Tema 12

Transacciones.

Transacción: conjunto de tareas relacionadas que o se realizan de forma satisfactoria como una unidad o fallan en conjunto.

Se consideran una unidad lógica de trabajo (LUW del inglés *Logic Unit of Work*).

En términos de procesamiento, las transacciones se confirman o se anulan (se vuelve al estado inicial).

Las transacciones tienes que cumplir con cuatro propiedades conocidas por las siglas ACID:

- Atomicidad (Atomicity): una transacción es indivisible, se realiza en su totalidad o no se realiza.
- Coherencia o consistencia (Consistency): antes de la transacción los datos están en un estado coherente. Al terminar la transacción deben estarlo también: en el momento en que se confirme la transacción (`COMMIT`), las operaciones de la transacción no deben violar ninguna restricción de la base de datos (claves primarias, claves únicas, claves externas, comprobaciones, etc.).
- Aislamiento (Isolation): para cada transacción, debe parecer que es la única que está operando sobre los datos en ese momento. Cuando hay varias transacciones concurrentes, el estado intermedio de una transacción no debe ser visto desde operaciones de otra. No se satisfice en la mayoría de los SGBD. Sin embargo debe ser considerado por los desarrolladores de aplicaciones.
- Permanencia (Durability): una vez se ha completado una transacción, los datos permanecerán en el sistema. Los SGBD modernos tienen mecanismos para asegurar la completitud de las operaciones incluso ante caídas del sistema.

En una transacción hay dos operaciones básicas:

- `COMMIT`: hace que los cambios realizados por la transacción sean definitivos e irrevocables. También se denomina **confirmar** la transacción. No entraremos en detalles pero hay muchas operaciones que realizan un `COMMIT` implícitamente.
- `ROLLBACK`: regresa a la situación anterior al inicio de la transacción. Anula definitivamente los cambios y se conoce también como **anular o revertir** la transacción.

IMPORTANTE:

Dependiendo del SGBD que estemos utilizando, un cierre de sesión con una transacción abierta puede provocar que se realice un `COMMIT` o `ROLLBACK` implícito sobre la misma. Por este motivo es muy importante realizar un control correcto de las transacciones.

Para poder usarlas en MySQL es necesario utilizar el motor InnoDB.

12.1. `AUTOCOMMIT`.

Las operaciones que hemos estado realizando, se ejecutaban inmediatamente porque el SGBD tiene una opción que se llama `AUTOCOMMIT` y está activa. Esto significa que cada operación se confirma inmediatamente (cada operación es atómica).

La implicación es que no podríamos hacer transacciones que engloben varias operaciones.

En MySQL podemos consultar su estado con:

```
SELECT @@AUTOCOMMIT;
```

En Oracle se haría con:

```
SHOW AUTOCOMMIT;
```

12.1.1. `AUTOCOMMIT` en Mysql.

En MySQL, una sesión comienza siempre con `AUTOCOMMIT` activo.

Para realizar una transacción que comprenda varias operaciones tenemos varias opciones:

- Indicar explícitamente que estamos realizando una transacción:

```
START TRANSACTION;
UPDATE empleados SET salario = 3000 WHERE id_empleado = 302;
UPDATE empleados SET comision = 0.15 WHERE id_empleado = 302;
COMMIT;
```

  Así `AUTOCOMMIT` seguirá activo.

- Desactivar `AUTOCOMMIT` para toda la sesión:

```
SET AUTOCOMMIT = 0;
```

  Podemos volver a activarlo para la sesión actual poniendo el valor a 1.

```
SET AUTOCOMMIT = 1;
```

En MySQL la sintaxis es:

```
START TRANSACTION
[transaction_characteristic [, transaction_characteristic] ... ]

transaction_characteristic: {
   WITH CONSISTENT SNAPSHOT
   | READ WRITE
   | READ ONLY
}
```

NOTA:

`BEGIN` y `BEGIN WORK` son sinónimos de `START TRANSACTION` en MySQL pero la última es estándar SQL y por eso es la que debemos usar.

En MySQL algunas sentencias realizan un `COMMIT` implícito.

12.1.2. Ejemplos.

12.1.2.1. Ejemplo 1.

```
CREATE TABLE prueba_tran (FECHA DATE);

START TRANSACTION;

INSERT INTO prueba_tran VALUES ('2023/01/15');
INSERT INTO prueba_tran VALUES ('2023/02/23');
INSERT INTO prueba_tran VALUES ('2023/03/07');
INSERT INTO prueba_tran VALUES ('2023/04/14');
INSERT INTO prueba_tran VALUES ('2023/05/01');

ROLLBACK;
```

aunque el uso de transacciones con `ROLLBACK` explícito tienen más sentido pudiendo introducir lógica con algún lenguaje de programación.

12.1.2.2. Ejemplo 2.

En el siguiente usamos una variable, un tema en el que entraremos más adelante.

```
START TRANSACTION;
SELECT @A:=SUM(salary) FROM table1 WHERE type=1;
UPDATE table2 SET summary=@A WHERE type=1;
COMMIT;
```

12.1.2.3. Ejemplo 3.

```
DROP DATABASE IF EXISTS test;
CREATE DATABASE test CHARACTER SET utf8mb4;
USE test;

CREATE TABLE cliente (
   id INT UNSIGNED PRIMARY KEY,
   nombre CHAR (20)
);
```



```
START TRANSACTION;
INSERT INTO cliente VALUES (1, 'Pepe');
COMMIT;
```

¿Qué devolverá esta consulta?

```
SELECT * FROM cliente;
```

¿Qué devolverá esta consulta?

```
SET AUTOCOMMIT=0;
INSERT INTO cliente VALUES (2, 'Maria');
INSERT INTO cliente VALUES (20, 'Juan');
DELETE FROM cliente WHERE nombre = 'Pepe';

SELECT * FROM cliente;
```

¿Qué devolverá esta consulta?

```
ROLLBACK;

SELECT * FROM cliente;
```

12.1.2.4. Ejemplo 4.

```
DROP DATABASE IF EXISTS test;
CREATE DATABASE test CHARACTER SET utf8mb4;
USE test;

CREATE TABLE cuentas (
   id INTEGER UNSIGNED PRIMARY KEY,
   saldo DECIMAL(11,2) CHECK (saldo >= 0)
);

INSERT INTO cuentas VALUES (1, 1000);
INSERT INTO cuentas VALUES (2, 2000);
INSERT INTO cuentas VALUES (3, 0);
```


Consultamos el estado actual de las cuentas

```
SELECT * FROM cuentas;
```

Supón que queremos realizar una transferencia de dinero entre dos cuentas bancarias con la siguiente transacción:

```
START TRANSACTION;
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;
COMMIT;
```

¿Qué devolverá esta consulta?

```
SELECT * FROM cuentas;
```

Supón que queremos realizar una transferencia de dinero entre dos cuentas bancarias con la siguiente transacción y una de las dos cuentas no existe:

```
START TRANSACTION;
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 9999;
UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;
COMMIT;
```

¿Qué devolverá esta consulta?

```
SELECT * FROM cuentas;
```

Supón que queremos realizar una transferencia de dinero entre dos cuentas bancarias con la siguiente transacción y la cuenta origen no tiene saldo:

```
START TRANSACTION;
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 3;
UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;
COMMIT;
```

¿Qué devolverá esta consulta?


```
SELECT * FROM cuentas;
```

12.2. Puntos de guardado.

Podemos establecer puntos intermedios dentro de una transacción para poder realizar `ROLLBACK` hasta ellos en lugar de hasta el principio de la transacción.

