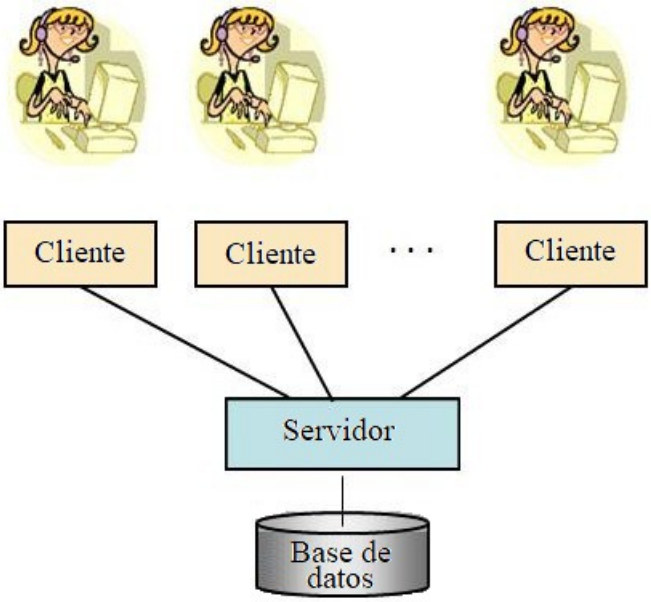
T10. DCL. Concurrencia y Transacciones.

10.1. Introducción.

Una de las principales funciones del lenguaje de control de datos (data control language, DCL) es trabajar de forma activa con los permisos de seguridad que se conceden a los usuarios sobre los objetos de la base de datos (tablas, usuarios, tareas de administración, etc.), de esto hablaremos en el tema siguiente. Así mismo, también sirve para gestionar transacciones, como se verá en este tema.

En los SGBD actuales más de un usuario pueden ejecutar diferentes sentencias sobre la misma BBDD, a esto se le llama **Concurrencia**. Por ejemplo, en la BBDD de un banco, un usuario puede acceder a consultar el saldo de una cuenta cuando al mismo tiempo otro usuario está sacando dinero de esa misma cuenta. Esto es muy peligroso, por lo que los SGBD utilizan unas estrategias de control que se llaman **bloqueos** o cierres. Volviendo al ejemplo del banco, pero ahora utilizando dicha estrategia, si un usuario quiere realizar una inserción de saldo en una cuenta del banco, esa porción de la BBDD se bloquea hasta que se completa la operación y así, si otro usuario quiere, insertar, consultar o retirar saldo de dicha cuenta en ese preciso momento, no podría hasta que se termine la primera operación de inserción.



Normalmente, los SGBD implementan dos tipos de estrategias de bloqueos:

- Bloqueo para cualquier operación de manipulación que se realice sobre una BBDD, es decir, se bloquea: consulta (SELECT), inserción (INSERT), actualización (UPDATE) o borrado (DELETE o TRUNCATE).
- Bloqueo para: inserción (INSERT), actualización (UPDATE) o borrado (DELETE o TRUNCATE), en este tipo de estrategia de bloqueo no entraría la consulta.

Ambas estrategias tienen sus propias ventajas y/o desventajas que irían desde la consistencia (o falta de ella) hasta el tiempo de ejecución de dichas consultas.

En MySQL se implementan las dos estrategias, la primera de ellas, es decir, todas las sentencias bloquean, se implementaría con el motor InnoDB y la segunda de ellas, en la cual la sentencia SELECT no se bloquea, se implementaría con el motor MyISAM.

10.2. Definición de Transacción.

En nuestro día a día realizamos tareas que afectan a varias tablas a la vez, incluso se pueden alterar 10, 20 e incluso 100 tablas en un mismo proceso. No es tarea fácil tener bien ordenadas las sentencias de un mismo proceso y sabiendo que basta con que una de ellas falle para que todo el proceso falle. Por eso los SGBD aportan una herramienta que se llama **transacción**.
Podríamos definir una transacción SQL como un mecanismo que nos permite agrupar "n" sentencias SQL en una sola, de manera que se ejecutan todas o no se ejecuta ninguna. Es decir, se ejecutan formando una unidad lógica de trabajo (*LUW* del inglés *Logic Unit of Work*), en forma indivisible o atómica.

