedidos.Identificador
LEFT JOIN productos
    ON lineaspedido.productos_nombre = productos.Identificador
GROUP BY
    pedidos.Identificador,
    pedidos.fecha,
    clientes.nombre;
```

Resultado de la consulta en el terminal:

Número de pedido	Fecha del pedido	Nombre del cliente	Total
2	2023-12-12	Nombre de la empresa 1	288.00
4	2023-12-22	Nombre de la empresa 1	2003.50

2 rows in set (0,00 sec)

Consulta SQL:

```
SELECT
    pedidos.Identificador AS 'Número de pedido',
    pedidos.fecha AS 'Fecha del pedido',
    clientes.nombre AS 'Nombre del cliente',
    SUM(productos.precio * lineaspedido.cantidad) AS 'Total'
FROM pedidos
LEFT JOIN clientes
    ON pedidos.clientes_nombre = clientes.Identificador
LEFT JOIN lineaspedido
    ON lineaspedido.pedidos_fecha = pedidos.Identificador
LEFT JOIN productos
    ON lineaspedido.productos_nombre = productos.Identificador
GROUP BY
    pedidos.Identificador,
    pedidos.fecha,
    clientes.nombre;
```

Resultado de la consulta en el terminal:

Número de pedido	Fecha del pedido	Nombre del cliente	Total
2	2023-12-12	Nombre de la empresa 1	288.00
4	2023-12-22	Nombre de la empresa 1	2003.50

2 rows in set (0,00 sec)

3.10. Vistas SQL

Las vistas en SQL son objetos virtuales que representan el resultado de una consulta. Se comportan como si fueran tablas, pero no almacenan datos por sí mismas, sino que muestran dinámicamente los datos de una o varias tablas subyacentes cada vez que se consultan.

Una vista se crea con la instrucción CREATE VIEW y puede incluir filtrados, uniones (JOIN), funciones, alias y cualquier otra lógica que se aplicaría en una consulta SELECT. Son especialmente útiles para:

- Simplificar consultas complejas y reutilizarlas fácilmente.
- Ocultar la complejidad de la estructura de datos al usuario final.
- Restringir el acceso a ciertas columnas o registros.
- Preparar informes predefinidos para facilitar su uso.

Al actuar como una capa intermedia entre los datos y los usuarios, las vistas ayudan a mantener la claridad, seguridad y consistencia en el acceso a la información dentro de la base de datos.

Consulta SQL:

```
CREATE VIEW totales_pedidos AS
SELECT
    pedidos.Identificador AS 'Número de pedido',
    pedidos.fecha AS 'Fecha del pedido',
    clientes.nombre AS 'Nombre del cliente',
    SUM(productos.precio * lineaspedido.cantidad) AS 'Total'
FROM pedidos
LEFT JOIN clientes
    ON pedidos.clientes_nombre = clientes.Identificador
LEFT JOIN lineaspedido
    ON lineaspedido.pedidos_fecha = pedidos.Identificador
LEFT JOIN productos
    ON lineaspedido.productos_nombre = productos.Identificador
GROUP BY
    pedidos.Identificador,
    pedidos.fecha,
    clientes.nombre;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,00 sec)
```

Consulta SQL:

```
SELECT * FROM `totales_pedidos`
ORDER BY `Total` DESC;
```

Resultado de la consulta en el terminal:

Número de pedido	Fecha del pedido	Nombre del cliente	Total
4	2023-12-22	Nombre de la empresa 1	2003.50
2	2023-12-12	Nombre de la empresa 1	288.00

```
2 rows in set (0,00 sec)
```

Consulta SQL:

```
CREATE VIEW detalle_pedidos AS
SELECT
pedidos.Identificador AS 'Número de pedido',
pedidos.fecha AS 'Fecha del pedido',
clientes.nombre AS 'Nombre del cliente',
productos.nombre AS 'Producto',
productos.precio AS 'Precio',
lineaspedido.cantidad AS 'Cantidad',
productos.precio*lineaspedido.cantidad AS 'Subtotal'
FROM pedidos
LEFT JOIN clientes
ON pedidos.clientes_nombre = clientes.Identificador
LEFT JOIN lineaspedido
ON lineaspedido.pedidos_fecha = pedidos.Identificador
LEFT JOIN productos
ON lineaspedido.productos_nombre = productos.Identificador;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,01 sec)
```

3.11. Exportación e importación

La exportación e importación de datos en SQL permiten transferir información entre sistemas, realizar copias de seguridad o restaurar el contenido de una base de datos en otro entorno. Estas operaciones son esenciales tanto para la administración como para el mantenimiento de bases de datos.

La **exportación** consiste en volcar el contenido de una base de datos o de tablas concretas a un archivo externo, generalmente en formato SQL. Este archivo incluye las instrucciones necesarias para reconstruir la estructura y los datos. Una herramienta muy utilizada para esta tarea es mysqldump.

La **importación** es el proceso inverso: a partir de un archivo exportado previamente, se vuelcan los datos y estructuras de nuevo en un sistema de base de datos. Esto se realiza con herramientas como mysql o desde interfaces gráficas.

Estas acciones son fundamentales para realizar migraciones, sincronizar entornos de desarrollo y producción, automatizar procesos de respaldo y garantizar la recuperación ante errores o fallos del sistema.

3.11.1. Formatos de exportación

3.11.2. MySQL DUMP

Consulta SQL:

```
GRANT PROCESS ON *.* TO 'josevicente'@'localhost';  
mysqldump.exe -u josevicente -p empresa > empresa.sql
```

Resultado de la consulta en el terminal:

```
-
```

Consulta SQL:

```
mysqldump.exe -u empresa -p empresa > C:/copiasdeseguridad/20240201empresa.sql
```

Resultado de la consulta en el terminal:**Consulta SQL:**

```
mysql.exe -u empresa2 -p empresa2 < C:/copiasdeseguridad/20240130empresa.sql
```

Resultado de la consulta en el terminal:**Consulta SQL:**