Para definir el punto de guardado:

```
SAVEPOINT identificador;
```

Si queremos volver a él:

```
ROLLBACK [WORK] TO [SAVEPOINT] identificador;
```

Es posible liberar el punto de guardado antes de que termine la transacción:

```
RELEASE SAVEPOINT identificador;
```

12.2.1. Ejemplos.

12.2.1.1. Ejemplo 1.

```
CREATE TABLE prueba_tran (FECHA DATE);

START TRANSACTION;

INSERT INTO prueba_tran VALUES ('2023/01/15');
INSERT INTO prueba_tran VALUES ('2023/02/23');

SAVEPOINT febrero;

INSERT INTO prueba_tran VALUES ('2023/03/07');
INSERT INTO prueba_tran VALUES ('2023/04/14');

SAVEPOINT abril;

INSERT INTO prueba_tran VALUES ('2023/05/01');

ROLLBACK TO febrero;
COMMIT;
```

mantendría únicamente las dos primeras inserciones.

12.2.1.2. Ejemplo 2.

```
DROP DATABASE IF EXISTS test;
CREATE DATABASE test CHARACTER SET utf8mb4;
USE test;

CREATE TABLE producto (
   id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
   nombre VARCHAR(100) NOT NULL,
   precio DOUBLE
);

INSERT INTO producto (id, nombre) VALUES (1, 'Primero');
INSERT INTO producto (id, nombre) VALUES (2, 'Segundo');
INSERT INTO producto (id, nombre) VALUES (3, 'Tercero');
```


Comprobamos las filas que existen en la tabla:

```
SELECT * FROM producto;
```

Ejecutamos una transacción que incluye un `SAVEPOINT`:

```
START TRANSACTION;
INSERT INTO producto (id, nombre) VALUES (4, 'Cuarto');
SAVEPOINT sp1;
INSERT INTO producto (id, nombre) VALUES (5, 'Cinco');
INSERT INTO producto (id, nombre) VALUES (6, 'Seis');
ROLLBACK TO sp1;
```

¿Qué devolverá esta consulta?

```
SELECT * FROM producto;
```

Ejercicios 12.1

Realiza los ejercicios sobre una base de datos `prueba`, que luego elimines.

1. Crea una tabla `mascotas` que lleve `id_mascota` (`INT`), `nombre` (`VARCHAR(30)`), `edad` (`SMALLINT`).

   Añade tres mascotas (ids 1, 2 y 3) a la tabla y comprueba que se han insertado.

   Deshaz las inserciones con `ROLLBACK`. ¿Están las mascotas todavía en la tabla? ¿por qué?

   Repite las operaciones de este ejercicio de tal forma que al realizar el `ROLLBACK` las mascotas no aparezcan en nuestra tabla.

2. Justo a continuación de las operaciones del ejercicio anterior, ejecuta las siguientes operaciones:

   ```
   INSERT INTO mascotas (id_mascota, nombre, edad)
      VALUES (5, 'Micifú', 8);
   ROLLBACK;
   SELECT * FROM mascotas;
   ```

   ¿se realiza el `ROLLBACK`? ¿por qué? ¿qué dos opciones hay para solucionarlo?

3. Sobre lo hecho en los ejercicios anteriores realiza las siguientes operaciones:


```
SET AUTOCOMMIT=0;
INSERT INTO mascotas (id_mascota, nombre, edad)
   VALUES (8, 'Calcetines', 6);
CREATE TABLE lista (id INT);
INSERT INTO lista (id) VALUES (1);
SELECT * FROM lista;
ROLLBACK;
```

¿Qué ha sucedido con la tabla `mascotas` y sus datos?

¿Qué ha sucedido con la tabla `lista` y sus datos?

Explica los resultados.

4. Ahora realiza las siguientes operaciones, ejecutándolas una a una para ver qué está sucediendo en cada caso.

   Vuelve a ejecutar las operaciones pero ahora en bloque y compara los resultados:

```
SET AUTOCOMMIT=0;
INSERT INTO mascotas (id_mascota, nombre, edad)
   VALUES (10, 'Un nombre demasiado largo para mascotas', 2);
-- Actalizamos una tupla inexistente
UPDATE mascotas SET nombre = 'Fantasma'
   WHERE id_mascota = 333;
-- Borramos una tupla inexistente
DELETE FROM mascotas WHERE id_mascota = 444;

-- Insertamos con el id que usamos antes.
INSERT INTO mascotas (id_mascota, nombre, edad)
    VALUES (10, 'Yummy', 3);
-- Desbordamos el vampo edad
INSERT INTO mascotas (id_mascota, nombre, edad)
   VALUES (11, 'Tom', 32769);
-- ¿Sigue activa la transacción
INSERT INTO mascotas (id_mascota, nombre, edad)
   VALUES (12, 'Peti', 8);
SELECT * FROM mascotas;
COMMIT;
```

¿Cómo se comporta MySQL ante errores en una transacción? ¿afecta el error a las siguientes operaciones? ¿y a las anteriores? Si no lo tienes claro, realiza modificaciones sobre las operaciones anteriores.

Introduce puntos de guardado antes de cada `INSERT`. ¿Cambia lo que sucede?

