Transacciones

Clase 05_20-05 Transacciones

1- **Gestión de Transacciones y bloqueos**. Objetivos:

a. Aprender a gestionar transacciones de manera segura.
b. Comprender el concepto de bloqueo y cómo afecta la integridad de los datos.
c. Aplicar comandos para iniciar, confirmar y deshacer transacciones.
d. Identificar y solucionar problemas de bloqueos y deadlocks.

2. ¿Qué es una Transacción?

Una **Transacción** es un conjunto de operaciones SQL que se ejecutan de forma atómica, es decir, o se completan todas o no se ejecuta ninguna.
Esto asegura la consistencia e integridad de los datos.

Ejemplo: Transferir dinero entre cuentas.

- Debe descontarse de una cuenta y acreditarse a otra.
- Si una operación falla, se debe revertir la anterior para mantener la integridad de los datos.

3. Comandos Principales:

- `START TRANSACTION;` o `BEGIN;`: Inicia una transacción. Si bien no son exactamente lo mismo, porque `BEGIN` es un alias de `START TRANSACTION`.
- `COMMIT;`: Confirma los cambios realizados en la transacción.
- `ROLLBACK;`: Revierte los cambios realizados **dentro de la transacción actual** y vuelve al estado anterior al `START TRANSACTION`.

Ejemplo de transacción:

```
START TRANSACTION;
UPDATE cuentas SET saldo = saldo - 500 WHERE id = 1;
UPDATE cuentas SET saldo = saldo + 500 WHERE id = 2;
COMMIT;
ROLLBACK;
```

Si ocurre un error entre las dos actualizaciones, podemos usar `ROLLBACK;` para revertir ambas operaciones.

4. Bloqueos ó Locks:

Los bloqueos garantizan que los datos no se modifiquen por otras transacciones mientras se ejecuta la transacción actual.

Tipos de Bloqueos:

- **READ LOCK:** Permite leer pero no modificar los datos.
- **WRITE LOCK:** Impide leer y modificar los datos hasta que se libere el bloqueo.
- **READ WRITE LOCK:** Permite leer y escribir mientras se mantiene el bloqueo.
- **Los bloqueos pueden ser a nivel de tabla o a nivel de fila.**

Ejemplo de Bloqueo:

```
LOCK TABLES cuentas WRITE;
UPDATE cuentas SET saldo = saldo - 300 WHERE id = 1;
UNLOCK TABLES;
```

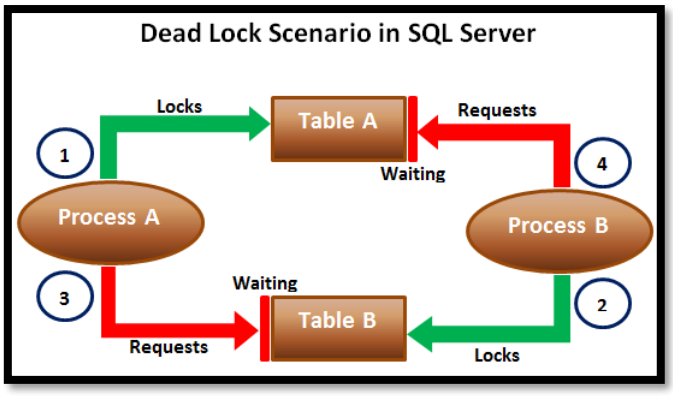
En este caso, nadie podrá leer ni modificar la tabla `cuentas` hasta que se libere el bloqueo.

5. Deadlocks:

Un deadlock ocurre cuando dos o más transacciones quedan bloqueadas esperando recursos que la otra tiene bloqueados.

Ejemplo de Deadlock:

- Transacción A bloquea la tabla `cuentas` y espera a `transacciones`.
- Transacción B bloquea la tabla `transacciones` y espera a `cuentas`.



Para solucionar un **Deadlock**, se puede:

- Cancelar una de las transacciones (`ROLLBACK;`).
- Diseñar cuidadosamente el orden de las operaciones.

6. Ejercicio de aplicación para realizar en MySQL:

En este ejemplo, se creará un sistema básico de gestión de transferencias bancarias utilizando transacciones y bloqueos. Se realizarán las siguientes operaciones:

1. Crear las tablas `cuentas` y `transacciones`.
2. Insertar datos de prueba.
3. Realizar una transacción de transferencia entre cuentas. Agregar una verificación del saldo antes de realizar la transferencia para evitar que una cuenta quede en negativo
4. Aplicar un bloqueo a las tablas para asegurar la integridad de los datos.
5. Simular un deadlock y solucionarlo. Usar el comando **SHOW ENGINE INNODB STATUS;**

