Guía de fundamentos de SQL y bases de datos

Con más de 50 comandos

mouredev pro

mouredev.pro/recursos

Índice

Introducción

Esta es una guía de referencia que hace un recorrido por los principales fundamentos del lenguaje de base de datos relacionales SQL (Structured Query Language).

Puedes utilizarla para conocer rápidamente las características de SQL, apoyar tu aprendizaje y obtener información sobre más de 50 comandos, y su uso real, de manera rápida y sencilla, consultando sentencias, cláusulas y sintaxis.

Comentarios

```sql
-- Comentario en una línea

/*
Comentario en
varias líneas
*/
```

SELECT y cláusulas básicas

El comando **SELECT** se usa para consultar datos de una o varias tablas en una base de datos relacional. Las principales cláusulas básicas son:

Cláusula	Definición
SELECT	Lista las columnas a recuperar (usa * para recuperar todas las columnas).
FROM	Indica la tabla (o tablas) de la cual obtener los datos.
WHERE	Filtra las filas según una condición dada. Se pueden usar operadores lógicos (**AND**, **OR**, **NOT**) y de comparación (**=**, **<>** ó **!=**, **>**, **<**, **>=**, **<=**). Por ejemplo, *WHERE edad >= 18 AND ciudad = 'Madrid'*.
DISTINCT	Colocado después de SELECT, devuelve sólo valores únicos, eliminando duplicados en los resultados.
ORDER BY	Ordena los resultados según una o más columnas, de forma ascendente (**ASC**, por defecto) ó descendente (**DESC**). Ej: *ORDER BY precio DESC*.
LIMIT	*(MySQL, PostgreSQL, etc.)* Limita el número de filas devueltas. Ej: *LIMIT 10* devuelve sólo las 10 primeras filas del resultado.

Ejemplos

```sql
-- Seleccionar todas las columnas de una tabla
SELECT *
FROM Productos;

-- Seleccionar columnas específicas con una condición
SELECT nombre, precio
FROM Productos
WHERE categoria = 'Electrónica' AND precio < 500
ORDER BY precio ASC;

-- Seleccionar valores únicos (sin duplicados)
SELECT DISTINCT categoria
FROM Productos;
```

Funciones de agregación

Las funciones de agregación operan sobre múltiples filas para devolver un único valor resumen. Las más comunes son: **COUNT** (cuenta filas), **SUM** (suma), **AVG** (promedio), **MAX** (máximo) y **MIN** (mínimo). Estas funciones suelen usarse junto con **GROUP BY** para agrupar resultados por una o varias columnas, y opcionalmente con **HAVING** para filtrar los grupos.

Cláusula/s	Definición
`COUNT(*)`	Cuenta el número de filas (* cuenta todas, también se puede usar una columna para contar filas no nulas en esa columna).
`SUM, AVG`	Operan sobre columnas numéricas para sumar o promediar valores.
`MAX, MIN`	Obtienen el valor máximo o mínimo de una columna.
`GROUP BY`	Se usa para agrupar filas que tienen el mismo valor en una o varias columnas y así aplicar funciones de agregación sobre cada grupo.
`HAVING`	Filtra los grupos resultantes de un GROUP BY según una condición (similar a WHERE pero aplicado después de agrupar).

Ejemplos

```sql
-- Contar el total de registros en una tabla
SELECT COUNT(*) AS total_filas
FROM Ventas;

-- Sumar ventas y obtener promedio por categoría
SELECT categoria, SUM(cantidad) AS cantidad_total, AVG(monto) AS
cantidad_promedio
FROM Ventas
GROUP BY categoria;

-- Obtener cuántos productos hay por categoría, filtrando categorías con más
de 50 productos
SELECT categoria, COUNT(*) AS cantidad_productos
FROM Productos
GROUP BY categoria
HAVING COUNT(*) > 50;
```

Insertar, actualizar y eliminar datos

Las sentencias **DML** (Data Manipulation Language) permiten modificar los datos almacenados.