5. Crea la siguiente tabla:

```
CREATE TABLE cuentas (
id_operacion INTEGER NOT NULL PRIMARY KEY,
saldo INTEGER NOT NULL CONSTRAINT no_num_rojos
   CHECK (saldo ⩾ 0)
);
```

Realiza las siguientes operaciones y comenta los resultados:

```
SET AUTOCOMMIT=0;
INSERT INTO cuentas (id_operacion, saldo) VALUES (101, 1000);
INSERT INTO cuentas (id_operacion, saldo) VALUES (202, 2000);
SELECT * FROM cuentas;
COMMIT;
```

```
-- intentemos hacer una transaferencia
UPDATE cuentas SET saldo = saldo - 100
   WHERE id_operacion = 101;
UPDATE cuentas SET saldo = saldo + 100
   WHERE id_operacion = 202;
SELECT * FROM cuentas;
ROLLBACK;
```

```
-- Comprobemos si funciona el uso de CHECK:
UPDATE cuentas SET saldo = saldo - 2000
   WHERE id_operacion = 101;
UPDATE cuentas SET saldo = saldo + 2000
   WHERE id_operacion = 202;
SELECT * FROM cuentas ;
ROLLBACK;
```



```
-- Transferencia a cuenta que no existe
UPDATE cuentas SET saldo = saldo - 500
   WHERE id_operacion = 101;
UPDATE cuentas SET saldo = saldo + 500
   WHERE id_operacion = 777;
SELECT * FROM cuentas ;
ROLLBACK;
```

Sustituye los `ROLLBACK` por `COMMIT` y vuelve a comentar los resultados, comparándolos con los obtenidos con `ROLLBACK`.

12.3. Concurrencia de transacciones.

Puede producir varios problemas:

- Actualización perdida (*Lost Update*): dos transacciones están actualizando los mismos datos en paralelo y la segunda sobrescribe una actualización de datos de la primera antes de que haya sido confirmada (`COMMIT`).

  Todos los SGBD modernos implementan algún mecanismo de control de la concurrencia que protege las operaciones de escritura de ser sobrescritas por transacciones concurrentes antes del final de la transacción.

  En el ejemplo vamos a suponer que dos personas sacan dinero de la misma cuenta a la vez. (Figura 12.1)
- Lectura sucia (*Dirty Read*): si una transacción lee datos que otra está modificando unos datos y no ha terminado. (Figura 12.2)
- Lectura no repetible (*Non-repeateable Read*): una transacción lee los datos que otra modifica, antes (o durante) y después de la modificación, por lo que ante la misma operación recibirá información diferente. (Figura 12.3)
- Lectura fantasma (*Phantom Read*): una transacción ejecuta dos veces la misma consulta pero obtiene resultados diferentes porque entre ambas lecturas, otra transacción ha insertado o eliminado filas. (Figura 12.4)