Paso 1: Crear las tablas:

```
CREATE TABLE cuentas (
id INT PRIMARY KEY,
titular VARCHAR(50),
saldo DECIMAL(10, 2)
);

CREATE TABLE transacciones (
id INT PRIMARY KEY,
cuenta_origen INT,
cuenta_destino INT,
monto DECIMAL(10, 2),
fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
FOREIGN KEY (cuenta_origen) REFERENCES cuentas(id),
FOREIGN KEY (cuenta_destino) REFERENCES cuentas(id)
);
```

Paso 2: Insertar datos de prueba:

```
INSERT INTO cuentas (id, titular, saldo) VALUES
(1, 'Juan Pérez', 1000),
(2, 'Ana Gómez', 1500),
(3, 'Luca Santillana',2100),
(4, 'Marcela Fernandez',3000),
)5, 'Elisa Dodenoa', 5000);
```

Paso 3: Realizar una transferencia utilizando una transacción:

```
-- Inicia la transacción

START TRANSACTION;
```

```
-- Verificar si la cuenta de origen tiene saldo suficiente

SELECT saldo FROM cuentas WHERE id = 1;

-- Actualizar saldo en la cuenta de origen (restar 200)

UPDATE cuentas

SET saldo = saldo - 200

WHERE id = 1 AND saldo >= 200;

-- Actualizar saldo en la cuenta de destino (sumar 200)

UPDATE cuentas

SET saldo = saldo + 200

WHERE id = 2;

-- Registrar la transacción en la tabla transacciones

INSERT INTO transacciones (cuenta_origen, cuenta_destino, monto)

VALUES (1, 2, 200);

-- Confirmar los cambios

COMMIT;
```

Si ocurre un error antes del `COMMIT`, se puede ejecutar:

```
ROLLBACK;
```

Paso 4: Aplicar un bloqueo:

```
-- Bloquear la tabla 'cuentas' para escritura

LOCK TABLES cuentas WRITE;

-- Realizar la actualización en la tabla bloqueada

UPDATE cuentas

SET saldo = saldo - 100

WHERE id = 1;

-- Liberar el bloqueo de la tabla

UNLOCK TABLES;
```

Limitaciones:

• Mientras la tabla está bloqueada con `WRITE`, **nadie puede leer ni escribir en** `cuentas` hasta que se libere con `UNLOCK TABLES.`
• Si hay muchas transacciones simultáneas, este enfoque puede ser problemático, ya que puede bloquear otras operaciones importantes.

Paso 5: Simular un deadlock:

En dos sesiones de MySQL diferentes, ejecutar los siguientes comandos:

- **Sesión 1:**

```
-- Iniciar la transacción en Sesión 1

START TRANSACTION;

-- Intentar restar 50 a la cuenta 1

UPDATE cuentas SET saldo = saldo - 50 WHERE id = 1;
```

- **Sesión 2:**

```
START TRANSACTION;
UPDATE cuentas SET saldo = saldo + 50 WHERE id = 2;
```

- En la **Sesión 1**, intentar ejecutar:

```
UPDATE cuentas SET saldo = saldo - 50 WHERE id = 2;
```

- En la **Sesión 2**, intentar ejecutar:

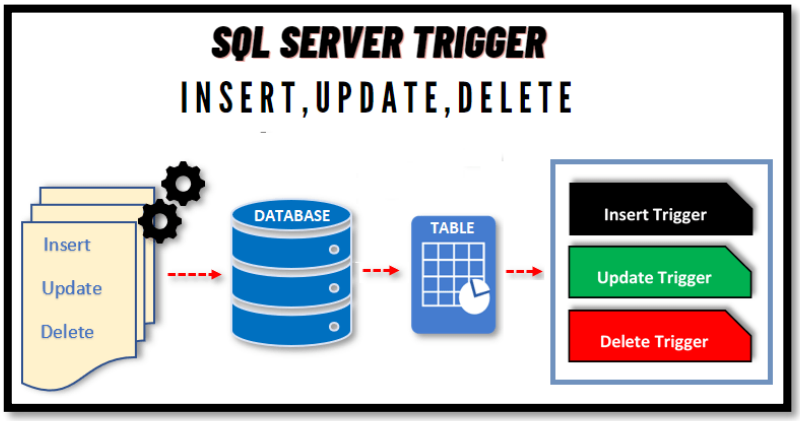
```
UPDATE cuentas SET saldo = saldo + 50 WHERE id = 1;
```

Esto causará un deadlock. La solución es ejecutar `ROLLBACK;` en una de las sesiones para liberar el bloqueo.

7. Ejercicios de aplicación para realizar en MySQL:

- Crear dos tablas `cuentas` y `transacciones`. Insertar datos de ejemplo.
- Realizar una transacción que mueva dinero entre cuentas.
- Aplicar bloqueos a las tablas e intentar realizar operaciones concurrentes.
- Simular un deadlock y solucionarlo usando `ROLLBACK`.

8. Procedimientos Almacenados, Funciones y Triggers. Objetivos:



- Crear procedimientos almacenados y funciones en MySQL.
- Implementar triggers para automatizar acciones en las tablas.
- Realizar ejemplos prácticos que permitan entender cada componente.