Cláusula	Definición
`INSERT`	Añade nuevas filas a una tabla. Se especifica la tabla, las columnas opcionales, y los valores a insertar.
`UPDATE`	Modifica columnas de filas existentes. Se usa **SET** para indicar las nuevas asignaciones y normalmente una cláusula WHERE para elegir las filas a actualizar (si se omite WHERE, se actualizarán todas las filas de la tabla).
`DELETE`	Elimina filas de una tabla. También puede llevar WHERE para borrar filas específicas (sin WHERE eliminará todas las filas de la tabla).

Ejemplos

```sql
-- Insertar una nueva fila en la tabla Clientes
INSERT INTO Clientes (nombre, email, pais)
VALUES ('Brais Moure', 'braismoure@example.com', 'España');

-- Insertar múltiples filas a la vez
INSERT INTO Productos (nombre, categoria, precio)
VALUES
  ('Teclado Mecánico', 'Electrónica', 75.50),
  ('Ratón Inalámbrico', 'Electrónica', 20.00);

-- Actualizar datos (cambiando el precio de un producto específico)
UPDATE Productos
SET precio = 19.99
WHERE nombre = 'Ratón Inalámbrico';


-- Eliminar datos (borrar ventas antiguas)
DELETE FROM Ventas
WHERE fecha < '2020-01-01';
```

Creación y administración de bases de datos y tablas

Las sentencias **DDL** (Data Definition Language) se usan para crear y administrar bases de datos y sus objetos (tablas, etc.):

Cláusula	Definición
`CREATE DATABASE`	Crea una nueva base de datos.
`USE`	Selecciona una base de datos para que las siguientes operaciones se ejecuten en ella.
`DROP DATABASE`	Elimina una base de datos (y todas sus tablas, usar con precaución).
`CREATE TABLE`	Crea una nueva tabla dentro de la base de datos seleccionada, especificando sus columnas y tipos de datos, y opcionalmente restricciones (**constraints**) como **PRIMARY KEY**, **NOT NULL**, etc.
`ALTER TABLE`	Modifica la estructura de una tabla existente (por ejemplo, agregar o eliminar columnas, cambiar el tipo de una columna, agregar/quitar restricciones).
`DROP TABLE`	Elimina una tabla completa y todos sus datos.

Ejemplos

```sql
-- Crear una base de datos y usarla
CREATE DATABASE TiendaDB;
USE TiendaDB;

-- Crear una tabla con columnas y tipos de datos
CREATE TABLE Productos (
    id INT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    categoria VARCHAR(50),
    precio DECIMAL(10,2)
);

-- Alterar una tabla para agregar una columna nueva
ALTER TABLE Productos
ADD COLUMN stock INT DEFAULT 0;

-- Eliminar (dropear) una tabla
DROP TABLE Productos;

-- Eliminar una base de datos completa
DROP DATABASE TiendaDB;
```

Relaciones y claves foráneas

En las bases de datos relacionales, las **relaciones** describen cómo se asocian las filas de distintas tablas mediante claves. Los tipos de relaciones más comunes son:

Relación	Definición
`Uno a uno (1:1)`	A cada fila de la tabla A le corresponde como mucho una fila en la tabla B, y viceversa. Ej: una persona y su DNI, cada persona tiene un DNI único, y cada DNI pertenece a una persona. Se implementa usualmente usando una clave foránea con restricción **UNIQUE** para asegurar unicidad.
`Uno a muchos (1:N)`	A cada fila de la tabla A (lado "uno", por ejemplo Clientes) le pueden corresponder múltiples filas en la tabla B (lado "muchos", por ejemplo Pedidos), pero cada fila de B pertenece solo a un A. Se implementa poniendo una clave foránea en la tabla del lado "muchos" que referencia la clave primaria de la tabla del lado "uno".
`Muchos a muchos (N:M)`	Una fila de A puede relacionarse con muchas de B y viceversa. Ejemplo: Productos y Pedidos (un producto aparece en muchos pedidos y un pedido contiene muchos productos). Esta relación se implementa creando una tabla intermedia que contiene claves foráneas a ambas tablas A y B. Esta tabla puente suele tener una clave primaria compuesta (combinación de ambas claves foráneas) para evitar duplicados.
`Autoreferencia`	Ocurre cuando una tabla se relaciona consigo misma. Esto suele usarse para representar estructuras jerárquicas o relaciones dentro del mismo conjunto de datos.