Una transacción SQL finaliza con un COMMIT, para aceptar todos los cambios que la transacción ha realizado en la base de datos (todas las sentencias han tenido éxito), o un ROLLBACK para deshacerlos (alguna de las sentencias ha fallado).

Sintaxis de una transacción:

```
START TRANSACTION
    [transaction_characteristic [, transaction_characteristic] ...]
transaction_characteristic: {
    WITH CONSISTENT SNAPSHOT
  | READ WRITE
  | READ ONLY
}
BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
SET autocommit = {0 | 1}
```

Ejemplo:

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

Explicación:

Cuando se escribe `START TRANSACTION` le decimos al SGBD que a partir de ahora todas las sentencias se ejecuten como una sola y así, si lo deseamos, podremos (o no) deshacer todos los

cambios. Se modificaría de forma permanente mi BBDD con la orden `COMMIT`, por lo tanto se terminaría la transacción y ya no se podrían deshacer los cambios. O se revierten los cambios con la orden `ROLLBACK`, la transacción termina y se vuelve al estado anterior justo antes de que comenzara la transacción.

En resumidas cuentas: se empieza la transacción, se ejecutan todas las sentencias, si todo ha ido correctamente: `COMMIT`, si falla algo: `ROLLBACK`.

10.3. Propiedades ACID.

Las propiedades ACID garantizan que las transacciones se puedan realizar en una base de datos de forma fiable. Decimos que un Sistema Gestor de Bases de Datos es ACID *compliant* cuando permite realizar transacciones.

ACID es un acrónimo de *Atomicity*, *Consistency*, *Isolation* y *Durability*.

- Atomicidad: Esta propiedad quiere decir que una transacción es indivisible, o se ejecutan todas la sentencias (se confirma con COMMIT) o no se ejecuta ninguna (se deshacen con ROLLBACK).
- Consistencia: Esta propiedad asegura que después de una transacción la base de datos estará en un estado válido y consistente. las operaciones de la transacción no deben violar ninguna restricción de la base de datos (claves primarias, claves únicas, claves externas, comprobaciones). La mayoría de los SGBD aplican las restricciones inmediatamente a cada operación. La interpretación más restrictiva de la consistencia requiere que la lógica de la aplicación en la transacción tiene que ser correcta y adecuadamente probada (transacción bien-formada) incluyendo el manejo de excepciones.
- Aislamiento: Esta propiedad garantiza que cada transacción está aislada del resto de transacciones y que el acceso a los datos se hará de forma exclusiva. Por ejemplo, si una transacción que quiere acceder de forma concurrente a los datos que están siendo utilizados por otra transacción, no podrá hacerlo hasta que la primera haya terminado. No se satisfice en la mayoría de los SGBD, sin embargo debe ser considerado por los desarrolladores de aplicaciones.
- Durabilidad: Esta propiedad quiere decir que los cambios que realiza una transacción confirmada sobre la base de datos son permanentes.

10.4. Problemas de concurrencia.

Sin los servicios de control de la concurrencia apropiados en un sistema gestor de bases de datos o con la falta de conocimiento sobre cómo usar estos servicios de forma adecuada, el **contenido** en la base de datos o los **resultados** de nuestras consultas podrían corromperse, y dejar de ser fiables.