9. ¿Qué es un Procedimiento Almacenado?

Un procedimiento almacenado es un conjunto de sentencias SQL que se almacena en el servidor y se puede ejecutar cuando se necesite, sin tener que escribirlas nuevamente. Permite automatizar tareas repetitivas y consolidar operaciones complejas en un único bloque de código.

Sintaxis Básica:

```
DELIMITER //
CREATE PROCEDURE nombre_procedimiento()
BEGIN
  -- Sentencias SQL
END //
DELIMITER ;
```

Ejemplo: Crear un procedimiento que muestre todos los registros de la tabla `cuentas`:

```
DELIMITER //
CREATE PROCEDURE MostrarCuentas()
BEGIN
  SELECT * FROM cuentas;
END //
DELIMITER ;
```

Para ejecutarlo:

```
CALL MostrarCuentas();
```

10. Procedimientos con Parámetros

Los procedimientos pueden aceptar parámetros de entrada (`IN`), de salida (`OUT`) o ambos (`INOUT`).

Ejemplo: Crear un procedimiento que actualice el saldo de una cuenta:

```
DELIMITER //
```

```
CREATE PROCEDURE ActualizarSaldo(IN cuentaID INT, IN nuevoSaldo
DECIMAL(10, 2))
BEGIN
  UPDATE cuentas SET saldo = nuevoSaldo WHERE id = cuentaID;
END //
DELIMITER ;
```

Para ejecutarlo:

```
CALL ActualizarSaldo(1, 1200);
```

O También:

```
-- Cambiar delimitador para definir el procedimiento
DELIMITER //

CREATE PROCEDURE MostrarCuentas()
BEGIN
    SELECT * FROM cuentas;
END //

-- Restaurar delimitador original
DELIMITER ;

-- Ejecutar el procedimiento
CALL MostrarCuentas();
```

11. Funciones MySQL

Una función es un bloque de código que devuelve un valor. A diferencia de un procedimiento, siempre debe retornar un dato.

Sintaxis Básica:

```
DELIMITER //
CREATE FUNCTION nombre_funcion()
RETURNS tipo_de_dato
BEGIN
  DECLARE resultado tipo_de_dato;
  -- Operaciones
  RETURN resultado;
END //
DELIMITER ;
```

Ejemplo: Crear una función que calcule el total de saldo en todas las cuentas:

```
DELIMITER //
CREATE FUNCTION TotalSaldo()
RETURNS DECIMAL(10, 2)
BEGIN
  DECLARE total DECIMAL(10, 2);
  SELECT SUM(saldo) INTO total FROM cuentas;
  RETURN total;
END //
DELIMITER ;
```

Para usarla:

```
SELECT TotalSaldo();
```

12. Triggers (o eventos desencadenantes, disparadores) en MySQL

Un trigger (disparador) es un conjunto de instrucciones que se ejecuta automáticamente cuando ocurre un evento en una tabla (`INSERT`, `UPDATE`, `DELETE`).

Tipos de Triggers:

1. **BEFORE**: Se ejecuta antes del evento (`INSERT`, `UPDATE`, `DELETE`).
2. **AFTER**: Se ejecuta después del evento.

Sintaxis Básica:

```
CREATE TRIGGER nombre_trigger
AFTER INSERT ON nombre_tabla
FOR EACH ROW
BEGIN
  -- Operaciones
END;
```

Ejemplo 1: Crear un trigger que registre todas las transacciones realizadas en una tabla `log_transacciones`:

```
CREATE TABLE log_transacciones (
  id INT AUTO_INCREMENT PRIMARY KEY,
  cuenta_id INT,
  monto DECIMAL(10, 2),
  fecha TIMESTAMP
);

DELIMITER //
CREATE TRIGGER registrar_transaccion
AFTER INSERT ON transacciones
FOR EACH ROW
BEGIN
  INSERT INTO log_transacciones (cuenta_id, monto, fecha)
  VALUES (NEW.cuenta_origen, NEW.monto, NOW());
END //
DELIMITER ;
```

Con este trigger, cada vez que se inserte una transacción, se registrará automáticamente en `log_transacciones`.