La clave foránea (**FOREIGN KEY**) es una columna (o combinación de columnas) que referencia la clave primaria de otra tabla, estableciendo una integridad referencial. Al definir una foreign key, podemos opcionalmente especificar acciones en cascada, por ejemplo **ON DELETE CASCADE** para que al borrar una fila padre se borren automáticamente las hijas relacionadas, o **ON UPDATE CASCADE** para propagar cambios de clave primaria.

Ejemplos

```sql
-- Relación 1:N: Clientes y Pedidos (un cliente tiene muchos pedidos)
CREATE TABLE Clientes (
    id INT PRIMARY KEY,
    nombre VARCHAR(100)
);
CREATE TABLE Pedidos (
    id INT PRIMARY KEY,
    fecha DATE,
    cliente_id INT,
    -- Definir cliente_id como clave foránea referenciando a Clientes(id)
    FOREIGN KEY (cliente_id) REFERENCES Clientes(id)
);

-- Relación N:M: Estudiantes y Cursos (un estudiante puede inscribirse en
muchos cursos y cada curso tiene varios estudiantes)
CREATE TABLE Estudiantes (
    id INT PRIMARY KEY,
    nombre VARCHAR(100)
);
CREATE TABLE Cursos (
    id INT PRIMARY KEY,
    titulo VARCHAR(100)
);
-- Tabla intermedia para la relación muchos a muchos
CREATE TABLE EstudianteCurso (
    estudiante_id INT,
    curso_id INT,
    -- Clave primaria compuesta de ambas columnas (cada pareja
estudiante-curso es única)
    PRIMARY KEY (estudiante_id, curso_id),
    -- Claves foráneas que referencian a las tablas principales
    FOREIGN KEY (estudiante_id) REFERENCES Estudiantes(id),
    FOREIGN KEY (curso_id) REFERENCES Cursos(id)
);
```

JOINs y combinación de tablas

Las operaciones **JOIN** permiten combinar filas de dos (o más) tablas basándose en una columna común, facilitando consultas que requieren datos de varias tablas a la vez. Los principales tipos de JOIN son:

Cláusula	Definición
`INNER JOIN`	Devuelve sólo las filas que coinciden en ambas tablas según la condición de unión. Las filas sin correspondencia en la otra tabla se excluyen. Equivale a la intersección de tablas según la clave de unión.
`LEFT JOIN (LEFT OUTER JOIN)`	Devuelve todas las filas de la tabla izquierda, y las filas coincidentes de la tabla derecha. Las filas de la izquierda que no tengan correspondencia en la derecha aparecerán con valores NULL en las columnas de la derecha.
`RIGHT JOIN (RIGHT OUTER JOIN)`	Similar al LEFT JOIN, pero devuelve todas las filas de la tabla derecha y las correspondientes de la izquierda (las no coincidentes de la derecha tendrán NULL en columnas de la izquierda).
`FULL OUTER JOIN`	(*No soportado directamente en MySQL*) Devuelve todas las filas cuando hay coincidencia en una de las tablas u en la otra, incluyendo las no emparejadas de ambas (las no coincidentes muestran NULLs para la tabla opuesta).
`CROSS JOIN`	devuelve el producto cartesiano entre dos tablas (todas las combinaciones posibles de filas). Usado raramente, principalmente con pequeñas tablas o para generar combinaciones.