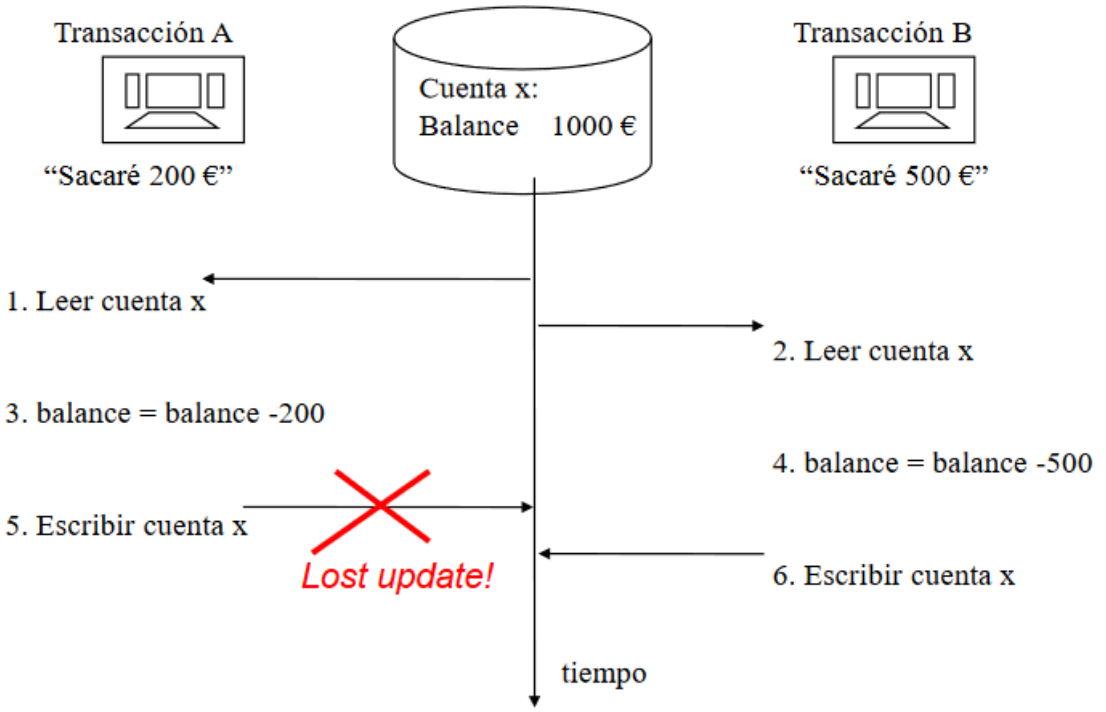

Figura 12.1: Actualización perdida. Imagen obtenida aquí

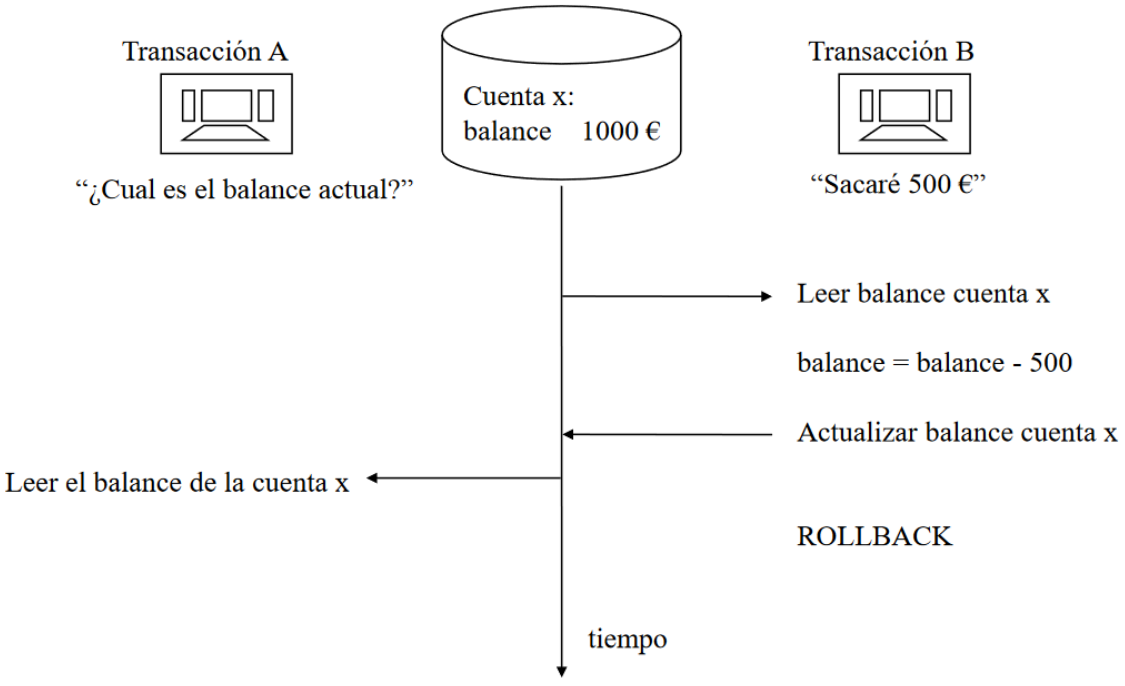

Figura 12.2: Lectura sucia. Imagen obtenida aquí

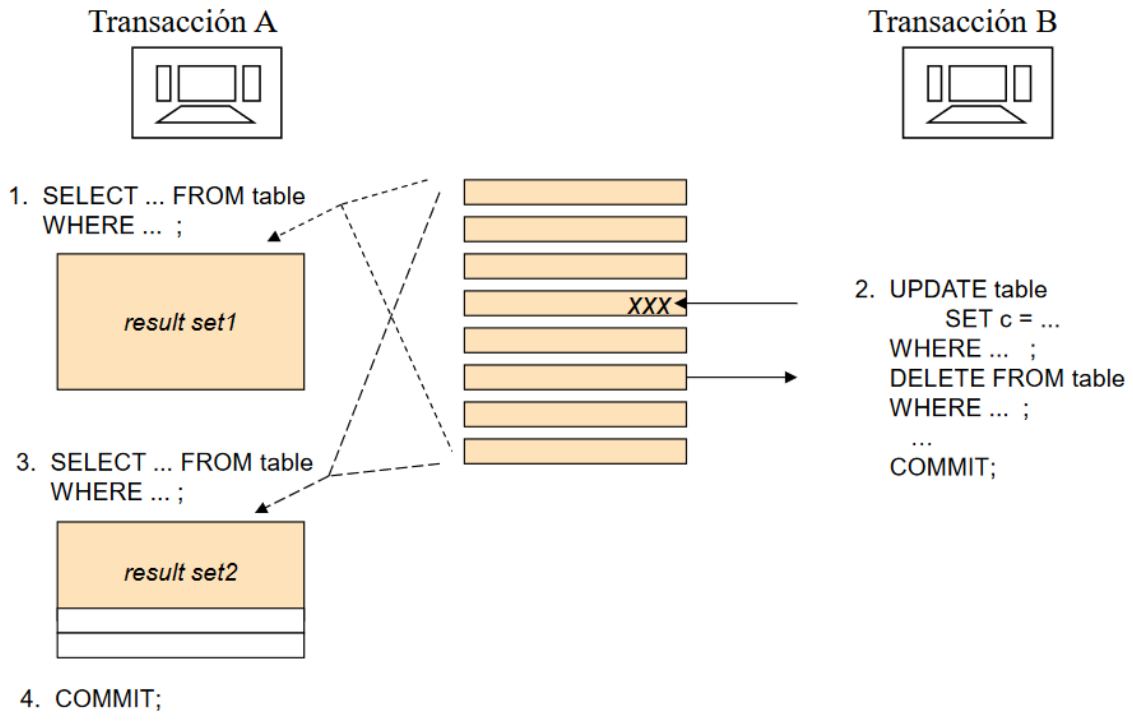

Figura 12.3: Lectura no repetible. Imagen obtenida aquí

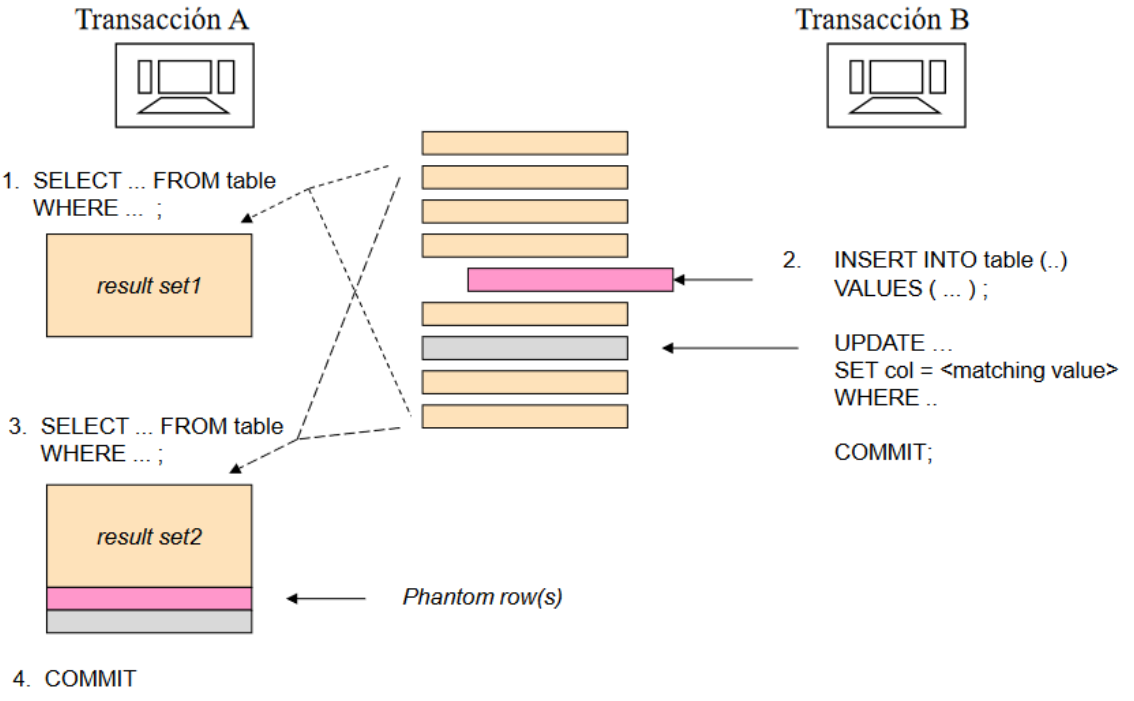

Figura 12.4: Lectura fantasma. Imagen obtenida aquí

12.3.1. Ejemplos.

No veremos ejemplo práctica de actualización perdida porque los SGBD modernos las evitan.