Ejemplo 2: Registro de Cambios en una Tabla

Vamos a crear un trigger que registre los cambios realizados en la tabla `cuentas` y guarde un historial en una tabla `log_cuentas`:

1. Crear la tabla `log_cuentas`:

```
CREATE TABLE log_cuentas (
  log_id INT AUTO_INCREMENT PRIMARY KEY,
  cuenta_id INT NOT NULL,
```

```
  accion VARCHAR(50)NOT NULL,
  fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  saldo DECIMAL(10, 2)NOT NULL
);
```

2. Crear el trigger:

```
DELIMITER //
CREATE TRIGGER after_update_cuentas
AFTER UPDATE ON cuentas
FOR EACH ROW
BEGIN
  INSERT INTO log_cuentas (cuenta_id, accion, saldo)
  VALUES (NEW.id, 'ACTUALIZACIÓN', NEW.saldo);
END//
DELIMITER ;

Luego para probar:
UPDATE cuentas SET saldo = saldo + 100 WHERE id = 1;

SELECT * FROM log_cuentas;
```

3. Ejecutar una actualización y verificar el log:

```
UPDATE cuentas SET saldo = saldo + 100 WHERE id = 1;
SELECT * FROM log_cuentas;
```

Ejemplo 3: Validar Saldo antes de una retirada

En este ejemplo, crearemos un trigger que verifique si el saldo es suficiente antes de realizar una retirada:

1. Crear el trigger:

```
DELIMITER //
CREATE TRIGGER before_update_cuentas
BEFORE UPDATE ON cuentas
FOR EACH ROW
BEGIN
  IF NEW.saldo < 0 THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Saldo insuficiente
para la transacción';
  END IF;
END//
DELIMITER ;
```

Explicación:

- El trigger se ejecuta **antes** de actualizar la tabla `cuentas`.
- Si el nuevo saldo (`NEW.saldo`) es menor que cero, lanza un error que detiene la operación.
- El código `SQLSTATE '45000'` es un estado general para errores definidos por el usuario.

2. Probar el trigger:

```
UPDATE cuentas SET saldo = -500 WHERE id = 1;
```

Esto generará un error y no permitirá la operación si el saldo es insuficiente.

13. Otros ejercicios de aplicación en MySQL:

En este ejercicio se creará **un sistema de gestión bancaria** que incluya:

1. Un procedimiento para transferir fondos entre cuentas.
2. Una función para calcular el total de saldo en todas las cuentas.
3. Un trigger que registre cada transacción realizada.

Paso 1: Crear las tablas:

```
CREATE TABLE cuentas (
  id INT PRIMARY KEY,
  titular VARCHAR(50),
  saldo DECIMAL(10, 2)
  fecha_actualizacion TIMESTAMP NULL  -- para el trigger de
actualización automática
);

CREATE TABLE transacciones (
  id INT PRIMARY KEY AUTO_INCREMENT,
  cuenta_origen INT,
  cuenta_destino INT,
  monto DECIMAL(10, 2),
  fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (cuenta_origen) REFERENCES cuentas(id),
  FOREIGN KEY (cuenta_destino) REFERENCES cuentas(id)
);

CREATE TABLE log_transacciones (
  id INT PRIMARY KEY AUTO_INCREMENT,
  cuenta_id INT,
  monto DECIMAL(10, 2),
  fecha TIMESTAMP
);

CREATE TABLE clientes (
  id INT PRIMARY KEY,
  nombre VARCHAR(100)
);

CREATE TABLE clientes_eliminados (
  id INT,
  nombre VARCHAR(100),
  fecha_eliminacion DATETIME
);
```

Paso 2: Insertar datos de prueba:

```
INSERT INTO cuentas (id, titular, saldo) VALUES (1, 'Juan Pérez',
1000), (2, 'Ana Gómez', 1500);

INSERT INTO clientes (id, nombre) VALUES (1, 'Cliente 1'),
(2, 'Cliente 2');
```

Paso 3: Procedimiento para transferir fondos:

```
DELIMITER //
CREATE PROCEDURE TransferirFondos(IN origen INT, IN destino INT, IN
monto DECIMAL(10, 2))
BEGIN
  UPDATE cuentas SET saldo = saldo - monto WHERE id = origen;
  UPDATE cuentas SET saldo = saldo + monto WHERE id = destino;
  INSERT INTO transacciones (cuenta_origen, cuenta_destino, monto)
VALUES (origen, destino, monto);
END //
DELIMITER ;
```

Paso 4: Función para calcular saldo total:

```
DELIMITER //
CREATE FUNCTION SaldoTotal()
RETURNS DECIMAL(10, 2)
BEGIN
  DECLARE total DECIMAL(10, 2);
  SELECT SUM(saldo) INTO total FROM cuentas;
  RETURN total;
END //
DELIMITER ;
```

Paso 5: Trigger para registrar transacciones:

```
DELIMITER //
CREATE TRIGGER registrar_log
AFTER INSERT ON transacciones
FOR EACH ROW
BEGIN
  INSERT INTO log_transacciones (cuenta_id, monto, fecha) VALUES
(NEW.cuenta_origen, NEW.monto, NOW());
END //

DELIMITER ;
```

Paso 6: Trigger para actualizer columnas automáticamente

Supongamos que tenemos una tabla llamada `cuentas` con columnas `id`, `saldo` y `fecha_actualizacion`.