Sintaxis básica

```SQL
SELECT A.columna, B.columna
FROM TablaA A
[INNER|LEFT|RIGHT] JOIN TablaB B
    ON A.col_comun = B.col_comun;
```

Ejemplos

```SQL
-- INNER JOIN: obtener pedidos con datos del cliente asociado
SELECT P.id, P.fecha, C.nombre AS cliente
FROM Pedidos P
INNER JOIN Clientes C ON P.cliente_id = C.id;

-- LEFT JOIN: listar todos los clientes y sus pedidos (incluye clientes sin pedidos)
SELECT C.nombre, P.id AS pedido_id, P.fecha
FROM Clientes C
LEFT JOIN Pedidos P ON P.cliente_id = C.id;

-- CROSS JOIN: combinar todos los productos con todas las tiendas (ejemplo de producto cartesiano)
SELECT Prod.nombre, Tienda.ciudad
FROM Productos Prod
CROSS JOIN Tiendas Tienda;
```

UNION

El operador **UNION** permite combinar verticalmente los resultados de dos o más consultas SELECT. Las columnas de las consultas a unir deben ser del mismo número y tipo, ya que el UNION apila las filas de resultados. Por defecto, UNION elimina duplicados entre los resultados combinados; si se desea conservar todas las filas (incluyendo duplicados), se puede usar **UNION ALL**.

Ejemplo

```sql
-- Obtener lista combinada de todos los nombres de clientes y proveedores
SELECT nombre, 'Cliente' AS tipo
FROM Clientes
UNION
SELECT nombre, 'Proveedor' AS tipo
FROM Proveedores;
```

Subconsultas (subqueries)

Una **subconsulta** es una consulta SQL anidada dentro de otra. Las subconsultas pueden ubicarse en varias partes de una sentencia SQL: en la cláusula WHERE (para filtrar usando resultados de otra consulta), en la cláusula FROM (como tabla derivada o subconsulta en FROM), o incluso en la selección (SELECT) para calcular valores derivados.

Subconsulta	Definición
`En WHERE`	Suele retornar un conjunto de valores para usar con operadores como **IN**, **EXISTS** o comparaciones con un solo valor.
`En SELECT (escalar)`	Retorna un solo valor que se utiliza como columna en la consulta externa.
`En FROM`	Define un resultado derivado que se puede tratar como tabla temporal en la consulta externa (hay que asignarle un alias).

Ejemplos

```SQL
-- Subconsulta en WHERE: obtener clientes que han realizado algún pedido de monto > 1000
SELECT nombre
FROM Clientes
WHERE id IN (
    SELECT cliente_id
    FROM Pedidos
    WHERE total > 1000
);

-- Subconsulta escalar en SELECT: obtener el total de pedidos como columna adicional
SELECT C.nombre,
       (SELECT COUNT(*) FROM Pedidos P WHERE P.cliente_id = C.id) AS cantidad_pedidos
FROM Clientes C;
```

- En el primer ejemplo, la subconsulta encuentra los id de clientes con pedidos mayores a 1000, y la consulta externa selecciona los nombres de esos clientes.

- En el segundo, la subconsulta calcula el número de pedidos de cada cliente y devuelve ese valor como una columna calculada cantidad_pedidos en el resultado.

CASE, IFNULL y Alias

En SQL existen expresiones y funciones útiles para transformar y manejar los datos en las consultas, así como la posibilidad de asignar **alias** (nombres temporales) a columnas o tablas para mejorar la legibilidad.