En esta sección se tratan los problemas (anomalías) típicos debidos a la concurrencia:

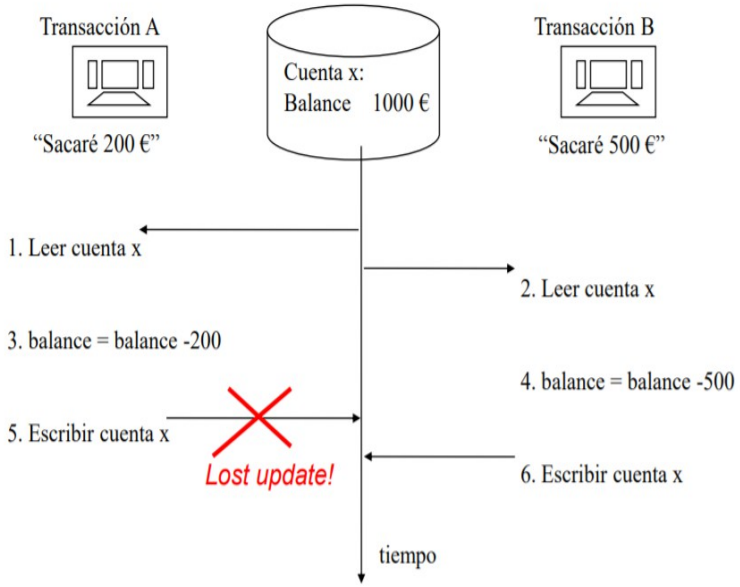
- Problema de la pérdida de actualizaciones (*lost update*).
- Problema de la lectura sucia (*dirty read*) debido a la lectura de datos sin confirmar de alguna transacción concurrente.
- Problema de la lectura no repetible (*non-repeatable read*) debido a una lectura que al repetirse no devuelva las mismas tuplas.

- Problema de la lectura fantasma (*phantom read*) debido a que durante la transacción algunas tuplas no son vistas por la transacción.

La solución a este tipo de problemas pasa por implementar distintos niveles de aislamiento que evitan los distintos problemas mediante políticas de bloqueo. Pero estas, a su vez, causan otros problemas. El más habitual es el *deadlock*, situación en la que dos transacciones quedan bloqueadas indefinidamente, ya que ambas intentan acceder a datos bloqueados por otra transacción. En ese caso, el SGBD debe implementar algoritmos de detección de *deadlocks*, forzando el **ROLLBACK** de una de las transacciones comprometidas.

Problema de la Actualización Perdida.

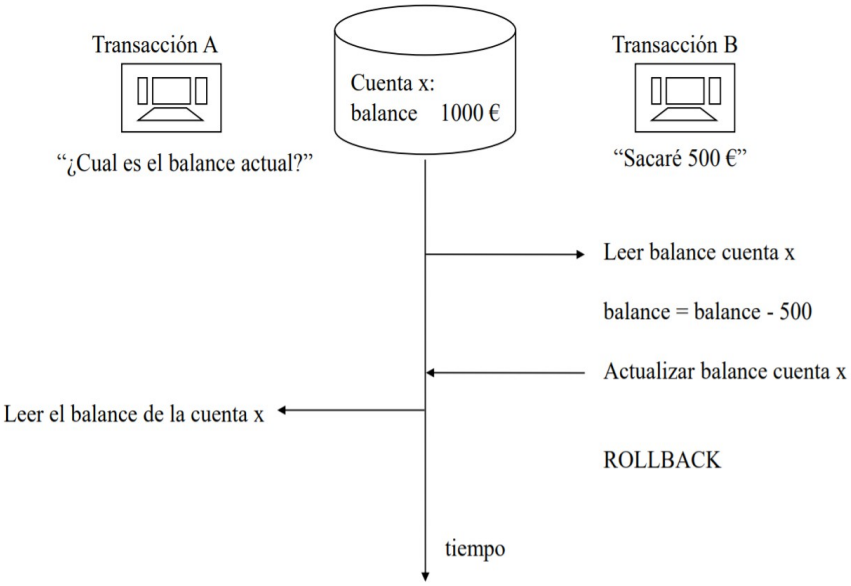
Supongamos dos usuarios en dos cajeros automáticos distintos que están sacando dinero de la misma cuenta con un balance inicial de 1.000 Euros.



Sin un control de la concurrencia, el resultado de 800 Euros de la operación de escritura de la transacción A en el paso 5 se perderá en el paso 6, ya que la transacción B escribirá el nuevo balance de 500 Euros que ha calculado. Si esto sucede antes de la finalización de la transacción A el fenómeno se denomina "**Actualización Perdida**". Sin embargo, 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.