Queremos crear un trigger que actualice automáticamente la columna `fecha_actualizacion` con la fecha actual cada vez que se modifique el saldo.

```
DELIMITER //

CREATE TRIGGER actualizar_fecha_saldo
BEFORE UPDATE ON cuentas
FOR EACH ROW
BEGIN
  SET NEW.fecha_actualizacion = NOW();
END;
//

DELIMITER ;
```

Con este trigger, cada vez que se actualice el saldo de una cuenta, la columna `fecha_actualizacion` se actualizará automáticamente con la fecha y hora actuales.

Paso 7: Trigger para auditar eliminaciones

Supongamos que tenemos una tabla `clientes` y queremos guardar un registro de las eliminaciones en una tabla llamada `clientes_eliminados`.

```
DELIMITER //

CREATE TRIGGER auditar_eliminacion_cliente
AFTER DELETE ON clientes
FOR EACH ROW
BEGIN
  INSERT INTO clientes_eliminados (id, nombre, fecha_eliminacion)
  VALUES (OLD.id, OLD.nombre, NOW());
END;
//


DELIMITER ;
```

Paso 8: Cómo eliminar un trigger

```
DROP TRIGGER nombre_trigger;
```

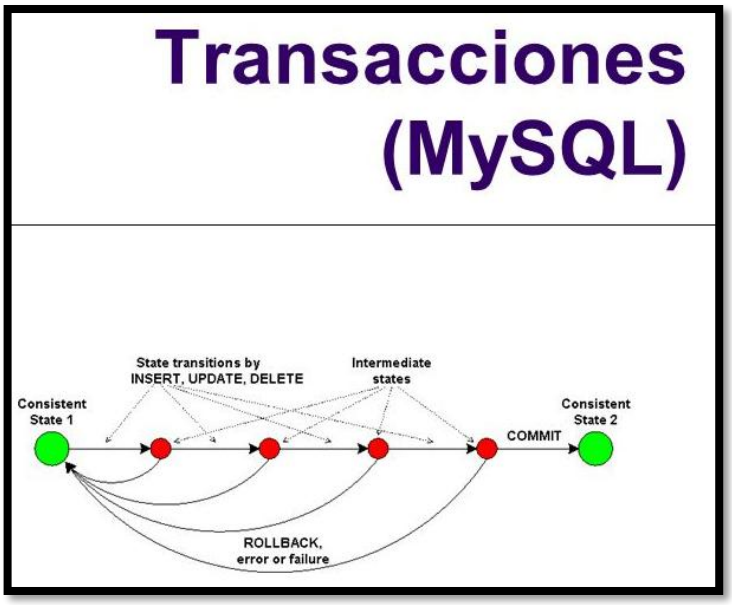
14. Ejercicios propuestos para practicar

- Crear un procedimiento para actualizar los saldos de todas las cuentas con un porcentaje de interés mensual del 2%
- Desarrollar una función que calcule el total de transacciones realizadas por un usuario específico.
- Implementar un trigger que registre en una tabla `log_operaciones` cada vez que se actualice el saldo de una cuenta.

15. Transacciones y bloqueos. Objetivos:

- Comprender el funcionamiento de las transacciones en MySQL.
- Aprender a implementar bloqueos para mantener la integridad de los datos.
- Realizar ejemplos prácticos que demuestren el uso de transacciones y bloqueos en situaciones reales.

16. ¿Qué es una Transacciones?



Una **Transacción** es un *conjunto de operaciones* SQL que se ejecutan de **forma secuencial** y que **deben cumplirse en su totalidad para mantener la integridad de los datos**.

Las **transacciones** permiten agrupar operaciones y asegurarse de que todas ellas se completen correctamente **antes de confirmar los cambios** en la base de datos.

Propiedades ACID:

1. **Atomicidad:** Todas las operaciones deben completarse o ninguna se ejecuta. En caso de error ninguna se ejecuta (todo o nada)
2. **Consistencia:** Los datos deben permanecer válidos antes y después de la transacción. Necesario para mantener las reglas de la integridad.
3. **Aislamiento:** Las operaciones dentro de una transacción deben ser independientes de otras transacciones. No deben interferir con las otras transacciones concurrentes
4. **Durabilidad:** Una vez confirmada la transacción (COMMIT), los cambios son permanentes y no se pierden.

Sintaxis Básica:

```
START TRANSACTION;
```

```
-- Operaciones SQL
COMMIT; -- Confirma los cambios
ROLLBACK; -- Revierte los cambios
```

Ejemplo: Transferir fondos entre dos cuentas:

```
START TRANSACTION;

UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;

COMMIT;
```

Si ocurre un error en alguna de las operaciones, podemos deshacer los cambios utilizando `ROLLBACK`:

```
START TRANSACTION;

UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;
ROLLBACK;
```

17. Bloqueos en MySQL

Los bloqueos (`LOCKS`) permiten controlar el acceso a los datos durante una transacción para evitar que otras transacciones los modifiquen al mismo tiempo. También garantiza integridad y consistencia.