Sentencia	Definición
`CASE`	Es una expresión condicional que permite retornar valores distintos según diferentes condiciones (similar a una estructura IF/ELSE). Se utiliza frecuentemente en la lista de selección (SELECT) o en cláusulas como ORDER BY.
`IFNULL (expr, valor_si_nulo)`	Función (propia de MySQL) que devuelve valor_si_nulo si expr resulta ser NULL; en caso contrario devuelve el valor de expr. Es útil para sustituir valores nulos por algún valor por defecto en las consultas. (En SQL estándar existe la similar COALESCE(expr, valor_si_nulo) que admite múltiples expresiones).
`Alias`	Un alias es un nombre alternativo que se asigna a una columna o tabla en una consulta, usando la palabra clave AS (opcional para columnas en muchos sistemas). Los alias de columna renombran el encabezado del resultado para esa columna, y los alias de tabla permiten referirse a las tablas con un nombre corto (especialmente útil en JOINS o subconsultas).

Ejemplos

```SQL
-- CASE
SELECT nombre,
       CASE
         WHEN precio >= 100 THEN 'Caro'
         WHEN precio < 100 AND precio > 50 THEN 'Moderado'
         ELSE 'Barato'
       END AS rango_precio
FROM Productos;

-- IFNULL
SELECT nombre, IFNULL(telefono, 'No proporcionado') AS telefono_contacto
FROM Clientes;

-- Alias de columna
SELECT AVG(precio) AS precio_promedio
FROM Productos;

-- Alias de tabla (y alias de columna combinados en la consulta con JOIN)
SELECT p.nombre AS producto, c.nombre AS categoria
FROM Productos AS p
JOIN Categorias AS c ON p.categoria_id = c.id;
```

- En el ejemplo del **CASE**, según el valor de precio, la expresión CASE produce un texto 'Caro', 'Moderado' o 'Barato' para cada producto, y se presenta bajo la columna alias rango_precio.

- En el ejemplo del **IFNULL**, si el teléfono de un cliente es NULL (desconocido), la función IFNULL retornará 'No proporcionado' en su lugar, facilitando la presentación de datos.

- En el caso de los **alias**, en el primer ejemplo se usa **AS** precio_promedio para que la columna calculada por AVG aparezca con ese encabezado en el resultado. En el segundo, se asignan alias p y c a las tablas Productos y Categorias respectivamente, y alias de columna para clarificar la salida. Esto hace más legible la consulta, especialmente cuando se seleccionan columnas con el mismo nombre de tablas diferentes.

Restricciones

Las **restricciones** (constraints) definen reglas que garantizan la integridad de los datos en una tabla, normalmente especificadas al crear o alterar una tabla. Principales restricciones:

Restricción	Definición
`NOT NULL`	La columna no permite valores NULL. Cada fila debe tener un valor en esa columna.
`UNIQUE`	Todos los valores en esa columna (o conjunto de columnas) deben ser únicos, no se repiten entre filas. Se puede definir más de un UNIQUE por tabla (a diferencia de PRIMARY KEY que solo puede haber uno).
`PRIMARY KEY`	Identifica de forma única cada fila de la tabla. Equivale a una combinación de NOT NULL + UNIQUE sobre la(s) columna(s) definidas. Solo puede haber una clave primaria por tabla. Generalmente es un número ID auto-incremental o un código único. La mayoría de SGBD crean automáticamente un índice sobre la PK para optimizar búsquedas.
`CHECK`	Impone una condición booleana que los valores de la columna (o combinación de columnas) deben cumplir. Por ejemplo, CHECK (edad >= 0) asegura que no haya edades negativas. (Nota: MySQL soporta la sintaxis de CHECK pero hasta versiones recientes no las aplicaba; verificar soporte según versión.)
`DEFAULT`	Especifica un valor por defecto para la columna si en una inserción no se proporciona uno. Por ejemplo, DEFAULT 0 o DEFAULT 'N/A'.
`AUTO_INCREMENT`	(MySQL) Atributo que indica que la columna (generalmente la PK) se incrementará automáticamente en cada nueva inserción. Solo se puede aplicar a columnas numéricas enteras, y una tabla puede tener a lo sumo una columna AUTO_INCREMENT.

Estas restricciones se pueden declarar en línea junto a la definición de la columna, o como restricciones de tabla separadas.