Si este escenario se implementa usando secuencias "SELECT … UPDATE" y se protege bloqueando el esquema, entonces en vez de tener el problema de la actualización perdida, el escenario se convierte en un INTERBLOQUEO. En este caso, por ejemplo, la transacción B debería deshacerse (ROLLBACK) por el SGBD y la transacción A podría finalizar correctamente.

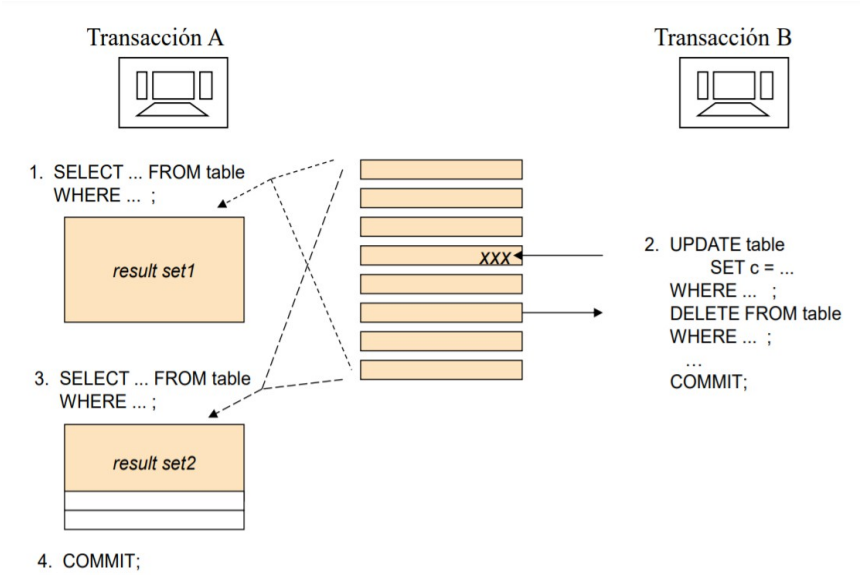
Problema de la Lectura Sucia (*Dirty Read*).



La anomalía de lectura sucia presentada significa que la transacción acepta el riesgo de leer datos no fiables (no confirmados) que podrían cambiar o actualizarse con datos que podrían ser deshechos (ROLLBACK). Este tipo de transacción no debe hacer ninguna actualización en la base de datos ya que podría llevar a datos corruptos. De hecho, cualquier uso de datos sin confirmar es arriesgado y puede llevar a decisiones o acciones incorrectas.

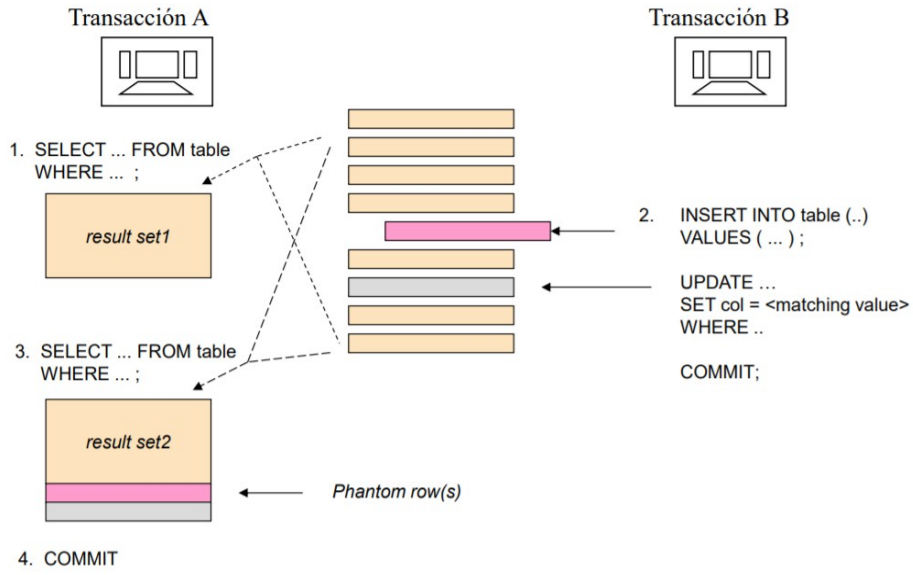
Problema de las No-Reproducible.