12.3.1.1. Lectura sucia.

```
/* dos transacciones, izquierda y derecha */

START TRANSACTION;
UPDATE empleados SET salario = 5000 WHERE id = 302;
-- sin commit aún


                                          START TRANSACTION;
                    SELECT salario FROM empleados WHERE id = 302;
                           -- lectura sucia, estaría leyendo 5000
                                                        COMMIT;


ROLLBACK;
```

12.3.1.2. Lectura no repetible.

```
/* dos transacciones, izquierda y derecha */

START TRANSACTION;
SELECT salario FROM empleados WHERE id = 302;


                                          START TRANSACTION;
              UPDATE empleados SET salario = 5000 WHERE id = 302;
                                                        COMMIT;


SELECT salario FROM empleados WHERE id = 302;
-- misma operación y transacción, resultado diferente
COMMIT;
```


12.3.1.3. Lectura fantasma.

```
/* dos transacciones, izquierda y derecha */

START TRANSACTION;
SELECT nombre_departamento FROM departamentos;



                                                START TRANSACTION;
                                            INSERT INTO departamentos
                                VALUES (90, 'Envolver regalos', NULL, 1000);
                                                                  COMMIT;


SELECT nombre_departamento FROM departamentos;
-- misma operación y transacción, resultado diferente
COMMIT;
```

12.3.2. Aislamiento de transacciones concurrentes.

La solución pasa por bloquear datos. Esto significa hacer que determinados datos no estén disponibles para realizar algunas operaciones (lectura, escritura, borrado y/o modificación) sobre ellos durante un periodo de tiempo (por ejemplo hasta que termine la transacción).

12.3.2.1. Políticas de bloqueo.

Hay varias políticas de bloqueo diferentes:

1. La forma más fácil sería bloquear la BdD (el esquema) completa, sin embargo el funcionamiento sería muy lento porque cualquier operación tendría que esperar a que terminara la anterior.
2. La siguiente sería bloquear una tabla (o tablas, según la operación). Esto solo afectaría al resto de operaciones sobre dicha tabla.
3. Si bloqueamos un conjunto de filas tenemos el concepto de bloqueo de rango (*range lock*).
4. Bloquear solo la fila afectada, sin afectar a operaciones sobre el resto.
5. Bloquear solo la columna afectada.

InnoDB utiliza por defecto el bloqueo a nivel de fila.

12.3.2.2. Ejemplo.

Podemos probarlo usando dos sesiones diferentes. Podríamos abrir dos terminales con conexiones a MySQL o dos sesiones en WorkBench.

Para ello necesitamos poder abrir más de una ventana de Workbench (solo es posible en Windows):

1. *Edit –> Preferences –> Others –> Allow more than one instance of MySQL Workbench to run*.
2. Marca la casilla y acepta los cambios.
3. Abre otra ventana de Workbench.

NOTA:

Este ejemplo está basado en uno de José Juan Sánchez Hernández al que agradezco que comparta su trabajo.

Vamos a demostrar que los SGBD modernos no permiten el problema de actualización perdida.

Usuario 1:


```sql
DROP DATABASE IF EXISTS test;
CREATE DATABASE test CHARACTER SET utf8mb4;
USE test;

CREATE TABLE cuentas (
id INTEGER UNSIGNED PRIMARY KEY,
saldo DECIMAL(11,2) CHECK (saldo >= 0)
);

INSERT INTO cuentas VALUES (1, 1000);
INSERT INTO cuentas VALUES (2, 2000);
INSERT INTO cuentas VALUES (3, 0);

-- 1. Ejecutamos una transacción para transfereir dinero entre dos
--    cuentas
START TRANSACTION;
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
```

Usuario 2:

```sql
-- 1. Seleccionamos la base de datos
USE test;

-- 2. Observamos los datos que existen en la tabla cuentas
SELECT * FROM cuentas;

-- 3. Intentamos actualizar el saldo de una de las cuentas que está
--    siendo utilizada en la transacción del Usuario 1
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;

-- 4. Finalizamos la transacción con COMMIT
COMMIT;

-- 5. Observamos los datos que existen en la tabla cuentas
SELECT * FROM cuentas;
```

Usuario 1:


```
-- 4. Finalizamos la transacción con COMMIT
COMMIT;

-- 5. Observamos los datos que existen en la tabla cuentas
SELECT * FROM cuentas;
```

12.3.2.3. Niveles de aislamiento.

El estándar ANSI/ISO de SQL (SQL92) define cuatro niveles de aislamiento:

1. Lectura no confirmada (*Read Uncommitted*): es el nivel más bajo. No se realiza ningún bloqueo, por lo tanto, permite que se den los tres problemas.
2. Lectura confirmada (*Read Committed*): No permite lecturas de datos sin confirmar de transacciones concurrentes. Los datos leídos por una transacción pueden ser modificados por otras transacciones, por lo tanto, se pueden dar los problemas *Non-Repeteable Read* y *Phantom Read*. Algunos SGBD leen siempre los últimos datos confirmados mientras que otros retrasan las lecturas hasta que se confirmen los datos. Evita *Dirty Read* porque solo se puede leer los datos una vez han sido confirmados.
3. Lectura reproducible (*Repeatable Read*): ningún registro leído con un `SELECT` puede ser modificado en otra transacción, por lo tanto, sólo puede suceder el problema del *Phantom Read* debido a que no se gestionan los bloqueos de rango.
4. *Serializable*: En este caso las transacciones se ejecutan unas detrás de otras, sin que exista la posibilidad de concurencia. No podría darse ninguno de los problemas.

Podemos ver una tabla resumen: (Figura 12.5)

Read phenomenon / Isolation level	Dirty read	Non-repeatable read	Phantom read
Serializable	no	no	no
Repeatable read	no	no	yes
Read committed	no	yes	yes
Read uncommitted	yes	yes	yes

Figura 12.5: Niveles y problemas de lectura. Imagen obtenida aquí

12.3.3. Ejemplos.

Sobre los ejemplos anteriores podemos ver qué sucedería en cada caso.

12.3.3.1. Lectura sucia.

```
/* dos transacciones, izquierda y derecha */

START TRANSACTION;
UPDATE empleados SET salario = 5000 WHERE id = 302;
-- sin commit aún


                                                START TRANSACTION;
                                SELECT salario FROM empleados WHERE id = 302;
                                   -- READ UNCOMMITTED produce lectura sucia.
                             -- READ COMMITTED, REPEATABLE READ y SERIALIZABLE
                                                --   evitan lectura sucia.
                                                                    COMMIT;


ROLLBACK;
```

12.3.3.2. Lectura no repetible.

```
/* dos transacciones, izquierda y derecha */

START TRANSACTION;
SELECT salario FROM empleados WHERE id = 302;


                                                START TRANSACTION;
                UPDATE empleados SET salario = 5000 WHERE id = 302;
                                                                COMMIT;


SELECT salario FROM empleados WHERE id = 302;
-- READ UNCOMMITTED y READ COMMITTED
--  producen lectura no reproducible.
-- REPEATABLE READ y SERIALIZABLE la evitan
COMMIT;
```