Ejemplos

```SQL
CREATE TABLE Usuarios (
    id INT PRIMARY KEY AUTO_INCREMENT,     -- PK auto-incremental
    nombre VARCHAR(50) NOT NULL,           -- no se permite NULL
    email VARCHAR(100) UNIQUE,             -- no puede repetir valor entre
filas
    edad INT DEFAULT 18,                   -- valor por defecto si es NULL en
INSERT
    estado VARCHAR(10)
        CHECK (estado IN ('activo', 'inactivo'))  -- el valor debe ser
'activo' o 'inactivo'
);
```

En este ejemplo, la tabla Usuarios tiene id como clave primaria autoincremental, nombre no acepta valores nulos, email debe ser único, edad por defecto será 18 si no se especifica, y estado debe ser 'activo' o 'inactivo' gracias a la restricción CHECK.

Conceptos avanzados

En SQL, además de las consultas básicas, existen características avanzadas para optimizar y manejar lógica de negocio en la base de datos:

Sentencia	Definición
INDEX (Índice)	Estructuras que mejoran la velocidad de las consultas al permitir acceso rápido a los datos de una tabla basándose en las columnas indexadas (funcionan de forma similar a un índice de un libro). Un índice se puede crear sobre una o varias columnas. Sin embargo, los índices conllevan un costo en espacio y en tiempo de inserción/actualización (hay que mantener el índice).
TRIGGER (Disparador)	Son procedimientos asociados a una tabla que se ejecutan automáticamente ante ciertos eventos (INSERT, UPDATE o DELETE) en dicha tabla. Un trigger puede ser ANTES (BEFORE) o DESPUÉS (AFTER) de la operación, y puede usarse para validar datos, mantener logs, sincronizar tablas, etc. Dentro del cuerpo del trigger, en muchos SGBD (como MySQL) se pueden usar las pseudotablas NEW (nuevo registro) y OLD (registro anterior, en updates/deletes) para referirse a los valores.
VIEW (Vista)	Una vista es una consulta almacenada en la base de datos que actúa como una tabla virtual. Una vista no almacena datos por sí misma (salvo vistas materializadas en ciertos SGBD), sino que muestra los datos resultantes de la consulta definida. Las vistas se usan para simplificar consultas complejas, reutilizar lógica SQL o restringir el acceso a ciertos datos presentando solo algunas columnas/filas.
STORED PROCEDURE (Procedimiento almacenado)	Son rutinas SQL guardadas en el servidor de base de datos, que pueden incluir secuencias de varios comandos SQL, lógica condicional, bucles, variables, etc. Se pueden invocar para ejecutar tareas complejas en el servidor sin tener que enviar múltiples comandos desde la aplicación cliente. Pueden aceptar parámetros de entrada (IN), salida (OUT) o entrada/salida (INOUT).

TRANSACTION (Transacción)	Una transacción es un conjunto de operaciones SQL que se ejecutan de forma atómica, es decir, todas ocurren o ninguna (si alguna falla, se deshacen las ya realizadas). Las transacciones garantizan la integridad de los datos en operaciones que involucran múltiples cambios. Comandos básicos: • START TRANSACTION; (o BEGIN;) inicia una nueva transacción. • COMMIT; confirma la transacción, haciendo permanentes todos los cambios realizados desde el inicio de la transacción. • ROLLBACK; revierte todos los cambios hechos en la transacción, dejando la base de datos como estaba antes de iniciarla.