```
mysqldump.exe -u josevicente -p --all-databases > C:/copiasdeseguridad/20240123
```

Resultado de la consulta en el terminal:

4. Tipos de datos

Los tipos de datos en SQL definen la naturaleza del valor que puede almacenar cada columna de una tabla. Elegir el tipo de dato adecuado es esencial para optimizar el rendimiento, asegurar la integridad de la información y facilitar las operaciones posteriores sobre los datos.

Se dividen en varias categorías principales:

- **Numéricos:** almacenan números enteros (INT, SMALLINT, BIGINT) o decimales (DECIMAL, FLOAT, DOUBLE), según la precisión y tamaño requerido.
- **Cadenas de texto:** permiten guardar caracteres y palabras. Los más comunes son CHAR (longitud fija) y VARCHAR (longitud variable), así como TEXT para bloques grandes de texto.
- **Fechas y horas:** representan valores temporales como DATE, TIME, DATETIME, TIMESTAMP y permiten registrar eventos en el tiempo.
- **Booleanos:** almacenan valores de verdadero o falso, típicamente representados con TINYINT(1) en algunos motores como MySQL.
- **Especiales:** incluyen tipos como ENUM para listas cerradas de valores, SET para múltiples selecciones, y BLOB o MEDIUMBLOB para almacenar datos binarios como imágenes o archivos.
- **JSON:** algunos sistemas permiten guardar estructuras en formato JSON, facilitando el almacenamiento de datos semiestructurados.

La correcta asignación de tipos de datos garantiza un almacenamiento eficiente, reduce errores y contribuye a una base de datos bien diseñada y más fácil de mantener.

4.1. Numéricos

Los tipos de datos numéricos en SQL permiten almacenar valores numéricos, ya sean enteros o con decimales. Son fundamentales para trabajar con cantidades, precios, identificadores, porcentajes, estadísticas y cualquier valor que implique cálculo o comparación.

Se dividen en dos grandes grupos:

1. Números enteros Almacenan valores sin decimales. Se diferencian por el rango de valores que pueden representar:

- TINYINT: de -128 a 127 (o de 0 a 255 sin signo)
- SMALLINT: de -32,768 a 32,767
- MEDIUMINT: de -8.388.608 a 8.388.607
- INT o INTEGER: de -2.147.483.648 a 2.147.483.647
- BIGINT: para números enteros extremadamente grandes

2. Números decimales Permiten almacenar valores con parte decimal, ideales para precios, medias, porcentajes, etc.

- DECIMAL(m,d) o NUMERIC(m,d): precisión exacta con m dígitos en total y d decimales.

Muy usado en finanzas.

- FLOAT: punto flotante, menor precisión pero más eficiente.
- DOUBLE: como FLOAT, pero con mayor capacidad de precisión.

Elegir el tipo numérico correcto depende de la naturaleza del dato, el rango esperado y la precisión necesaria, buscando siempre el equilibrio entre exactitud y eficiencia de almacenamiento.

4.1.1. Enteros

Los tipos de datos enteros en SQL se utilizan para almacenar números sin decimales. Son ideales para valores como identificadores, cantidades, edades, contadores, códigos, entre otros. Cada tipo ofrece un rango distinto según la cantidad de bytes que ocupa en memoria.

Los principales tipos enteros son:

- TINYINT: ocupa 1 byte. Rango: de -128 a 127 (o de 0 a 255 si es UNSIGNED).
- SMALLINT: ocupa 2 bytes. Rango: de -32,768 a 32,767.
- MEDIUMINT: ocupa 3 bytes. Rango: de -8.388.608 a 8.388.607.
- INT o INTEGER: ocupa 4 bytes. Rango: de -2.147.483.648 a 2.147.483.647.
- BIGINT: ocupa 8 bytes. Rango: de -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807.

Todos estos tipos pueden declararse como UNSIGNED para duplicar el rango positivo y no permitir valores negativos. La elección del tipo debe hacerse en función del espacio necesario y del tamaño máximo esperado del dato, optimizando así el uso de memoria y el rendimiento de la base de datos.

4.1.2. Decimales

Los tipos de datos decimales en SQL se utilizan para representar números con parte fraccionaria, es decir, con decimales. Son esenciales en contextos donde se requiere precisión numérica, como cálculos financieros, precios, porcentajes o mediciones.

Los más utilizados son:

- DECIMAL(m,d) o NUMERIC(m,d): permiten definir con exactitud el número total de dígitos (m) y cuántos de ellos están después del punto decimal (d). Por ejemplo, DECIMAL(10,2) permite hasta 10 dígitos en total, con 2 decimales (es decir, hasta 99999999.99). Son ideales para operaciones que no toleran errores de redondeo.
- FLOAT: representa un número en punto flotante con precisión aproximada. Consumo menos espacio que DECIMAL pero puede perder exactitud en los cálculos, por lo que se usa en contextos donde esa precisión no es crítica.
- DOUBLE o DOUBLE PRECISION: similar a FLOAT, pero con mayor precisión y uso de más espacio en memoria. Se utiliza para cálculos científicos o estadísticos que requieren manejar valores más amplios o más precisos.

La elección entre DECIMAL y tipos de punto flotante (FLOAT, DOUBLE) depende del tipo de aplicación: para datos financieros, se recomienda DECIMAL; para cálculos complejos o científicos donde la precisión exacta no sea vital, se puede optar por FLOAT o DOUBLE.

4.1.3. Otros

4.2. Fechas

Los tipos de datos de fecha en SQL permiten almacenar y manipular información temporal, como fechas, horas o combinaciones de ambas. Son fundamentales para registrar eventos, controlar plazos, generar informes periódicos o calcular diferencias de tiempo.

Los principales tipos de datos de fecha son:

- DATE: almacena solo la fecha en formato YYYY-MM-DD. Por ejemplo, '2025-04-07'.
- TIME: almacena únicamente la hora en formato HH:MM:SS. Por ejemplo, '14:30:00'.
- DATETIME: combina fecha y hora en un solo valor. Formato: YYYY-MM-DD HH:MM:SS. Por ejemplo, '2025-04-07 14:30:00'.
- TIMESTAMP: similar a DATETIME, pero se actualiza automáticamente con la hora del sistema si se configura como valor por defecto o en actualización. Se usa frecuentemente para registrar marcas de tiempo (logs, auditorías).
- YEAR: almacena solo el año en formato YYYY. Por ejemplo, '2025'.

Estos tipos permiten realizar operaciones como comparar fechas, calcular diferencias de días, extraer partes específicas (año, mes, día, hora...), o dar formato personalizado a la salida. Elegir el tipo adecuado garantiza un correcto manejo del tiempo dentro de la base de datos.

4.3. Cadena

Los tipos de datos de cadena en SQL se utilizan para almacenar texto, como nombres, descripciones, direcciones, correos electrónicos o cualquier otro dato alfanumérico. Pueden tener longitud fija o variable, según el tipo elegido.

Los más comunes son:

- CHAR(n): almacena cadenas de longitud fija. Si el valor tiene menos caracteres, se rellena con espacios hasta alcanzar n. Es eficiente cuando todos los valores tienen la misma longitud, como códigos postales o códigos de país.
- VARCHAR(n): almacena cadenas de longitud variable hasta un máximo de n caracteres. Solo utiliza el espacio necesario para los datos, lo que lo hace más flexible y eficiente para la mayoría de los textos.
- TEXT: permite almacenar grandes cantidades de texto (hasta 65.535 caracteres). Es ideal para campos como descripciones extensas, comentarios o artículos, aunque tiene algunas limitaciones en comparación con VARCHAR, como no poder usarse directamente en índices.

- TINYTEXT, MEDIUMTEXT, LONGTEXT: variantes de TEXT que permiten almacenar volúmenes aún mayores de texto, desde 255 caracteres hasta varios gigabytes, según el tipo.

La elección del tipo de cadena debe hacerse en función del tamaño esperado del contenido y del uso que se le dará, equilibrando eficiencia, flexibilidad y compatibilidad con otras funciones o restricciones del sistema.

4.4. JSON

El tipo de dato JSON en SQL permite almacenar información estructurada en formato JSON (JavaScript Object Notation), lo que facilita el manejo de datos semiestructurados dentro de una base de datos relacional. Este tipo es útil cuando se requiere flexibilidad en la estructura de los datos o cuando se integran sistemas que intercambian información en este formato.

Este tipo de dato permite guardar objetos y arreglos JSON directamente en una columna, como por ejemplo:

```
{  
    "nombre": "Juan",  
    "edad": 30,  
    "direcciones": ["Valencia", "Madrid"]  
}
```

Algunas bases de datos, como MySQL y PostgreSQL, permiten:

- Validar que el contenido insertado tenga un formato JSON válido.
- Consultar y extraer partes del contenido usando funciones como JSON_EXTRACT(), ->, ->> o JSON_VALUE().
- Modificar datos específicos dentro del JSON con funciones como JSON_SET() o JSON_REPLACE().

El uso de JSON es especialmente útil cuando se necesita almacenar información dinámica o anidada sin necesidad de crear múltiples tablas relacionadas. Aun así, debe utilizarse con criterio, ya que puede dificultar ciertas operaciones relacionales y afectar al rendimiento si se abusa de él en lugar de un diseño estructurado tradicional.

5. Motores de almacenamiento

Los motores de almacenamiento en SQL definen cómo se guardan físicamente los datos en las tablas y cómo se gestionan aspectos como el acceso, el bloqueo, las transacciones o la integridad. Cada motor tiene sus propias características, ventajas y limitaciones, por lo que elegir el adecuado es clave para el rendimiento y comportamiento de la base de datos.

Algunos de los motores más utilizados son:

- **InnoDB:** es el motor por defecto en MySQL. Soporta transacciones, claves foráneas, bloqueo a nivel de fila e integridad referencial. Es adecuado para aplicaciones críticas donde la consistencia de los datos es prioritaria.
- **MyISAM:** motor más antiguo y sencillo. Ofrece alto rendimiento en lectura, pero no soporta transacciones ni claves foráneas. Se usa en aplicaciones donde se prioriza la velocidad de consulta sobre la integridad relacional.
- **Merge (MRG_MyISAM):** permite combinar varias tablas MyISAM con estructuras idénticas para que actúen como una sola. Es útil para gestionar grandes volúmenes de datos distribuidos.
- **CSV:** almacena los datos en archivos de texto separados por comas. No admite índices ni claves, pero permite importar y exportar fácilmente desde hojas de cálculo u otros sistemas.
- **Memory:** guarda los datos en memoria RAM, lo que proporciona gran velocidad, pero los datos se pierden al reiniciar el servidor. Ideal para tablas temporales o de trabajo intensivo.
- **Aria:** evolución de MyISAM que incorpora recuperación ante fallos. Se puede usar como alternativa más robusta sin llegar a la complejidad de InnoDB.
- **Sequence:** permite generar secuencias automáticas de valores, útil para crear identificadores únicos.

Cada motor responde mejor a diferentes necesidades: unos ofrecen velocidad, otros fiabilidad, otros flexibilidad. Por ello, es posible combinar distintos motores dentro de una misma base de datos según los requisitos de cada tabla.

5.1. InnoDB

InnoDB es el motor de almacenamiento por defecto en MySQL y uno de los más potentes y completos disponibles. Está diseñado para garantizar la integridad de los datos, ofrecer un alto rendimiento en entornos transaccionales y permitir operaciones concurrentes de forma segura.

Entre sus características más destacadas se encuentran:

- **Soporte para transacciones:** permite agrupar varias operaciones en una sola unidad lógica, utilizando START TRANSACTION, COMMIT y ROLLBACK, lo cual es fundamental para mantener la coherencia de los datos.
- **Integridad referencial:** admite claves foráneas (FOREIGN KEY) para definir relaciones entre tablas y garantizar que los datos se mantengan conectados de forma válida.
- **Bloqueo a nivel de fila:** mejora el rendimiento en sistemas multiusuario al permitir que varias operaciones ocurran simultáneamente sin conflictos.
- **Recuperación ante fallos:** registra las operaciones en un log interno para poder recuperar el estado anterior en caso de caída del sistema.
- **Soporte para restricciones y validaciones:** como claves primarias, únicas y comprobaciones (NOT NULL, CHECK, etc.).

InnoDB es ideal para aplicaciones que requieren fiabilidad, seguridad y consistencia, como sistemas financieros, plataformas de comercio electrónico, CRMs y cualquier entorno donde se trabaje con datos críticos.

5.2. MyISAM

MyISAM es uno de los motores de almacenamiento clásicos de MySQL, conocido por su simplicidad y su alto rendimiento en operaciones de lectura. Aunque ha sido reemplazado por InnoDB como motor por defecto, sigue siendo útil en ciertos escenarios donde la velocidad es más importante que la integridad transaccional.

Sus principales características son:

- **Alto rendimiento en consultas:** ideal para aplicaciones orientadas a lectura intensiva, como sistemas de informes o catálogos.
- **No soporta transacciones:** a diferencia de InnoDB, no permite agrupar operaciones en bloques atómicos con COMMIT o ROLLBACK.
- **No admite claves foráneas:** no puede establecer relaciones directas entre tablas mediante restricciones de integridad referencial.
- **Bloqueo a nivel de tabla:** al modificar datos, bloquea la tabla completa, lo que puede afectar el rendimiento en entornos con muchas escrituras simultáneas.
- **Ficheros independientes:** cada tabla genera tres archivos: uno para la estructura (.frm), uno para los datos (.MYD) y otro para los índices (.MYI).

MyISAM es adecuado para sistemas donde se realizan muchas consultas de lectura y pocas escrituras, y donde la integridad relacional no es crítica. Sin embargo, para aplicaciones modernas con múltiples usuarios y necesidad de consistencia, InnoDB suele ser la mejor opción.

5.3. Merge MyISAM

El motor **Merge MyISAM** (también conocido como MRG_MyISAM) permite combinar varias tablas MyISAM con estructuras idénticas en una sola tabla lógica. Esta tabla merge actúa como un contenedor virtual que unifica el acceso a las tablas subyacentes, facilitando consultas sobre grandes volúmenes de datos distribuidos.

Características principales:

- **Unión lógica:** todas las tablas unidas deben tener exactamente la misma estructura (mismos nombres y tipos de columnas).
- **Solo para MyISAM:** solo puede trabajar con tablas que utilicen el motor MyISAM.
- **Solo lectura o solo escritura:** si la tabla merge está configurada para escritura, solo puede escribir en una de las tablas base; si es para lectura, puede consultar todas.
- **Alto rendimiento:** útil para dividir datos en varias tablas por motivos de rendimiento (por ejemplo, por años) y consultarlos como si fueran una sola.

Un caso de uso típico es cuando se desea organizar datos históricos en diferentes tablas (una por mes o por año), pero permitir consultas globales sin necesidad de realizar múltiples UNION.

Aunque es una solución eficiente para ciertos escenarios, no soporta claves foráneas ni transacciones, y su uso ha disminuido con la adopción de InnoDB y otras estrategias más flexibles.

5.4. CSV

El motor de almacenamiento **CSV** en SQL permite guardar los datos de una tabla directamente en archivos de texto plano, con valores separados por comas (Comma Separated Values). Cada tabla CSV corresponde a un archivo .csv, lo que facilita la exportación, importación y lectura desde otras aplicaciones, como hojas de cálculo o editores de texto.

Características destacadas:

- **Formato abierto y compatible:** los datos se almacenan en un formato estándar que puede ser leído fácilmente por otros programas, como Excel o Google Sheets.
- **Sin índices ni claves:** no admite índices, claves primarias o foráneas, ni restricciones de integridad. Solo almacena datos puros.
- **Lectura y escritura sencilla:** ideal para tareas de integración de datos o para compartir información entre sistemas.
- **No soporta transacciones:** como otros motores ligeros, no ofrece funcionalidades avanzadas como ROLLBACK o COMMIT.

El motor CSV es útil para tablas temporales, importación de datos externos, generación de reportes simples o como puente para intercambiar datos entre aplicaciones. No está pensado para uso en producción, sino como herramienta auxiliar en procesos de entrada y salida de información.

5.5. Memory

El motor de almacenamiento **Memory** en SQL (también conocido como HEAP) guarda los datos directamente en la memoria RAM del servidor, lo que proporciona una velocidad de acceso extremadamente alta. Está diseñado para operaciones temporales, cálculos intermedios o datos que no necesitan persistencia.

Características principales:

- **Alta velocidad:** al estar en memoria, las consultas son mucho más rápidas que en disco.
- **Datos volátiles:** toda la información se pierde al reiniciar el servidor o cerrar la sesión de la base de datos.
- **Estructura similar a MyISAM:** admite índices, pero no transacciones ni claves foráneas.
- **Uso limitado:** ideal para tablas de trabajo, datos intermedios o almacenamiento temporal durante una sesión de usuario.

Este motor es muy útil cuando se requiere procesar datos de forma intensiva durante un corto periodo de tiempo, como en operaciones analíticas, generación de informes temporales o cachés en tiempo real. Sin embargo, no debe utilizarse para almacenar información crítica o persistente.

5.6. Aria

Aria es un motor de almacenamiento diseñado como sucesor mejorado de MyISAM, desarrollado por los creadores originales de MySQL. Está pensado para ser rápido, seguro y tolerante a fallos, manteniendo compatibilidad con muchas características de MyISAM, pero añadiendo capacidades adicionales.

Características principales:

- **Soporte de recuperación ante caídas:** a diferencia de MyISAM, Aria puede recuperarse de apagados inesperados o fallos del sistema gracias a su sistema de logs.
- **Compatible con MyISAM:** permite abrir y leer tablas MyISAM, facilitando la migración.
- **Alto rendimiento en lectura:** optimizado para consultas rápidas, especialmente en operaciones de solo lectura o con bajo nivel de escritura concurrente.
- **Sin soporte de transacciones completas:** aunque se ha diseñado con vistas a ello, aún no cuenta con soporte transaccional como InnoDB.
- **Multiplataforma y ligero:** ideal para sistemas embebidos o de recursos limitados donde se necesita fiabilidad sin demasiada complejidad.

Aria es una buena alternativa a MyISAM cuando se busca rendimiento con mayor seguridad frente a fallos, aunque no alcanza las funcionalidades avanzadas de InnoDB en cuanto a integridad referencial o transacciones.

5.7. Sequence

El tipo de motor **Sequence** en SQL, aunque no es común en todos los sistemas de bases de datos, está diseñado para generar secuencias automáticas de valores numéricos. Su objetivo principal es facilitar la creación de identificadores únicos o series numéricas sin necesidad de depender de funciones externas o contadores manuales.

Características principales:

- **Generación automática:** produce valores secuenciales de forma controlada, útiles para claves primarias o campos con incremento automático.
- **Configuración personalizada:** permite definir el valor inicial, el incremento entre números, y en algunos casos, límites máximo y mínimo.
- **Independiente de las tablas:** en algunos motores, las secuencias pueden existir como objetos separados de las tablas, reutilizables en varios contextos.

Aunque en MySQL el concepto de secuencia se suele manejar con AUTO_INCREMENT, en otros motores como PostgreSQL o MariaDB, las secuencias (SEQUENCE) son objetos más flexibles y potentes.

Este motor es útil cuando se necesita un control más detallado sobre la generación de identificadores, especialmente en entornos distribuidos o cuando múltiples tablas deben compartir una misma numeración.

6. Operaciones

6.1. Importar provincias

Consulta SQL:

```
CREATE TABLE IF NOT EXISTS `empresa`.`provincias` (`codigo` varchar(2), `provin
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 2 warnings (0,01 sec)
```

Consulta SQL:

```
INSERT INTO `empresa`.`provincias` (`codigo`, `provincia`) VALUES ('02', 'Albac
```

Resultado de la consulta en el terminal:

```
Query OK, 44 rows affected (0,01 sec)
Records: 44  Duplicates: 0  Warnings: 0
```

Consulta SQL:

```
ALTER TABLE `provincias` ADD `Identificador` INT(255) NOT NULL AUTO_INCREMENT F
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 1 warning (0,02 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

6.2. Codigos postales

Consulta SQL:

```
CREATE TABLE IF NOT EXISTS `empresa`.`codigospostales` (`codigopostal` int(5),
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 4 warnings (0,00 sec)
```

Consulta SQL:

```
INSERT INTO `empresa`.`codigospostales` (`codigopostal`, `idmunicipio`, `nombre`
```

Resultado de la consulta en el terminal:

```
Query OK, 27 rows affected (0,00 sec)
Records: 27  Duplicates: 0  Warnings: 0
```

Consulta SQL:

```
ALTER TABLE `codigospostales` ADD `Identificador` INT(255) NOT NULL AUTO_INCREMENT
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 1 warning (0,04 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

Consulta SQL:

```
ALTER TABLE `codigospostales` CHANGE `codigopostal` `codigopostal` VARCHAR(5) NOT NULL
```

Resultado de la consulta en el terminal:

```
Query OK, 27 rows affected (0,05 sec)
Records: 27  Duplicates: 0  Warnings: 0
```

Consulta SQL:

```
UPDATE codigospostales
SET codigopostal = LPAD(codigopostal,5,'0');
```

Resultado de la consulta en el terminal:

Rows matched: 27 Changed: 27 Warnings: 0

Consulta SQL:

```
ALTER TABLE `codigospostales` ADD `idprovincia` VARCHAR(255) NOT NULL AFTER `Ic
```

Resultado de la consulta en el terminal:

Query OK, 0 rows affected (0,02 sec)
Records: 0 Duplicates: 0 Warnings: 0

Consulta SQL:

```
UPDATE
codigospostales
SET idprovincia = LEFT(codigopostal,2);
```

Resultado de la consulta en el terminal:

Query OK, 27 rows affected (0,00 sec)
Rows matched: 27 Changed: 27 Warnings: 0

Consulta SQL:

```
SELECT
clientes.nombre,
clientes.idfiscal,
clientes.direccion,
clientes.nombrepersonacontacto,
clientes.emailpersonacontacto,
codigospostales.nombremunicipio,
codigospostales.codigopostal,
provincias.provincia

FROM `clientes`

LEFT JOIN codigospostales
ON clientes.codigopostal = codigospostales.codigopostal

LEFT JOIN provincias
ON codigospostales.idprovincia = provincias.codigo;
```

Resultado de la consulta en el terminal:

```
+-----+-----+-----+
| nombre | idfiscal | direccion | nombrepersona
+-----+-----+-----+
| Nombre de la empresa 1 | C000001 | Dirección de la empresa 1 | Juan Lopez
+-----+-----+-----+
1 row in set (0,00 sec)
```

Consulta SQL:

```
CREATE VIEW listado_clientes AS
SELECT
    clientes.nombre,
    clientes.idfiscal,
    clientes.direccion,
    clientes.nombrepersonacontacto,
    clientes.emailpersonacontacto,
    codigospostales.nombremunicipio,
    codigospostales.codigopostal,
    provincias.provincia

FROM `clientes`

LEFT JOIN codigospostales
ON clientes.codigopostal = codigospostales.codigopostal

LEFT JOIN provincias
ON codigospostales.idprovincia = provincias.codigo;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,01 sec)
```

6.3. Creación de impuestos

Consulta SQL:

```
CREATE TABLE `empresa`.`impuestos` (`Identificador` INT(255) NOT NULL AUTO_INCREMENT,
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 2 warnings (0,02 sec)
```

Consulta SQL:

```
INSERT INTO `impuestos`(`Identificador`, `nombre`, `tipoimpositivo`, `descripci  
INSERT INTO `impuestos`(`Identificador`, `nombre`, `tipoimpositivo`, `descripci  
INSERT INTO `impuestos`(`Identificador`, `nombre`, `tipoimpositivo`, `descripci  
INSERT INTO `impuestos`(`Identificador`, `nombre`, `tipoimpositivo`, `descripci
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 2 warnings (0,02 sec)
```

```
mysql> INSERT INTO `impuestos`(`Identificador`, `nombre`, `tipoimpositivo`, `d  
Query OK, 1 row affected (0,00 sec)
```

```
mysql> INSERT INTO `impuestos`(`Identificador`, `nombre`, `tipoimpositivo`, `d  
Query OK, 1 row affected (0,00 sec)
```

```
mysql> INSERT INTO `impuestos`(`Identificador`, `nombre`, `tipoimpositivo`, `d  
Query OK, 1 row affected (0,01 sec)
```

```
mysql> INSERT INTO `impuestos`(`Identificador`, `nombre`, `tipoimpositivo`, `d  
Query OK, 1 row affected (0,01 sec)
```

Consulta SQL:

```
CREATE TABLE pedidos  
(  
    Identificador INT(10) NOT NULL AUTO_INCREMENT ,  
    fecha DATE NOT NULL ,  
    clientes_nombre INT(10) NOT NULL ,  
    impuestos_nombre INT(10) NOT NULL ,  
    FOREIGN KEY (clientes_nombre) REFERENCES clientes(Identificador),  
    FOREIGN KEY (impuestos_nombre) REFERENCES impuestos(Identificador),  
    PRIMARY KEY (`Identificador`)  
) ENGINE = InnoDB;
```

Resultado de la consulta en el terminal:**Consulta SQL:**

```
ALTER TABLE pedidos ADD COLUMN impuestos_nombre INT(10);
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 1 warning (0,01 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

Consulta SQL:

```
UPDATE pedidos SET impuestos_nombre = 1
WHERE impuestos_nombre IS NULL
    OR impuestos_nombre NOT IN (SELECT Identificador FROM impuestos);
```

Resultado de la consulta en el terminal:

```
Query OK, 4 rows affected (0,00 sec)
Rows matched: 4  Changed: 4  Warnings: 0
```

Resultado de la consulta en el terminal:**Consulta SQL:**

```
ALTER TABLE pedidos
ADD CONSTRAINT fk_impuestos
FOREIGN KEY (impuestos_nombre) REFERENCES impuestos(Identificador);
```

Resultado de la consulta en el terminal:

```
Query OK, 4 rows affected (0,03 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

Consulta SQL:

```
ALTER TABLE pedidos MODIFY impuestos_nombre INT(10);
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 1 warning (0,00 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

Consulta SQL:

```
UPDATE pedidos SET impuestos_nombre = NULL
WHERE impuestos_nombre NOT IN (SELECT Identificador FROM impuestos);
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,00 sec)
Rows matched: 0  Changed: 0  Warnings: 0
```

Consulta SQL:

```
ALTER TABLE pedidos
ADD FOREIGN KEY (impuestos_nombre) REFERENCES impuestos(Identificador);
```

Resultado de la consulta en el terminal:

```
Query OK, 4 rows affected (0,03 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

Consulta SQL:

```
CREATE VIEW totales_pedidos AS
SELECT
pedidos.Identificador AS 'Número de pedido',
pedidos.fecha AS 'Fecha del pedido',
clientes.nombre AS 'Nombre del cliente',
SUM(productos.precio*lineaspedido.cantidad) AS 'SubTotal',
impuestos.nombre AS 'Descripción impuesto',
ROUND(SUM(productos.precio*lineaspedido.cantidad)*(impuestos.tipoimpositivo/100)) AS 'Iva',
ROUND(SUM(productos.precio*lineaspedido.cantidad) + SUM(productos.precio*lineaspedido.cantidad*(impuestos.tipoimpositivo/100))) AS 'Total',
FROM pedidos
LEFT JOIN clientes
ON pedidos.clientes_nombre = clientes.Identificador
LEFT JOIN lineaspedido
ON lineaspedido.pedidos_fecha = pedidos.Identificador
LEFT JOIN productos
ON lineaspedido.productos_nombre = productos.Identificador
LEFT JOIN impuestos
ON pedidos.impuestos_nombre = impuestos.Identificador
GROUP BY clientes.Identificador;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,01 sec)
```

6.4. Modificación y ampliación

Consulta SQL:

```
ALTER TABLE `clientes` CHANGE `imagen` `imagen` MEDIUMBLOB NOT NULL;
```

Resultado de la consulta en el terminal:

```
Query OK, 1 row affected (0,02 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

Consulta SQL:

```
UPDATE `clientes` SET `imagen` = 0ffd8ffe000104a4649460001010100600060000ffdt;
```

Resultado de la consulta en el terminal:

Consulta SQL:

```
CREATE TABLE `empresa`.`categorias` (`Identificador` INT(255) NOT NULL AUTO_INCREMENT,
```

Resultado de la consulta en el terminal:

Query OK, 0 rows affected, 1 warning (0,01 sec)

Consulta SQL:

```
ALTER TABLE `productos` CHANGE `categoria` `categoria` INT(255) NOT NULL;
```

Resultado de la consulta en el terminal:

Query OK, 16 rows affected, 1 warning (0,03 sec)
Records: 16 Duplicates: 0 Warnings: 1

Consulta SQL:

```
UPDATE `productos` SET `categoria` = 0;
```

Resultado de la consulta en el terminal:

Query OK, 16 rows affected (0,00 sec)
Rows matched: 16 Changed: 16 Warnings: 0

Consulta SQL:

```
ALTER TABLE `productos` CHANGE `categoria` `categorias_nombre` INT(255) NOT NULL;
```

Resultado de la consulta en el terminal:

Query OK, 0 rows affected, 1 warning (0,01 sec)
Records: 0 Duplicates: 0 Warnings: 1

Consulta SQL:

```
ALTER TABLE productos  
ADD FOREIGN KEY (categorias_nombre) REFERENCES categorias(Identificador);
```

Resultado de la consulta en el terminal:**Consulta SQL:**

```
INSERT INTO `categorias` (`Identificador`, `nombre`) VALUES (NULL, 'Fisico');
```

Resultado de la consulta en el terminal:

```
Query OK, 1 row affected (0,00 sec)
```

Consulta SQL:

```
ALTER TABLE `productos` CHANGE `imagen` `imagen` MEDIUMBLOB NOT NULL;
```

Resultado de la consulta en el terminal:

```
Query OK, 16 rows affected (0,02 sec)  
Records: 16  Duplicates: 0  Warnings: 0
```

```
SHOW INDEXES FROM `empresa`.`pedidos`;
```

Resultado de la consulta en el terminal:

Table	Non_unique	Key_name	Seq_in_index	Column_name
pedidos	0	PRIMARY	1	Identificador
pedidos	1	clientes_nombre	1	clientes_nombre
pedidos	1	impuestos_nombre	1	impuestos_nombre

3 rows in set (0,01 sec)

Consulta SQL:

```
ALTER TABLE `empresa`.`pedidos` DROP INDEX , ADD UNIQUE `numeropedido` (`0`);
```

Resultado de la consulta en el terminal:**Consulta SQL:**

```
ALTER TABLE `pedidos` ADD `pagado` INT(10) NOT NULL AFTER `impuestos_nombre`;
```

Resultado de la consulta en el terminal:**Consulta SQL:**

```
ALTER TABLE `pedidos` ADD `comentarios` TEXT NOT NULL AFTER `pagado`;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 1 warning (0,01 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

Consulta SQL:

```
ALTER TABLE `lineaspedido` ADD `comentarios` TEXT NOT NULL AFTER `cantidad`;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

6.5. Borrar

Consulta SQL:

```
DELETE FROM productos;
```

Resultado de la consulta en el terminal:

```
Query OK, 16 rows affected (0,00 sec)
```

Consulta SQL:

```
ALTER TABLE productos AUTO_INCREMENT = 1;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,00 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Consulta SQL:

```
ALTER TABLE `pedidos` ADD `numerodepedido` VARCHAR(255) NOT NULL AFTER `Identif
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Consulta SQL:

```
ALTER TABLE `pedidos` ADD `impuestos2_nombre` INT(255) NOT NULL AFTER `impuestoc
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 1 warning (0,01 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

7. Base de datos de empresa

8. Procedimientos almacenados

Los procedimientos almacenados son bloques de código SQL que se guardan en el servidor de base de datos y pueden ejecutarse de forma repetida cuando se necesiten. Permiten encapsular una serie de instrucciones en una unidad lógica que puede recibir parámetros, realizar operaciones complejas y devolver resultados.

Características principales:

- **Reutilización:** una vez creado, un procedimiento puede ejecutarse cuantas veces se desee sin volver a escribir su lógica.
- **Modularidad:** ayuda a estructurar el código y separar tareas, facilitando el mantenimiento.
- **Parámetros de entrada y salida:** permite recibir valores al ser invocado (IN), devolver resultados (OUT), o ambos (INOUT).
- **Mejora del rendimiento:** al ejecutarse en el servidor, reduce la necesidad de múltiples viajes entre cliente y base de datos.
- **Seguridad:** puede usarse para limitar el acceso directo a las tablas, permitiendo que los usuarios interactúen con los datos solo a través del procedimiento.

Son especialmente útiles para operaciones recurrentes como actualizaciones masivas, cálculos, migraciones de datos, validaciones o tareas administrativas automatizadas dentro de la base de datos.

Consulta SQL:

```
INSERT INTO `codigospostales` (`Identificador`, `idprovincia`, `codigopostal`,
```

Resultado de la consulta en el terminal:

```
Query OK, 1 row affected (0,01 sec)
```

Consulta SQL:

```
DELIMITER //  
CREATE PROCEDURE ActualizarCodigoProvincia()  
BEGIN  
  
UPDATE  
codigospostales  
SET idprovincia = LEFT(codigopostal,2)  
WHERE idprovincia = '';  
  
END //  
DELIMITER ;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,01 sec)
```

Consulta SQL:

```
DROP PROCEDURE IF EXISTS ActualizarCodigoProvincia;
```

Resultado de la consulta en el terminal:

```
DROP PROCEDURE IF EXISTS ActualizarCodigoProvincia;
```

Consulta SQL:

```
UPDATE pedidos  
SET pagado = 100;
```

Resultado de la consulta en el terminal:

```
Query OK, 4 rows affected (0,00 sec)  
Rows matched: 4  Changed: 4  Warnings: 0
```

Consulta SQL:

```
DELIMITER //  
  
CREATE PROCEDURE ActualizarPagado(IN cantidadpagado INT)  
BEGIN  
    UPDATE pedidos  
        SET pagado = cantidadpagado  
    WHERE pagado = 0;  
END //  
DELIMITER ;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,00 sec)
```

Consulta SQL:

```
DELIMITER //  
  
CREATE PROCEDURE ActualizarPagado(IN cantidadpagado INT)  
BEGIN  
    UPDATE pedidos  
        SET pagado = cantidadpagado;  
END //  
DELIMITER ;
```

9. Eventos

Los eventos en SQL permiten programar la ejecución automática de instrucciones o procedimientos almacenados en momentos concretos o a intervalos definidos. Funcionan de forma similar a las tareas programadas en un sistema operativo, y son muy útiles para automatizar tareas dentro del servidor de base de datos.

Características principales:

- **Ejecución programada:** se pueden configurar para ejecutarse una sola vez en una fecha y hora específicas, o de forma repetitiva (cada minuto, día, semana, etc.).
- **Automatización de tareas:** ideales para realizar copias de seguridad, limpiar registros antiguos, actualizar datos periódicamente o lanzar procedimientos almacenados automáticamente.
- **Gestión desde la base de datos:** no requiere intervención externa ni scripts adicionales; todo se controla desde SQL.
- **Requiere habilitación del programador de eventos:** en MySQL, es necesario activar el planificador (EVENT_SCHEDULER) para que los eventos funcionen correctamente.

Los eventos son una herramienta poderosa para mantener la base de datos organizada y actualizada sin intervención manual, ayudando a reducir errores, ahorrar tiempo y garantizar la consistencia en procesos periódicos.

Consulta SQL:

```
DELIMITER //  
  
CREATE EVENT IF NOT EXISTS ActualizarProvinciaEvento  
ON SCHEDULE EVERY 1 MINUTE  
DO  
    CALL ActualizarCodigoProvincia()  
  
DELIMITER ;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,00 sec)
```

Consulta SQL:

```
CREATE EVENT `ActualizarProvinciaEvento` ON SCHEDULE EVERY 1 MINUTE STARTS '202
```

Resultado de la consulta en el terminal:

Query OK, 0 rows affected, 1 warning (0,00 sec)

10. TCL

TCL (Transaction Control Language) es el subconjunto de SQL encargado de gestionar las transacciones, es decir, bloques de operaciones que deben ejecutarse como una unidad lógica. Su objetivo principal es garantizar que los datos se mantengan consistentes incluso en situaciones de error, interrupciones o conflictos de acceso concurrente.

Las instrucciones más importantes de TCL son:

- START TRANSACTION: inicia una nueva transacción, agrupando las operaciones siguientes.
- COMMIT: confirma todos los cambios realizados durante la transacción, haciéndolos permanentes en la base de datos.
- ROLLBACK: revierte todos los cambios realizados desde que comenzó la transacción, restaurando el estado anterior.

Estas instrucciones permiten controlar cuándo los cambios deben aplicarse y ofrecen mecanismos de seguridad frente a errores o inconsistencias. Son fundamentales cuando se necesita garantizar que un conjunto de acciones (por ejemplo, registrar un pedido y descontar el stock) se ejecute completamente o no se ejecute en absoluto.

Consulta SQL:

```
START TRANSACTION  
  
COMMIT;  
  
ROLLBACK;
```

Consulta SQL:

```
ALTER TABLE `productos` ADD `existencias` INT(10) NOT NULL AFTER `imagen` ;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 1 warning (0,02 sec)  
Records: 0  Duplicates: 0  Warnings: 1
```

Consulta SQL:

```
START TRANSACTION;  
  
INSERT INTO lineaspedido VALUES (NULL, '2024001', '1', '1', '');  
  
UPDATE productos SET existencias = existencias - 1 WHERE Identificador = 1;  
  
COMMIT;
```

Resultado de la consulta en el terminal:**Consulta SQL:**

```
START TRANSACTION;  
  
SELECT existencias INTO @existencias FROM productos WHERE Identificador = 1;  
  
IF @existencias < 1 THEN  
    ROLLBACK;  
ELSE  
    INSERT INTO lineaspedido VALUES (NULL, '2024001', '1', '1', '');  
    UPDATE productos SET existencias = existencias - 1 WHERE Identificador  
    COMMIT;  
END IF;
```

Resultado de la consulta en el terminal:

Consulta SQL:

```
DELIMITER //  
  
CREATE PROCEDURE LineaPedido()  
BEGIN  
    START TRANSACTION;  
  
    SELECT existencias INTO @existencias FROM productos WHERE Identificador = 1  
  
    IF @existencias < 1 THEN  
        ROLLBACK;  
    ELSE  
        INSERT INTO lineaspedido VALUES (NULL, '2024001', '1', '1', '' );  
        UPDATE productos SET existencias = existencias - 1 WHERE Identificador  
        COMMIT;  
    END IF;  
END //  
DELIMITER ;
```

Resultado de la consulta en el terminal:**Consulta SQL:**

```
DELIMITER //  
  
CREATE PROCEDURE LineaPedidoParametro(IN idproducto INT)  
BEGIN  
    START TRANSACTION;  
  
    SELECT existencias INTO @existencias FROM productos WHERE Identificador = i  
  
    IF @existencias < 1 THEN  
        ROLLBACK;  
    ELSE  
        INSERT INTO lineaspedido VALUES (NULL, '2024001', idproducto, '1', '' );  
        UPDATE productos SET existencias = existencias - 1 WHERE Identificador  
        COMMIT;  
    END IF;  
END //  
DELIMITER ;
```

Resultado de la consulta en el terminal:

Consulta SQL:

11. Disparadores

Los disparadores (o triggers) en SQL son bloques de código que se ejecutan automáticamente en respuesta a ciertos eventos que ocurren en una tabla, como inserciones, actualizaciones o eliminaciones de datos. Funcionan como mecanismos de automatización y control, permitiendo ejecutar lógica definida por el usuario sin necesidad de intervención manual.

Características principales:

- **Activación automática:** se ejecutan de forma implícita cuando se produce el evento que los activa (INSERT, UPDATE, DELETE).
- **Antes o después del evento:** se pueden configurar para actuar BEFORE (antes de que se realice la acción) o AFTER (después de que se haya completado).
- **Acceso a valores antiguos y nuevos:** pueden usar las variables OLD y NEW para comparar el estado anterior y posterior de los datos.
- **Automatización de tareas:** útiles para auditar cambios, mantener registros históricos, validar datos o mantener sincronizadas varias tablas.

Los disparadores permiten incorporar lógica de negocio directamente en el nivel de base de datos, lo que mejora la integridad, reduce la duplicación de código en las aplicaciones y garantiza que ciertas reglas se cumplan siempre, sin importar desde qué parte del sistema se modifiquen los datos.

Consulta SQL:

```
CREATE TABLE `empresa`.`registros` (`Identificador` INT(255) NOT NULL AUTO_INCREMENT,
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 3 warnings (0,01 sec)
```

Consulta SQL:

```
DELIMITER //  
  
CREATE TRIGGER insertar_pedido  
AFTER INSERT ON pedidos  
FOR EACH ROW  
BEGIN  
    INSERT INTO registros VALUES(NULL,'INSERT',NOW(),'pedidos',NEW.Identificador);  
END;  
  
//  
DELIMITER ;
```

Resultado de la consulta en el terminal:

Query OK, 0 rows affected (0,01 sec)

Consulta SQL:

```
DELIMITER //  
  
CREATE TRIGGER eliminar_pedido  
AFTER DELETE ON pedidos  
FOR EACH ROW  
BEGIN  
    INSERT INTO registros VALUES(NULL,'DELETE',NOW(),'pedidos',OLD.Identifi  
END;  
  
//  
DELIMITER ;
```

Resultado de la consulta en el terminal:

Query OK, 0 rows affected (0,01 sec)

Consulta SQL:

```
ALTER TABLE `registros` ADD `usuario` VARCHAR(100) NOT NULL AFTER `id`;
```

Resultado de la consulta en el terminal:

Query OK, 0 rows affected (0,02 sec)
Records: 0 Duplicates: 0 Warnings: 0

Consulta SQL:

```
DROP TRIGGER IF EXISTS `insertar_pedido`;
```

Resultado de la consulta en el terminal:

Query OK, 0 rows affected (0,00 sec)

Consulta SQL:

```
DELIMITER //  
  
CREATE TRIGGER insertar_pedido  
AFTER INSERT ON pedidos  
FOR EACH ROW  
BEGIN  
    INSERT INTO registros VALUES(NULL,'INSERT',UNIX_TIMESTAMP(),'pedidos',N  
END;  
  
//  
DELIMITER ;
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected (0,00 sec)
```

Consulta SQL:

```
DELIMITER //  
  
CREATE TRIGGER eliminar_pedido  
AFTER DELETE ON pedidos  
FOR EACH ROW  
BEGIN  
    INSERT INTO registros VALUES(NULL,'DELETE',UNIX_TIMESTAMP(),'pedidos',C  
END;  
  
//  
DELIMITER ;
```

Resultado de la consulta en el terminal:

12. SQL

Consulta SQL:

```
CREATE VIEW facturas_por_año AS
SELECT YEAR(fecha) AS año,
       COUNT(*) AS numero
FROM pedidos
GROUP BY YEAR(fecha)
ORDER BY año;
```

Resultado de la consulta en el terminal:

Consulta SQL:

```
CREATE VIEW facturas_por_cliente AS
SELECT
clientes.nombre as cliente,
COUNT(`clientes_nombre`) AS numero
FROM `pedidos`
LEFT JOIN clientes
ON pedidos.clientes_nombre = clientes.Identificador
GROUP BY `clientes_nombre`
ORDER BY numero DESC
```

Resultado de la consulta en el terminal:

Consulta SQL:

```
CREATE VIEW total_pedido AS
SELECT
pedidos.numerodepedido,
clientes.nombre,
SUM(lineaspedido.cantidad*productos.precio) AS base,
impuestos.nombre AS nombreimpuesto1,
ROUND(SUM(lineaspedido.cantidad*productos.precio)*(impuestos.tipoimpositivo/100)) AS baseimpuesto1,
SUM(lineaspedido.cantidad*productos.precio) - ROUND(SUM(lineaspedido.cantidad*productos.precio)*(impuestos.tipoimpositivo/100)) AS baseimpuesto2,
impuestos2.nombre AS nombreimpuesto2,
ROUND((SUM(lineaspedido.cantidad*productos.precio) - ROUND(SUM(lineaspedido.cantidad*productos.precio)*(impuestos.tipoimpositivo/100)))*impuestos2.tipoimpositivo/100) AS baseimpuesto3,
ROUND((SUM(lineaspedido.cantidad*productos.precio) - ROUND(SUM(lineaspedido.cantidad*productos.precio)*(impuestos.tipoimpositivo/100)))*impuestos2.tipoimpositivo/100) - ROUND((SUM(lineaspedido.cantidad*productos.precio) - ROUND(SUM(lineaspedido.cantidad*productos.precio)*(impuestos.tipoimpositivo/100)))*impuestos2.tipoimpositivo/100) AS baseimpuesto4
FROM pedidos
LEFT JOIN lineaspedido ON pedidos.Identificador = lineaspedido.pedidos_fecha
LEFT JOIN clientes ON pedidos.clientes_nombre = clientes.Identificador
LEFT JOIN productos ON lineaspedido.productos_nombre = productos.Identificador
LEFT JOIN impuestos ON pedidos.impuestos_nombre = impuestos.Identificador
LEFT JOIN impuestos2 ON pedidos.impuestos2_nombre = impuestos2.Identificador
GROUP BY pedidos.numerodepedido
ORDER BY pedidos.numerodepedido ASC
```

Resultado de la consulta en el terminal:

13. Joins

En las bases de datos relacionales, los datos suelen estar distribuidos en múltiples tablas relacionadas entre sí. Para poder extraer información completa y coherente, es necesario combinar los datos de esas tablas, y eso se logra mediante el uso de las operaciones JOIN.

Los JOIN permiten unir filas de dos o más tablas en función de una relación entre ellas, normalmente establecida a través de claves primarias y foráneas. Gracias a ellos, es posible realizar consultas que integren datos dispersos, como obtener los pedidos de un cliente, los productos de una factura o los cursos en los que está matriculado un alumno.

Existen varios tipos de JOIN (como INNER JOIN, LEFT JOIN, RIGHT JOIN, entre otros), y cada uno tiene un comportamiento específico respecto a qué registros se incluyen en el resultado. Comprender cómo y cuándo utilizar cada tipo de unión es esencial para aprovechar al máximo el poder de SQL en entornos relacionales. --- Comenzamos esta sección dedicada al estudio de los distintos tipos de **JOIN** en SQL con la creación de una nueva base de datos que utilizaremos como entorno de pruebas. Esta base de datos servirá como contenedor lógico donde almacenaremos todas las tablas, vistas y datos necesarios para simular un escenario real de gestión formativa.

El nombre elegido para esta base de datos es centrodeformacion, ya que representa un contexto típico en el que intervienen entidades como alumnos, matrículas y categorías de formación. A lo largo de esta sección, utilizaremos este entorno para crear consultas cada vez más complejas, centrándonos en cómo combinar datos de diferentes tablas utilizando los distintos tipos de **JOIN** disponibles en SQL.

Consulta SQL:

```
CREATE DATABASE centrodeformacion;
```

Resultado de la consulta en el terminal:

```
Query OK, 1 row affected (0,01 sec)
```

Una vez creada la base de datos, el siguiente paso consiste en activarla para comenzar a trabajar dentro de ella. Para ello, utilizamos una instrucción que establece el contexto de trabajo sobre la base de datos recién creada.

Esto es fundamental, ya que en SQL es posible tener múltiples bases de datos en un mismo servidor, y necesitamos indicarle al sistema con cuál de ellas queremos operar. Al activar centrodeformacion, cualquier tabla que creamos o consulta que realicemos a partir de este momento se ejecutará dentro de este entorno específico.

Consulta SQL:

```
USE centrodeformacion;
```

Resultado de la consulta en el terminal:

```
Database changed
```

A continuación, definimos una de las tablas clave del proyecto: la tabla de **matrículas**. Esta tabla servirá para registrar qué alumnos se han inscrito en qué cursos, así como la fecha en la que se realizó la inscripción.

La estructura de la tabla incluye los siguientes campos:

- Identificador: un campo numérico que se autoincrementa y actúa como clave primaria. Garantiza que cada matrícula registrada tenga un identificador único.
- nombre: el nombre del curso o formación en la que se ha matriculado el alumno.
- alumno: el nombre del alumno que se ha matriculado.
- fecha: la fecha en que se ha realizado la matrícula.

Esta tabla es esencial para los ejemplos que veremos más adelante, ya que nos permitirá realizar combinaciones (JOIN) con la tabla de alumnos y obtener información detallada sobre qué personas están inscritas en cada curso. Además, se define utilizando el motor de almacenamiento InnoDB, lo que nos garantiza soporte para claves foráneas y transacciones, características clave en entornos relacionales.

Consulta SQL:

```
CREATE TABLE `centrodeformacion`.`matriculas` (`Identificador` INT(255) NOT NULL
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 1 warning (0,01 sec)
```

Para complementar la tabla de matrículas, creamos ahora la tabla de **alumnos**, que contendrá los datos básicos de las personas que forman parte del centro de formación.

Esta tabla está compuesta por los siguientes campos:

- Identificador: un número entero que se autoincrementa y actúa como clave primaria. Cada alumno registrado tendrá un identificador único.
- nombre: el nombre del alumno.

Aunque en este ejemplo estamos simplificando la estructura a un solo campo informativo (nombre), es común que en una base de datos real esta tabla incluya más

información, como apellidos, correo electrónico, teléfono, etc. No obstante, para los objetivos de este capítulo —centrados en la práctica con combinaciones de tablas mediante JOIN— esta estructura es suficiente.

Gracias a esta tabla podremos, más adelante, relacionar los alumnos con sus matrículas y explorar las distintas formas de obtener información combinada entre ambas entidades.

Consulta SQL:

```
CREATE TABLE `centrodeformacion`.`alumnos` (`Identificador` INT(255) NOT NULL A
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 1 warning (0,00 sec)
```

Con estas instrucciones insertamos los primeros datos en nuestras tablas, lo cual nos permitirá probar posteriormente los diferentes tipos de combinaciones con JOIN.

Primero añadimos cuatro registros a la tabla de alumnos. Cada alumno se inserta con un nombre y un identificador que se genera automáticamente gracias al campo AUTO_INCREMENT. Los nombres utilizados (juan, jorge, julia y jose vicente) son ejemplos representativos de estudiantes que podrían estar dados de alta en un centro de formación.

A continuación, se insertan tres registros en la tabla de matrículas. En todos los casos, el curso matriculado es "bases de datos", pero lo interesante es observar que:

- Dos de los alumnos matriculados (julia y jorge) coinciden exactamente con nombres presentes en la tabla de alumnos.
- Uno de los nombres (javier) **no existe** en la tabla de alumnos.

Esta situación no es casual. El hecho de introducir un alumno en matriculas que no esté en alumnos nos permitirá demostrar cómo se comportan los diferentes tipos de JOIN (por ejemplo, cómo un LEFT JOIN puede incluir registros huérfanos y cómo un INNER JOIN los excluye). Así, generaremos intencionadamente un pequeño desfase en los datos para que las diferencias entre las combinaciones queden bien reflejadas.

Consulta SQL:

```
INSERT INTO `alumnos` (`Identificador`, `nombre`) VALUES (NULL, 'juan');
INSERT INTO `alumnos` (`Identificador`, `nombre`) VALUES (NULL, 'jorge');
INSERT INTO `alumnos` (`Identificador`, `nombre`) VALUES (NULL, 'julia');
INSERT INTO `alumnos` (`Identificador`, `nombre`) VALUES (NULL, 'jose vicente')

INSERT INTO `matriculas` (`Identificador`, `nombre`, `alumno`, `fecha`) VALUES
INSERT INTO `matriculas` (`Identificador`, `nombre`, `alumno`, `fecha`) VALUES
INSERT INTO `matriculas` (`Identificador`, `nombre`, `alumno`, `fecha`) VALUES
```

Resultado de la consulta en el terminal:

```
Query OK, 1 row affected (0,00 sec)
```

Aquí realizamos nuestra primera combinación de datos utilizando un **INNER JOIN**, uno de los tipos de JOIN más utilizados en SQL.

En este caso, combinamos las tablas alumnos y matriculas a través del campo nombre de la tabla de alumnos y el campo alumno de la tabla de matrículas. La condición de unión especifica que solo deben combinarse aquellos registros en los que ambos campos coincidan exactamente.

El resultado de esta consulta incluirá únicamente los registros en los que **existe correspondencia entre un alumno y una matrícula**. Es decir, se mostrarán los datos de aquellos alumnos que estén matriculados en algún curso y que estén correctamente registrados en ambas tablas.

En este ejemplo, como hemos introducido un alumno (javier) que no figura en la tabla de alumnos, ese registro **no aparecerá** en el resultado del INNER JOIN. Esta es precisamente la característica principal de este tipo de combinación: **excluye los registros que no tienen coincidencia en ambas tablas**.

Consulta SQL:

```
SELECT *
FROM alumnos
INNER JOIN matriculas
ON alumnos.nombre = matriculas.alumno;
```

Resultado de la consulta en el terminal:

```
+-----+-----+-----+-----+
| Identificador | nombre | Identificador | nombre      | alumno | fecha
+-----+-----+-----+-----+
|           2 | jorge |           2 | bases de datos | jorge | 2024-02-2
|           3 | julia |           1 | bases de datos | julia | 2024-02-2
+-----+-----+-----+-----+
2 rows in set (0,00 sec)
```

Con esta consulta pasamos a utilizar un **LEFT JOIN**, también conocido como **LEFT OUTER JOIN**, que se comporta de forma diferente al INNER JOIN.

En este caso, la combinación se realiza igualmente entre las tablas `alumnos` y `matriculas`, comparando los campos `nombre` y `alumno`, respectivamente. Sin embargo, a diferencia del INNER JOIN, el LEFT JOIN **muestra todos los registros de la tabla situada a la izquierda** de la unión —en este caso, la tabla `alumnos`— **aunque no tengan coincidencia en la tabla de la derecha** (`matriculas`).

Esto significa que:

- Los alumnos que están matriculados aparecerán junto con los datos de su matrícula.
- Los alumnos que **no estén matriculados** también aparecerán en el resultado, pero con los campos correspondientes a `matriculas` en **valor nulo (NULL)**.

Este tipo de combinación es muy útil cuando queremos obtener una visión completa de todos los elementos de una tabla (como los `alumnos`), independientemente de si tienen o no datos asociados en otra (como sus `matrículas`). En nuestro ejemplo, podremos detectar fácilmente qué alumnos están sin matricular.

Consulta SQL:

```
SELECT *
FROM alumnos
LEFT JOIN matriculas
ON alumnos.nombre = matriculas.alumno;
```

Resultado de la consulta en el terminal:

```
+-----+-----+-----+-----+-----+
| Identificador | nombre      | Identificador | nombre      | alumno | fec
+-----+-----+-----+-----+-----+
|           1 | juan        |           NULL | NULL       | NULL   | NUL
|           2 | jorge       |           2    | bases de datos | jorge   | 202
|           3 | julia       |           1    | bases de datos | julia   | 202
|           4 | jose vicente |           NULL | NULL       | NULL   | NUL
+-----+-----+-----+-----+-----+
4 rows in set (0,00 sec)
```

Con esta consulta empleamos un **RIGHT JOIN** o **RIGHT OUTER JOIN**, que es la operación complementaria al **LEFT JOIN**.

En este caso, la unión se hace entre las tablas `alumnos` y `matriculas`, comparando los campos `nombre` (de `alumnos`) y `alumno` (de `matriculas`). La diferencia clave está en que ahora se mostrarán **todos los registros de la tabla de la derecha**, es decir, **todas las matrículas**, aunque no exista un alumno correspondiente en la tabla `alumnos`.

Esto implica que:

- Si la matrícula tiene un alumno que existe en la tabla `alumnos`, se mostrará toda la información combinada.
- Si la matrícula hace referencia a un alumno que **no está registrado**, igualmente se mostrará el registro de matrícula, pero los datos del alumno aparecerán **como valores nulos**.

En nuestro ejemplo, el registro de `javier` en la tabla de `matriculas` aparecerá, a pesar de que no exista un alumno con ese nombre, lo cual nos ayuda a detectar inconsistencias o datos huérfanos en el sistema. Este tipo de JOIN es útil cuando queremos ver **todas las relaciones posibles desde la tabla secundaria**, incluso si no están respaldadas por un dato en la tabla principal.

Consulta SQL:

```
SELECT *
FROM alumnos
RIGHT JOIN matriculas
ON alumnos.nombre = matriculas.alumno;
```

Resultado de la consulta en el terminal:

Identificador	nombre	Identificador	nombre	alumno	fecha
3	julia	1	bases de datos	julia	2024-02-2
2	jorge	2	bases de datos	jorge	2024-02-2
NULL	NULL	3	bases de datos	javier	2024-02-2

Esta consulta utiliza la cláusula UNION para combinar los resultados de un LEFT JOIN y un RIGHT JOIN, lo que nos permite obtener una **unión completa de ambas tablas**, similar al comportamiento de un **FULL OUTER JOIN**, que no está disponible directamente en MySQL.

Veamos cómo funciona:

- El LEFT JOIN incluye todos los registros de la tabla alumnos, aunque no tengan matrícula.
- El RIGHT JOIN incluye todas las matrículas, incluso si no tienen un alumno asociado.

Al aplicar UNION sobre ambas consultas, se fusionan los resultados eliminando duplicados, de forma que obtenemos **todos los alumnos**, estén o no matriculados, y **todas las matrículas**, estén o no asociadas a un alumno existente.

Este patrón es muy útil cuando se necesita una visión global que incluya:

- Alumnos sin matrícula.
- Matrículas sin alumno.
- Alumnos con matrícula correctamente enlazada.

Gracias a esta combinación, conseguimos un resultado que representa de forma completa todas las posibles relaciones (y ausencias de relación) entre las dos tablas.

Consulta SQL:

```
SELECT *
FROM alumnos
LEFT JOIN matriculas ON alumnos.nombre = matriculas.alumno

UNION

SELECT *
FROM alumnos
RIGHT JOIN matriculas ON alumnos.nombre = matriculas.alumno;
```

Resultado de la consulta en el terminal:

Identificador	nombre	Identificador	nombre	alumno	fec
1	juan	NULL	NULL	NULL	NULL
2	jorge	2	bases de datos	jorge	202
3	julia	1	bases de datos	julia	202
4	jose vicente	NULL	NULL	NULL	NULL
NULL	NULL	3	bases de datos	javier	202

5 rows in set (0,00 sec)

En esta consulta utilizamos un **CROSS JOIN**, también conocido como **producto cartesiano**, que genera una combinación de **todos los registros de una tabla con todos los registros de la otra**.

Esto significa que:

- Cada alumno se combinará con **todas las matrículas**, sin tener en cuenta ningún criterio de relación.
- El resultado será una tabla en la que el número total de filas es el producto del número de alumnos por el número de matrículas.

Por ejemplo, si hay 4 alumnos y 3 matrículas, el resultado tendrá 12 filas (4×3).

Este tipo de combinación no es habitual en escenarios de uso real, ya que no establece ninguna conexión lógica entre los datos. Sin embargo, puede ser útil en ciertos contextos como:

- Generación de combinaciones posibles.
- Comparaciones masivas.
- Pruebas de carga o análisis exploratorios.

En nuestro caso, sirve para visualizar claramente lo que ocurre cuando se ignora cualquier relación entre las tablas y se combinan todos los elementos entre sí.

Consulta SQL:

```
SELECT * FROM
alumnos
CROSS JOIN matriculas;
```

Resultado de la consulta en el terminal:

Identificador	nombre	Identificador	nombre	alumno	fec
1	juan	3	bases de datos	javier	202
1	juan	2	bases de datos	jorge	202
1	juan	1	bases de datos	julia	202
2	jorge	3	bases de datos	javier	202
2	jorge	2	bases de datos	jorge	202
2	jorge	1	bases de datos	julia	202
3	julia	3	bases de datos	javier	202
3	julia	2	bases de datos	jorge	202
3	julia	1	bases de datos	julia	202
4	jose vicente	3	bases de datos	javier	202
4	jose vicente	2	bases de datos	jorge	202
4	jose vicente	1	bases de datos	julia	202

12 rows in set (0,00 sec)

Con esta instrucción creamos una nueva tabla llamada categorias, que nos permitirá trabajar con **jerarquías o relaciones entre elementos de la misma tabla**, lo que se conoce como una **relación autorreferenciada**.

La estructura de esta tabla incluye los siguientes campos:

- Identificador: actúa como clave primaria única para cada categoría.
- nombre: almacena el nombre de la categoría, como por ejemplo "Informática", "Marketing" o "DAM".
- pariente: hace referencia al identificador de otra categoría dentro de la misma tabla, es decir, define **una relación de dependencia jerárquica** entre categorías.

Gracias a esta organización, podremos representar estructuras en forma de árbol, donde una categoría puede depender de otra que actúa como su categoría padre. Este modelo es muy útil en contextos donde se requiere una clasificación por niveles, como catálogos, planes de formación, menús o estructuras organizativas.

En los próximos pasos veremos cómo introducir datos en esta tabla y cómo relacionarlos entre sí usando técnicas de auto-unión (self-join).

Consulta SQL:

```
CREATE TABLE `centrodeformacion`.`categorias` (`Identificador` INT(255) NOT NUL
```

Resultado de la consulta en el terminal:

```
Query OK, 0 rows affected, 2 warnings (0,01 sec)
```

Con estas instrucciones insertamos datos en la tabla categorias para construir una estructura jerárquica que simula un sistema de clasificación por niveles.

Los registros se insertan de la siguiente manera:

- El primer registro, 'principal', se establece como la categoría raíz. Al no tener pariente, se le asigna un valor vacío, indicando que no depende de ninguna otra.
- Las siguientes categorías ('informática' y 'marketing') dependen de la categoría con identificador 1, es decir, de 'principal'. Esto crea un segundo nivel jerárquico.
- Finalmente, 'DAM' y 'DAW' se insertan como subcategorías de 'informática', utilizando el identificador 2 como referencia en el campo pariente.

Este modelo nos permite representar relaciones de tipo **padre-hijo**, donde cada categoría puede tener subcategorías, y estas a su vez pueden tener otras. Posteriormente, usaremos técnicas de JOIN dentro de la misma tabla (self-join) para obtener vistas más completas de estas relaciones jerárquicas.

Consulta SQL:

```
INSERT INTO `categorias` (`Identificador`, `nombre`, `pariente`) VALUES (NULL, 'informática', NULL)
INSERT INTO `categorias` (`Identificador`, `nombre`, `pariente`) VALUES (NULL, 'marketing', NULL)
INSERT INTO `categorias` (`Identificador`, `nombre`, `pariente`) VALUES (NULL, 'DAM', 1)
INSERT INTO `categorias` (`Identificador`, `nombre`, `pariente`) VALUES (NULL, 'DAW', 1)
```

Resultado de la consulta en el terminal:

```
Query OK, 1 row affected (0,00 sec)
```

Esta consulta implementa un **self-join**, es decir, una unión de una tabla consigo misma. En este caso, se aplica sobre la tabla categorias para representar las relaciones jerárquicas entre categorías y sus correspondientes categorías superiores.

Se utilizan dos alias (A y B) para distinguir entre la categoría hija y la categoría padre:

- A representa las subcategorías (las que tienen un valor en el campo pariente).
- B representa las categorías padre (las que aparecen referenciadas en ese campo).

La condición de unión A.pariente = B.Identificador establece que una subcategoría (A) está vinculada a su categoría superior (B) mediante la relación entre los identificadores.

El resultado es una lista que muestra dos columnas: el nombre de la subcategoría (categoria) y el nombre de su categoría padre (pariente). Esta forma de consulta es muy útil para visualizar estructuras jerárquicas y navegar por relaciones de

dependencia dentro de una misma tabla.

Consulta SQL:

```
SELECT
A.nombre AS 'categoria',
B.nombre AS 'pariente'
FROM categorias A, categorias B
WHERE A.pariente = B.Identificador;
```

Resultado de la consulta en el terminal:

categoria	pariente
informática	informática
marketing	informática
DAM	marketing
DAW	marketing

4 rows in set (0,00 sec)

14. Insertar y entrar

Consulta SQL:

```
sudo apt install mysql-server
```

Resultado de la consulta en el terminal:

Consulta SQL:

```
sudo mysql -u root -p
```

Resultado de la consulta en el terminal:

Consulta SQL:

```
CREATE DATABASE blog;
```

Resultado de la consulta en el terminal:

Consulta SQL:

```
USE blog;
```

Resultado de la consulta en el terminal:

Consulta SQL:

```
CREATE TABLE entradas (`Identificador` INT(255) NOT NULL AUTO_INCREMENT , `titu
```

Resultado de la consulta en el terminal:

Consulta SQL:

```
exit
```

Resultado de la consulta en el terminal:

Consulta SQL:

```
SHOW TABLES;
```

15. Subconsultas

Las subconsultas, también conocidas como consultas anidadas, son instrucciones SELECT que se encuentran dentro de otra consulta SQL. Permiten usar el resultado de una consulta interna como parte de una externa, lo que permite construir consultas más potentes y dinámicas.

Características principales:

- **Flexibilidad:** pueden colocarse en cláusulas como WHERE, FROM o SELECT.
- **Aislamiento:** cada subconsulta se evalúa de forma independiente y su resultado se integra en la consulta principal.
- **Uso de operadores especiales:** se combinan frecuentemente con IN, EXISTS, ANY, ALL, o comparaciones directas.
- **Soporte de consultas correlacionadas:** en algunos casos, la subconsulta puede hacer referencia a valores de la consulta externa, generando una relación directa entre ambas.

Las subconsultas permiten responder preguntas complejas, como: ¿qué clientes han realizado pedidos?, ¿cuál es el producto más caro?, ¿cuántos registros cumplen una condición dinámica?, etc. Son herramientas fundamentales para el análisis avanzado y la construcción de filtros condicionales sofisticados.

15.1. Preparacion

Consulta SQL:

```
CREATE TABLE `subconsultas`.`clientes` (`Identificador` INT(255) NOT NULL AUTO_
```

Resultado de la consulta en el terminal:

Consulta SQL:

```
INSERT INTO `clientes` (`Identificador`, `nombre`, `email`, `telefono`) VALUES
```

Resultado de la consulta en el terminal:

Consulta SQL:

```
CREATE TABLE `subconsultas`.`pedidos` (`Identificador` INT(255) NOT NULL AUTO_INCREMENT,
```

Resultado de la consulta en el terminal:**Consulta SQL:**

```
INSERT INTO `pedidos` (`Identificador`, `clientes_id`, `fecha`, `total`) VALUES  
INSERT INTO `pedidos` (`Identificador`, `clientes_id`, `fecha`, `total`) VALUES
```

Resultado de la consulta en el terminal:**Consulta SQL:****Resultado de la consulta en el terminal:**

15.2. Where

Consulta SQL:

```
SELECT  
    clientes.nombre,  
    clientes.email,  
    clientes.telefono  
FROM  
    clientes  
WHERE clientes.Identificador IN (  
        SELECT DISTINCT pedidos.clientes_id FROM pedidos  
    );
```

Resultado de la consulta en el terminal:

15.3. From

Consulta SQL:

```
ALTER TABLE `clientes` ADD `pais` VARCHAR(255) NOT NULL AFTER `telefono`;  
  
SELECT  
c.nombre,  
c.email,  
c.telefono  
FROM  
(SELECT * FROM clientes WHERE clientes.pais = 'ES') AS c
```

Resultado de la consulta en el terminal:

15.4. Select

Consulta SQL:

```
SELECT  
clientes.nombre,  
clientes.email,  
clientes.telefono,  
(SELECT COUNT(*) FROM pedidos WHERE pedidos.clientes_id = clientes.Identificado)  
FROM clientes;
```

Resultado de la consulta en el terminal:

15.5. Subconsulta relacion

Consulta SQL:

```
SELECT
c.nombre,
c.email,
c.telefono
FROM clientes c
WHERE EXISTS(
    SELECT 1 FROM pedidos WHERE pedidos.clientes_id = c.Identificador
);
```

Resultado de la consulta en el terminal:

16. Epílogo

Hemos recorrido juntos un apasionante camino a través del mundo del lenguaje SQL, explorando desde los fundamentos más básicos hasta técnicas avanzadas que fortalecen la gestión de bases de datos. A lo largo de estas páginas has adquirido herramientas y habilidades fundamentales que te permitirán enfrentar con seguridad los desafíos de manejar grandes volúmenes de información, optimizando así procesos y decisiones en cualquier entorno profesional.

El aprendizaje de SQL no termina aquí. La verdadera maestría se alcanza con la práctica continua, resolviendo retos reales y adaptando lo aprendido a situaciones diversas. Te invito a seguir profundizando, experimentando y manteniendo siempre la curiosidad viva por explorar nuevas soluciones y enfoques que potencien tus proyectos.

Recuerda que los datos son el activo más valioso de cualquier organización en la actualidad. Administrarlos de forma efectiva y eficiente marca una diferencia sustancial en el éxito y la competitividad empresarial. Confío en que este libro te haya proporcionado la base sólida que necesitas para continuar creciendo y desarrollándote en este fascinante campo.

Gracias por acompañarme en este viaje. ¡Éxitos en tus próximos proyectos SQL y hasta pronto!