12.3.3.3. Lectura fantasma.

```
/* dos transacciones, izquierda y derecha */

START TRANSACTION;
SELECT nombre_departamento FROM departamentos;


                                                START TRANSACTION;
                                        INSERT INTO departamentos
                        VALUES (90, 'Envolver regalos', NULL, 1000);
                                                                COMMIT;


SELECT nombre_departamento FROM departamentos;
-- READ UNCOMMITTED, READ COMMITTED y REPEATABLE READ
--  producen lectura fantasma.
-- SERIALIZABLE la evita.
COMMIT;
```


12.3.3.4. Niveles de aislamiento en MySQL.

Para consultar el valor de la variable global:

```
SELECT @@GLOBAL.transaction_isolation;
```

Aunque el importante es el establecido para nuestra sesión, que inicialmente toma el valor de la global.

```
SELECT @@SESSION.transaction_isolation;
```

Para modificarlo, usaremos alguna de las siguientes líneas:

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

12.3.4. Ejemplos.

NOTA:

Estos ejemplos están basados en otros del material de José Juan Sánchez Hernández al que agradezco que comparta su trabajo.

12.3.4.1. Ejemplo: niveles de aislamiento ante el problema *Dirty Read*.

En este ejemplo vamos a simular que hay dos usuarios que quieren acceder de forma concurrente a los mismos datos de una tabla.

Usuario 1:

```sql
DROP DATABASE IF EXISTS test;
CREATE DATABASE test CHARACTER SET utf8mb4;
USE test;

CREATE TABLE cuentas (
   id INTEGER UNSIGNED PRIMARY KEY,
   saldo DECIMAL(11,2) CHECK (saldo >= 0)
);

INSERT INTO cuentas VALUES (1, 1000);
INSERT INTO cuentas VALUES (2, 2000);
INSERT INTO cuentas VALUES (3, 0);

-- 1. Configuramos que en esta sesión vamos a utilizar el nivel de
--    aislamiento READ UNCOMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- 2. Ejecutamos una transacción para transfereir dinero entre dos
--    cuentas
START TRANSACTION;
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
```

NOTA: Observa que la transacción que está ejecutando el Usuario 1 todavía no ha finalizado, porque no ha ejecutado `COMMIT` ni `ROLLBACK`.

Usuario 2:


```
-- 1. Seleccionamos la base de datos
USE test;

-- 2. Configuramos que en esta sesión vamos a utilizar el nivel de
--    aislamiento READ UNCOMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- 3. Iniciamos una transacción y observamos los datos que existen
--    en la tabla cuentas
START TRANSACTION;
SELECT * FROM cuentas WHERE id = 1;
```

Usuario 1:

```
-- 3. Deshacemos las operaciones realizadas en la transacción
ROLLBACK;
```

Usuario 2:

```
-- 4. Observamos los datos que existen en la tabla cuentas
SELECT * FROM cuentas WHERE id = 1;
COMMIT;
```

¿Qué es lo que ha sucedido?

Repite el ejemplo utilizando los otros niveles de aislamiento (`READ COMMITTED`, `REPEATABLE READ` y `SERIALIZABLE`).

Explica los resultados.

12.3.4.2. Ejemplo: niveles de aislamiento ante el problema *Non-Repeatable Read*.

En este ejemplo vamos a simular que hay dos usuarios que quieren acceder de forma concurrente a los mismos datos de una tabla.

Usuario 1:


```sql
DROP DATABASE IF EXISTS test;
CREATE DATABASE test CHARACTER SET utf8mb4;
USE test;

CREATE TABLE cuentas (
   id INTEGER UNSIGNED PRIMARY KEY,
   saldo DECIMAL(11,2) CHECK (saldo >= 0)
);

INSERT INTO cuentas VALUES (1, 1000);
INSERT INTO cuentas VALUES (2, 2000);
INSERT INTO cuentas VALUES (3, 0);

-- 1. Configuramos que en esta sesión vamos a utilizar el nivel de
--    aislamiento READ UNCOMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;


-- 2. Ejecutamos una transacción para transfereir dinero entre dos
--    cuentas
START TRANSACTION;
SELECT * FROM cuentas WHERE id = 1;
```

Usuario 2:


```
-- 1. Seleccionamos la base de datos
USE test;

-- 2. Configuramos que en esta sesión vamos a utilizar el nivel de
--    aislamiento READ UNCOMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- 3. Iniciamos una transacción y atualizamos los datos de la tabla
--    cuentas
START TRANSACTION;
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;

-- 4. Finalizamos la transacción con COMMIT
COMMIT;
```

Usuario 1:

```
-- 4. Volvemos a ejecutar la misma sentencia para observar los datos
--    que existen en la tabla cuentas
SELECT saldo FROM cuentas WHERE id = 1;
COMMIT;
```

¿Qué es lo que ha sucedido?

Repite el ejemplo utilizando los otros niveles de aislamiento (`READ COMMITTED`, `REPEATABLE READ` y `SERIALIZABLE`).

Explica los resultados.

Si vuelves a leer el valor del saldo, verás que sí se ha ejecutado la actualización (porque ya hemos terminado ambas transacciones), simplemente no se realiza mientras la otra transacción sigue abierta.

12.3.4.3. Ejemplo: niveles de aislamiento ante el problema *Phantom Read*.

En este ejemplo vamos a simular que hay dos usuarios que quieren acceder de forma concurrente a los mismos datos de una tabla.

Usuario 1:

```
DROP DATABASE IF EXISTS test;
CREATE DATABASE test CHARACTER SET utf8mb4;
USE test;

CREATE TABLE cuentas (
   id INTEGER UNSIGNED PRIMARY KEY,
   saldo DECIMAL(11,2) CHECK (saldo >= 0)
);

INSERT INTO cuentas VALUES (1, 1000);
INSERT INTO cuentas VALUES (2, 2000);
INSERT INTO cuentas VALUES (3, 0);

-- 1. Configuramos que en esta sesión vamos a utilizar el nivel de
--    aislamiento READ UNCOMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;


-- 2. Ejecutamos una transacción para transfereir dinero entre dos
--    cuentas
START TRANSACTION;
SELECT SUM(saldo) FROM cuentas;
```

Usuario 2:


```
-- 1. Seleccionamos la base de datos
USE test;

-- 2. Configuramos que en esta sesión vamos a utilizar el nivel de
--    aislamiento READ UNCOMMITTED
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- 3. Iniciamos una transacción y atualizamos los datos de la tabla
--    cuentas
START TRANSACTION;
INSERT INTO cuentas VALUES (4, 3000);

-- 4. Finalizamos la transacción con COMMIT
COMMIT;
```

Cada SGBD gestiona los niveles de aislamiento a su manera. En MySQL, el problema de la lectura fantasma, con el nivel de aislamiento `REPEATABLE READ`,solo se produce si la transacción que realiza la lectura opera antes sobre la fila en cuestión.