Tipos de Bloqueos:

- **LOCK IN SHARE MODE:** Permite a otras transacciones leer los datos, pero no modificarlos.
- **FOR UPDATE:** Bloquea los datos para actualización, impidiendo que otras transacciones los lean o los modifiquen.

Ejemplo: Bloquear una fila para actualización:

```
START TRANSACTION;

SELECT saldo FROM cuentas WHERE id = 1 FOR UPDATE;

-- Realizar operaciones

COMMIT;
```

18. Ejercicio de aplicación en MySQL

1. **Crear las tablas necesarias:**

```
CREATE TABLE cuentas (
  id INT PRIMARY KEY,
  titular VARCHAR(50),
  saldo DECIMAL(10, 2)
);
```

```
INSERT INTO cuentas (id, titular, saldo) VALUES (1, 'Juan Pérez',
1000), (2, 'Ana Gómez', 1500);
```

2. **Realizar una transacción que transfiera fondos de** `Juan Pérez` a `Ana Gómez`:

```
START TRANSACTION;

UPDATE cuentas SET saldo = saldo - 300 WHERE id = 1;
UPDATE cuentas SET saldo = saldo + 300 WHERE id = 2;

COMMIT;
```

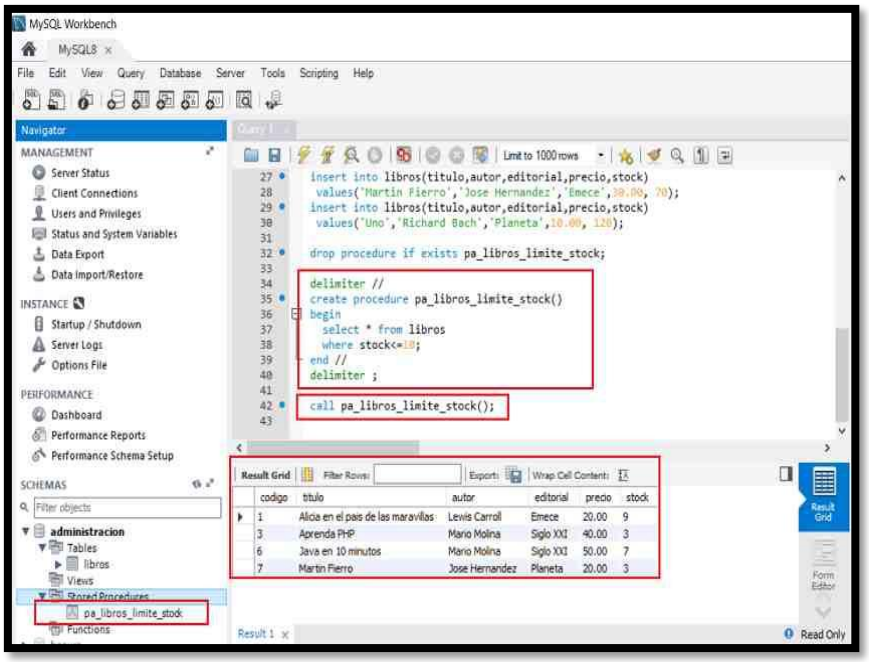
3. **Implementar un bloqueo para evitar modificaciones durante la transacción**:

```
START TRANSACTION;

SELECT saldo FROM cuentas WHERE id = 1 FOR UPDATE;

UPDATE cuentas SET saldo = saldo - 200 WHERE id = 1;
UPDATE cuentas SET saldo = saldo + 200 WHERE id = 2;


COMMIT;
```

19. Ejercicios de aplicación

1. Crear una transacción que realice dos depósitos en dos cuentas diferentes y luego confirme los cambios.
2. Implementar un bloqueo con `LOCK IN SHARE MODE` para evitar que otras transacciones lean los datos mientras se actualizan.
3. Simular una transferencia que falle en la segunda operación y utilice `ROLLBACK` para revertir los cambios.

20. Procedimientos Almacenados y Funciones. Objetivos



- Comprender qué son los procedimientos almacenados y las funciones en MySQL.
- Aprender a crear, ejecutar y modificar procedimientos y funciones.
- Realizar ejemplos prácticos para aplicar estos conceptos en operaciones

comunes de base de datos.

Ventajas:

- Reutilización del código.
- Mejora del rendimiento.
- Seguridad al restringir el acceso a ciertos procesos.

Sintaxis Básica:

```
DELIMITER //
CREATE PROCEDURE nombre_procedimiento()
BEGIN
  -- Operaciones SQL
END//
DELIMITER ;
```

Ejemplo: Crear un procedimiento para listar todas las cuentas:

```
DELIMITER //
CREATE PROCEDURE listarCuentas()
BEGIN
  SELECT * FROM cuentas;
END//
DELIMITER ;

CALL listarCuentas();
```

21. Parámetros en Procedimientos Almacenados:

Los procedimientos pueden aceptar parámetros de entrada (`IN`), salida (`OUT`) o ambos (`INOUT`).