Ejemplos

```sql
-- Crear índice
CREATE INDEX idx_nombre ON Usuarios(nombre);

-- Crear trigger
DELIMITER $$
CREATE TRIGGER trg_log_pedido
AFTER INSERT ON Pedidos
FOR EACH ROW
BEGIN
    INSERT INTO Pedidos_Log(pedido_id, fecha_creacion)
    VALUES (NEW.id, NOW());
END $$
DELIMITER ;

-- Crear vista
CREATE VIEW VentasAnuales AS
SELECT YEAR(fecha) AS anio, SUM(total) AS total_anual
FROM Ventas
GROUP BY YEAR(fecha);


-- Crear procedimiento almacenado
DELIMITER $$
```

```
CREATE PROCEDURE ObtenerPedidosPorCliente(IN p_cliente_id INT)
BEGIN
    SELECT *
    FROM Pedidos
    WHERE cliente_id = p_cliente_id;
END $$
DELIMITER ;

-- Llamar al procedimiento almacenado
CALL ObtenerPedidosPorCliente(123);

-- Transacción (transferencia entre cuentas)
START TRANSACTION;

-- Retira 100 de la cuenta 1
UPDATE Cuentas SET saldo = saldo - 100 WHERE id = 1;

-- Deposita 100 a la cuenta 2
UPDATE Cuentas SET saldo = saldo + 100 WHERE id = 2;

-- Verificar saldos, etc., luego decidir:
COMMIT;  -- Si todo está correcto, confirmar cambios
-- ROLLBACK;  -- Si algo falla antes del COMMIT, se puede ejecutar rollback
para deshacer
```

- El ejemplo del index se crea un índice llamado idx_nombre sobre la columna nombre de la tabla Usuarios. Ahora las búsquedas filtrando por nombre serán más rápidas (Para un índice único, se puede usar CREATE UNIQUE INDEX o declarar UNIQUE en la tabla).

- El ejemplo del disparador se define un trigger trg_log_pedido que se ejecuta después de cada inserción en la tabla Pedidos. Por cada nueva fila insertada, inserta un registro en Pedidos_Log con el ID del pedido y la fecha actual (NOW() en MySQL). Se usa DELIMITER en MySQL para poder incluir el trigger en un script, cambiando temporalmente el delimitador de comandos debido al BEGIN...END que contiene ; internos.

- En el ejemplo de las view, VentasAnuales es una vista que resume la tabla Ventas por año, mostrando para cada año (anio) la suma total de ventas (total_anual). Luego podríamos consultar SELECT * FROM VentasAnuales;

como si fuera una tabla normal. Las vistas se actualizan automáticamente al cambiar los datos subyacentes (porque siempre calculan sobre la consulta base al usarse).

- En el ejemplo de los procedimientos almacenados, se crea uno llamado ObtenerPedidosPorCliente que recibe un ID de cliente como parámetro de entrada y devuelve todas las filas de Pedidos correspondientes a ese cliente. Por otra parte muestra cómo llamarlo. Los procedimientos almacenados permiten encapsular lógica de negocio en la BD. (En MySQL y otros, se cambia el delimitador al crear procedimientos debido a la presencia de BEGIN...END.)

- En el último ejemplo sobre transacciones, se realizan dos operaciones de actualización como parte de una transacción. Si ambas tienen éxito y las reglas de negocio se cumplen, se ejecuta COMMIT para aplicar los cambios. Si ocurre algún problema (por ejemplo, la segunda operación no puede realizarse), un ROLLBACK desharía la primera operación, asegurando que no se retire dinero de la cuenta 1 sin acreditarlo en la cuenta 2. Las transacciones siguen las propiedades ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad) para asegurar confiabilidad en las operaciones de la base de datos.

¿Quieres aprender más sobre SQL y bases de datos?

Aquí tienes mis cursos para aprender SQL y bases de datos relacionales desde cero.

Curso gratis en YouTube:
https://mouredev.link/sql

Curso con extras (lecciones por tema, ejercicios prácticos reales, soporte, comunidad, test para validar conocimientos y certificado) en el campus de estudiantes mouredev pro:
https://mouredev.pro/cursos/sql-desde-cero

(Utiliza el cupón "PRO" para acceder con un 10% de descuento a todas las suscripciones y cursos del campus).

mouredev pro

Estudia programación y desarrollo de software de manera diferente

mouredev.pro