El `UPDATE` del siguiente código es necesario únicamente para probarlo con dicho nivel de aislamiento.

Usuario 1:

```
-- 4. Volvemos a ejecutar la misma sentencia tras actualizar la
-- nueva fila para observar los datos que existen en la tabla
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 4;
SELECT SUM(saldo) FROM cuentas;
COMMIT;
```

¿Qué es lo que ha sucedido?

Repite el ejemplo utilizando los otros niveles de aislamiento (`READ COMMITTED`, `REPEATABLE READ` y `SERIALIZABLE`).

Explica los resultados.

IMPORTANTE:

Por todo lo visto anteriormente, es muy importante hacer las transacciones lo más cortas posible.
Por otro lado, aunque no entremos en detalles, dos transacciones se pueden bloquear mutuamente dando lugar a una situación irresoluble.

Ejercicios 12.2

Necesitarás dos ventanas de Workbench tal y como se explica en el apartado 12.3.2.2.

IMPORTANTE: en la concurrencia de transacciones hay más factores implicados de los que vemos por lo que no se pretende una comprensión completa y profunda de los siguientes ejercicios sino que el alumno obtenga una idea general de los problemas que se producen.

Consulta el nivel de aislamiento de MySQL, tanto a nivel global como de sesión:

```
SELECT @@GLOBAL.transaction_isolation, @@transaction_isolation;
```

Borramos creamos de nuevo la tabla cuentas para no arrastrar problemas:

```
DROP TABLE cuentas;
CREATE TABLE cuentas (
   id_operacion INTEGER NOT NULL PRIMARY KEY,
   saldo INTEGER NOT NULL CONSTRAINT no_num_rojos
   CHECK (saldo >= 0)
);

SET AUTOCOMMIT=0;
INSERT INTO cuentas (id_operacion, saldo) VALUES (101, 1000);
INSERT INTO cuentas (id_operacion, saldo) VALUES (202, 2000);
SELECT * FROM cuentas;
COMMIT;
```

1. El problema de la actualización perdida es imposible de reproducir en las pruebas ya que todos los SGBD modernos con servicios de control de la concurrencia lo impiden. Sin embargo, después de la operación de confirmación cualquier transacción simultánea puede sobrescribir los resultados sin antes leer el valor

que confirmado la otra.

Este caso se conoce como "Sobrescritura Ciega" o "Escritura Sucia", y es fácil de producir. No es tanto un problema en sí como un fallo en el diseño de las transacciones.

La situación de sobrescritura ciega se produce, por ejemplo, cuando el programa de aplicación lee los valores de la base de datos, actualiza los valores en memoria, y luego escribe el valor actualizado de nuevo en la base de datos.

Vamos a simularlo utilizando variables locales en MySQL.

```
-- USUARIO 1
SET AUTOCOMMIT=0;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

SET @cantU1 = 200;
SET @saldoU1 = 0;
SELECT saldo INTO @saldoU1
FROM cuentas WHERE id_operacion = 101;
SET @saldoU1 = @saldoU1 - @cantU1;
SELECT @saldoU1;
```

```
-- USUARIO 2
SET AUTOCOMMIT=0;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

SET @cantU2 = 500;
SET @saldoU2 = 0;
SELECT saldo INTO @saldoU2
FROM cuentas WHERE id_operacion = 101;
SET @saldoU2 = @saldoU2 - @cantU2;
```

```
-- USUARIO 1
UPDATE cuentas SET saldo = @saldoU1
   WHERE id_operacion = 101;
```



```
-- USUARIO 2
UPDATE cuentas SET saldo = @saldoU2
   WHERE id_operacion = 101;
```

```
-- USUARIO 1
SELECT id_operacion, saldo FROM cuentas
   WHERE id_operacion = 101;
COMMIT;
```

```
-- USUARIO 2
SELECT id_operacion, saldo FROM cuentas
   WHERE id_operacion = 101;
COMMIT;
```

Se pide que:

a) Realices las operaciones anteriores y expliques qué están haciendo el usuario 1 y el 2 en lenguaje natural (en conjunto, no cada línea del código).

b) ¿Se comporta el sistema de la forma esperada?

c) ¿Hay evidencia de los datos perdidos en este caso?

2. Repite los pasos del ejercicio anterior utilizando el nivel de aislamiento `SERIALIZABLE`. Antes de nada, devuelve la tabla al estado inicial.

   ¿Qué conclusiones sacas?

   Reemplaza `SERIALIZABLE` por `REPEATABLE READ` en ambas transacciones y comenta las diferencias que observes.

3. Vamos a ver cómo compiten `SELECT` y `UPDATE` por los mismos recursos.

   Devuelve la tabla al estado inicial.

```
-- USUARIO 1
SET AUTOCOMMIT=0;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT saldo FROM cuentas
   WHERE id_operacion = 101;
```

```
-- USUARIO 2
SET AUTOCOMMIT=0;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT saldo FROM cuentas
   WHERE id_operacion = 101;
```

```
-- USUARIO 1
UPDATE cuentas
   SET saldo = saldo - 200
   WHERE id_operacion = 101;
```

```
-- USUARIO 2
UPDATE cuentas
   SET saldo = saldo - 500
   WHERE id_operacion = 101;
```

```
-- USUARIO 1
SELECT id_operacion, saldo
   FROM cuentas
   WHERE id_operacion = 101;
COMMIT;
```

```
-- USUARIO 2
SELECT id_operacion, saldo
   FROM cuentas
   WHERE id_operacion = 101;
COMMIT;
```

Observa qué sucede y comenta los resultados.

Repite las pruebas realizando los dos primeros bloques de cada usuario juntos (primero el bloque uno y dos del usuario 1 y luego el bloque 1 y 2 del usuario 2).

4. Vamos a ver cómo compiten un `UPDATE` con otro por los mismos recursos.

   Devuelve la tabla al estado inicial.


```
-- USUARIO 1
SET AUTOCOMMIT=0;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE cuentas
   SET saldo = saldo - 100
   WHERE id_operacion = 101;
```

```
-- USUARIO 2
SET AUTOCOMMIT=0;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE cuentas
   SET saldo = saldo -200
   WHERE id_operacion = 202;
```

```
-- USUARIO 1
UPDATE cuentas
   SET saldo = saldo+100
   WHERE id_operacion = 202;
```

```
-- USUARIO 2
UPDATE cuentas
   SET saldo = saldo + 200
   WHERE id_operacion = 101;
```

```
-- USUARIO 1
COMMIT;
```

```
-- USUARIO 2
COMMIT;
```

¿Que conclusiones sacas de esta ejecución?