Ejemplo con Parámetros:

```
DELIMITER //
CREATE PROCEDURE transferirFondos(IN origen INT, IN destino INT, IN
monto DECIMAL(10,2))
BEGIN
  UPDATE cuentas SET saldo = saldo - monto WHERE id = origen;
  UPDATE cuentas SET saldo = saldo + monto WHERE id = destino;
END//
DELIMITER ;

CALL transferirFondos(1, 2, 500);
```

22. Funciones en MySQL

Sintaxis Básica:

```
DELIMITER //
CREATE FUNCTION nombre_funcion(parametros)
RETURNS tipo_de_dato
DETERMINISTIC
BEGIN
```

```
  RETURN valor;
END//
DELIMITER ;
```

Ejemplo: Crear una función para obtener el saldo de una cuenta:

```
DELIMITER //
CREATE FUNCTION obtenerSaldo(cuenta_id INT)
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
  DECLARE saldo DECIMAL(10,2);
  SELECT saldo INTO saldo FROM cuentas WHERE id = cuenta_id;
  RETURN saldo;
END//
DELIMITER ;

SELECT obtenerSaldo(1);
```

Ejercicio de aplicación en MySQL

1. Crear un procedimiento para realizar una transferencia de fondos con verificación de saldo:

```
DELIMITER //
CREATE PROCEDURE transferirVerificado(IN origen INT, IN destino INT,
IN monto DECIMAL(10,2))
BEGIN
  DECLARE saldo_origen DECIMAL(10,2);
  SELECT saldo INTO saldo_origen FROM cuentas WHERE id = origen;

  IF saldo_origen >= monto THEN
    UPDATE cuentas SET saldo = saldo - monto WHERE id = origen;
    UPDATE cuentas SET saldo = saldo + monto WHERE id = destino;
  ELSE
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Fondos insuficientes';
  END IF;
END//
DELIMITER ;

CALL transferirVerificado(1, 2, 300);
```

2. Crear una función que devuelva la suma total del saldo de todas las cuentas:

```
DELIMITER //
CREATE FUNCTION totalSaldo()
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
  DECLARE total DECIMAL(10,2);
  SELECT SUM(saldo) INTO total FROM cuentas;
  RETURN total;
END//
DELIMITER ;

SELECT totalSaldo();
```

23. Ejercicios de aplicación propuestos

1. Crear un procedimiento que acepte un ID de cuenta y un monto a depositar. Verificar que el monto sea positivo antes de realizar el depósito.
2. Crear una función que reciba el ID de una cuenta y devuelva el titular de la cuenta.
3. Modificar el procedimiento `transferirVerificado` para incluir un parámetro `OUT` que devuelva un mensaje de confirmación o error.

24. Control de Concurrencia

Para evitar problemas de concurrencia, MySQL implementa diferentes niveles de aislamiento:

- **READ UNCOMMITTED** – Permite leer datos no confirmados.
- **READ COMMITTED** – Solo permite leer datos confirmados.
- **REPEATABLE READ** – Mantiene los datos consistentes durante una transacción.
- **SERIALIZABLE** – Aísla completamente las transacciones.

Para establecer un nivel de aislamiento:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

25. Ejercicio de aplicación en MySQL

1. Crear un escenario donde dos usuarios intenten transferir fondos simultáneamente:

```
START TRANSACTION;
  UPDATE cuentas SET saldo = saldo - 300 WHERE id = 1;
  -- Simular un retraso intencionado
  SELECT SLEEP(5);
  UPDATE cuentas SET saldo = saldo + 300 WHERE id = 2;
COMMIT;
```

2. Aplicar un bloqueo para evitar que un tercero modifique los saldos durante la transferencia:

```
LOCK TABLES cuentas WRITE;
  UPDATE cuentas SET saldo = saldo - 300 WHERE id = 1;
  UPDATE cuentas SET saldo = saldo + 300 WHERE id = 2;
UNLOCK TABLES;
```

26. Otros ejercicios:

1. Crear una transacción que realice dos depósitos y luego se confirme o revierta según el saldo final.
2. Implementar un bloqueo para impedir que se realicen retiros durante una actualización masiva de saldos.

3. Cambiar el nivel de aislamiento a `SERIALIZABLE` y observar el impacto en la ejecución concurrente de transacciones.

27. Ejercicio completo

1. Crear una base de datos llamada `empresa`.

```
CREATE DATABASE empresa;
```

Explicación:

- `CREATE DATABASE empresa;` — Crea una nueva base de datos llamada `empresa`.