Nota: El nivel de aislamiento no tiene ningún papel en este escenario pero es una buena práctica definirlo.

5. Vamos a intentar producir una situación de lectura sucia. La transacción del usua-

rio 1 se ejecuta en modo `REPEATABLE READ`, mientras que la transacción del usuario 2 se configura para ejecutarse a nivel `READ UNCOMMITTED`.

Devuelve la tabla al estado inicial.

```
-- USUARIO 1
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
UPDATE cuentas
   SET saldo = saldo - 100
   WHERE id_operacion = 101;
UPDATE cuentas
   SET saldo = saldo + 100
   WHERE id_operacion = 202;
```

```
-- USUARIO 2
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM cuentas;
COMMIT;
```

```
-- USUARIO 1
ROLLBACK;
SELECT * FROM cuentas;
COMMIT;
```

Preguntas:

a) ¿Conclusiones?

b) ¿Y si reemplazamos `READ UNCOMMITTED` por `READ COMMITTED` en la transacción del usuario 2?

c) ¿Y si reemplazamos `READ UNCOMMITTED` por `REPEATABLE READ` en la transacción del usuario 2?

d) ¿Y si reemplazamos `READ UNCOMMITTED` por `SERIALIZABLE` en la transacción del usuario 2?

6. A continuación veremos la anomalía de lectura no repetible:

   Devuelve la tabla al estado inicial.

```
-- USUARIO 1
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT *
   FROM cuentas
   WHERE saldo > 500;
```

```
-- USUARIO 2
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
UPDATE cuentas SET saldo = saldo - 500
   WHERE id_operacion = 101;
UPDATE cuentas SET saldo = saldo + 500
   WHERE id_operacion = 202;
COMMIT;
```

```
-- USUARIO 1
-- repetimos la consulta
SELECT *
   FROM cuentas
   WHERE saldo > 500;
COMMIT;
```

Preguntas:

a) ¿Lee la transacción del usuario 1 los mismos resultados en el paso 3 y el paso 1?

b) ¿Qué sucede si se establece el nivel de aislamiento de la transacción del usuario 1 como `REPEATABLE READ`?

7. Intentamos ahora producir el caso de inserción fantasma:

   Devuelve la tabla al estado inicial.

```
-- USUARIO 1
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ;
SELECT * FROM cuentas
   WHERE saldo > 1000;
```



```
-- USUARIO 2
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
INSERT INTO cuentas (id_operacion, saldo)
   VALUES (303,3000);
COMMIT;
```

```
-- USUARIO 1
-- ¿Se muestra la cuenta 303?
SELECT * FROM cuentas
   WHERE saldo > 1000;
COMMIT;
```

Preguntas:

a) ¿Tiene que esperar la transacción del usuario 2 a la del 1?

b) ¿ Se ve la cuenta 303 que inserta la transacción 2 visible desde el entorno de la transacción 1? ¿Por qué?

c) ¿Afecta esto al resultado del paso 3 si cambiamos el orden de los pasos 1 y 2?

d) ¿Qué conclusiones sacas?

8. Una prueba en la que se producen varios tipos de fantasmas.

```
-- Creamos una nueva tabla:
DROP TABLE T;

CREATE TABLE T (id INT NOT NULL PRIMARY KEY,
   nombre VARCHAR(30), numero SMALLINT);
INSERT INTO T (id, nombre, numero) VALUES (1, 'primero', 1);
INSERT INTO T (id, nombre, numero) VALUES (2, 'segundo', 2);
INSERT INTO T (id, nombre, numero) VALUES (3, 'tercero', 1);
INSERT INTO T (id, nombre, numero) VALUES (4, 'cuarto', 2);
INSERT INTO T (id, nombre, numero)
   VALUES (5, 'ser o no ser', 1);
COMMIT;
```

NOTA: ten en cuenta que los dos últimos pasos los hace el usuario 1. Se han separado para facilitar el ver los resultados.


```sql
-- USUARIO 1
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM T WHERE numero = 1;
```

```sql
-- USUARIO 2
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
INSERT INTO T (id, nombre, numero)
   VALUES (6, 'Inserción fantasma', 1);
UPDATE T SET nombre = 'Actualización fantasma', numero = 1
   WHERE id = 2;
DELETE FROM T WHERE id = 5;
SELECT * FROM T;
```

```sql
-- USUARIO 1
-- Repetimos la consulta después de la modificación
SELECT * FROM T WHERE numero = 1;
INSERT INTO T (id, nombre, numero)
   VALUES (7, 'Insertado por U1', 1);
UPDATE T SET nombre = 'Actualizado por U1'
   WHERE id = 3;
UPDATE T SET nombre = 'Actualizado por U1 tras U2'
   WHERE id = 1;
```

```sql
-- USUARIO 2
COMMIT;
```

```sql
-- USUARIO 1
SELECT * FROM T WHERE numero = 1;
UPDATE T SET nombre = '¿Actu después de borrar?'
   WHERE id = 5;
SELECT * FROM T WHERE numero = 1;
```



```
-- USUARIO 1
COMMIT;
SELECT * FROM T;
```

Preguntas:

a) ¿Son los `insert` y `update` que realiza la transacción del usuario 2 visibles en la del usuario 1? Recuerda que MySQL nos protege en `REPEATABLE READ` de las inserciones en rango, así que si quieres ver el cambio, deberías hacer un `UPDATE` antes del `SELECT` tal y como vimos en el último ejemplo antes de los ejercicios. Esto es una peculiaridad de MySQL.

b) ¿Qué sucede si U1 trata de actualizar la tupla 1 actualizada por la transacción de U2?

c) ¿Qué sucede al tratar U1 de actualizar la tupla 3 que ha sido eliminada por la transacción de U2?

d) ¿Cómo se pueden prevenir los fantasmas mientras la transacción de U1 está activa?

12.4. Bibliografía.

Algunos ejercicios de este tema los he obtenido en el material de José Juan Sánchez Hernández al que agradezco que comparta su trabajo.

- José Juan Sánchez Hernández (2021/2022). Bases de datos / Gestión de Bases de datos
- Gestión de Bases de Datos. IES Luis Vélez de Guevara - Departamento de Informática.
- Structured Query Language. Wikibook. Wikipedia.
- MySQL 8.0 Reference Manual.
- Isolation (database systems). Wikipedia.
- SQL Transactions. Martti Laiho, Dimitris A. Dervos, Kari Silpiö. DBTech VET.
- Iván López Montalbán, Manuel de Castro Vázquez, John Ospino Rivas (2022). Bases de Datos (2ª Edición) . Garceta.
- what are range-locks?. StackOverflow.