2. Creación de Tablas: Crear una tabla llamada `empleados` con columnas: `id` (entero, clave primaria), `nombre` (texto), `apellido` (texto) y `salario` (decimal).

```
CREATE TABLE empleados (
  id INT PRIMARY KEY AUTO_INCREMENT,
  nombre VARCHAR(50),
  apellido VARCHAR(50),
  salario DECIMAL(10,2)
  fecha_modificacion DATETIME  -- para trigger posterior
);
```

Explicación:

- `CREATE TABLE empleados (...)` — Crea una tabla llamada `empleados`.
- `id INT PRIMARY KEY` — Columna `id` de tipo entero que será la clave primaria (única y no nula).
- `nombre VARCHAR(50)` — Columna para el nombre, texto hasta 50 caracteres.
- `apellido VARCHAR(50)` — Columna para el apellido, texto hasta 50 caracteres.
- `salario DECIMAL(10,2)` — Columna para el salario con hasta 10 dígitos y 2 decimales.
- Fecha de modificación — Tipo DATATIME para futuro Trigger

3. Inserción de Datos: Insertar tres empleados en la tabla `empleados`.

```
INSERT INTO empleados (id, nombre, apellido, salario) VALUES
(1, 'Juan', 'Pérez', 3000.00),
(2, 'Ana', 'Gómez', 3500.50),
(3, 'Luis', 'Martínez', 2800.75);
```

Explicación:

- `INSERT INTO empleados (...) VALUES (...)` — Inserta datos en la tabla.

- Cada conjunto entre paréntesis representa una fila con los valores para cada columna en el orden declarado.

4. Consultas Básicas: Obtener todos los empleados cuyo salario es mayor a 3000.

```
SELECT * FROM empleados WHERE salario > 3000;
```

Explicación:

- `SELECT *` — Selecciona todas las columnas.
- `FROM empleados` — De la tabla `empleados`.
- `WHERE salario > 3000` — Solo filas donde el salario es mayor a 3000.

5. Actualización de Datos: Aumentar el salario de Ana Gómez a 4000.

```
UPDATE empleados SET salario = 4000 WHERE nombre = 'Ana' AND apellido
= 'Gómez';
```

Explicación:

- `UPDATE empleados` — Indica que se modificará la tabla empleados.
- `SET salario = 4000` — Cambia el salario a 4000.
- `WHERE nombre = 'Ana' AND apellido = 'Gómez'` — Solo para la fila donde el nombre y apellido coinciden.

6. Eliminación de Datos: Eliminar al empleado con id 3.

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

Explicación:

- `DELETE FROM empleados` — Elimina filas de la tabla empleados.
- `WHERE id = 3` — Solo la fila donde `id` es 3.

7. Relaciones entre Tablas: Crear tabla `departamentos` y relacionarla con `empleados`.

```
CREATE TABLE departamentos (
  id INT PRIMARY KEY,
  nombre VARCHAR(50)
);

ALTER TABLE empleados ADD COLUMN departamento_id INT;
```

```
ALTER TABLE empleados ADD CONSTRAINT fk_departamento FOREIGN KEY
(departamento_id) REFERENCES departamentos(id);
```

Explicación:

- Crea tabla `departamentos` con id y nombre.
- En `empleados` agrega columna `departamento_id` para relación.
- `FOREIGN KEY` asegura que el valor en `departamento_id` debe existir en `departamentos`.

8. Procedimientos Almacenados: Crear un procedimiento para aumentar el salario de un empleado dado su id.

```
DELIMITER //

CREATE PROCEDURE aumentarSalario(IN emp_id INT, IN aumento
DECIMAL(10,2))
BEGIN
  UPDATE empleados SET salario = salario + aumento WHERE id = emp_id;
END//

DELIMITER ;

CALL aumentarSalario(1, 500);
```

Explicación:

- `CREATE PROCEDURE` define procedimiento con parámetros.
- `UPDATE` suma aumento al salario donde `id` coincide.
- `CALL` ejecuta el procedimiento.

9. Funciones: Crear una función que devuelva el salario promedio de los empleados.

```
DELIMITER //

CREATE FUNCTION salarioPromedio()
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
  DECLARE promedio DECIMAL(10,2);
  SELECT AVG(salario) INTO promedio FROM empleados;
  RETURN promedio;
END//

DELIMITER ;

SELECT salarioPromedio();
```

Explicación:

- `CREATE FUNCTION` define función que retorna un valor.
- `SELECT AVG(salario)` calcula promedio.
- `RETURN` devuelve el promedio calculado.

10. Triggers: Crear un trigger que actualice la fecha de última modificación al cambiar el salario.

```
DELIMITER //

CREATE TRIGGER actualizar_fecha_modificacion
BEFORE UPDATE ON empleados
FOR EACH ROW
BEGIN
  SET NEW.fecha_modificacion = NOW();
END//

DELIMITER ;
```

Explicación:

- `CREATE TRIGGER` crea un disparador.
- `BEFORE UPDATE` ejecuta antes de actualizar un registro.
- `SET NEW.fecha_modificacion = NOW()` asigna fecha y hora actual.

Nota: Para este ejercicio, la tabla `empleados` debe tener una columna `fecha_modificacion` de tipo `DATETIME`.