La anomalía de no-reproducible significa que los resultados de las consultas en la transacción no son estables, por lo que si las consultas deben repetirse, algunas de las tuplas previamente obtenidas podrían no estar disponibles. Esto no excluye tampoco el caso de que puedan aparecer nuevas tuplas en las consultas repetidas.



Problema Lectura de Fantasmas.

La anomalía de la lectura fantasma significa que los resultados de las consultas en la transacción pueden incluir nuevas tuplas en consultas repetidas. Esto puede incluir tanto la inclusión de nuevas tuplas como la modificación de valores en una tupla que afecten a las condiciones de las consultas realizadas.



10.5. Niveles de aislamiento

Para evitar que sucedan los problemas de acceso concurrente que hemos comentado en el punto anterior podemos establecer diferentes niveles de aislamiento. Dependiendo del mecanismo de control de la concurrencia se pueden dar tiempos de espera muy elevados, bajando la velocidad de producción de la base de datos.

Los niveles de aislamiento no indican nada acerca de restricciones de escritura. Para las operaciones de escritura se usa típicamente alguna protección de bloqueo, y una escritura se protege siempre contra sobre-escritura de otras transacciones hasta el final de la transacción.

Niveles de aislamiento:

- *Lectura no confirmada (Read Uncommited*.). En este nivel no se realiza ningún bloqueo, por lo tanto, permite que sucedan: lectura sucia, lectura no-reproducible y lectura fantasma. La actualización perdida queda protegida por la protección de sobre-escritura de transacciones no confirmadas.
- *Lectura confirmada (Read Commited)*. Solo se leen datos que han sido confirmados. Puede haber dos situaciones, según cada SGDB:que se lea el último dato confirmado o que se espere a que terminen otras transacciones en curso. Aunque los datos leídos por una transacción pueden ser modificados por otras transacciones, por lo tanto, se pueden dar los problemas lectura no repetible y lectura fantasma (*Phantom Read - Non Repeateable Read)*.

- *Lectura repetible (Repeateable Read*). En este nivel se leen datos confirmados y no se permite realizar: UPDATE o DELETE. Esto permite que se puedan volver a leer (SELECT) en la misma transacción y obtener el mismo resultado. No se evita que aparezcan nuevos registros, que sería el problema: lectura fantasma (*Phantom Read*).
- Serializable *(Serializable)*. Permite la lectura solo de datos confirmados, y es posible repetir la lectura sin ningún INSERT, UPDATE, o DELETE hecho por las transacciones concurrentes en el conjunto de tablas accedidas. El efecto que produce es que las transacciones se ejecutan unas detrás de otras, sin que exista la posibilidad de concurrencia.

Nivel de Aislamiento	Actualización Perdida	Lectura Sucia	Lectura No-Reproducible	Fantasmas
LECTURA NO CONFIRMADA	NO posible	Posible	Posible	Posible
LECTURA CONFIRMADA	NO posible	NO posible	Posible	Posible
LECTURA REPRODUCIBLES	NO posible	NO posible	NO posible	Posible
SERIALIZABLE	NO posible	NO posible	NO posible	NO posible

El nivel de aislamiento de transacciones que utiliza InnoDB por defecto es lectura reproducible (*Repeateable Read).*

10.6. Otros aspectos.

AUTOCOMMIT.

Algunos Sistemas Gestores de Bases de Datos, como MySQL (si trabajamos con el motor InnoDB) tienen activada por defecto la variable `AUTOCOMMIT`. Esto quiere decir que automáticamente se aceptan todos los cambios realizados después de la ejecución de una sentencia SQL y no es posible deshacerlos.

Aunque la variable `AUTOCOMMIT` está activada por defecto al inicio de una sesión SQL, podemos configurarlo para indicar si queremos trabajar con transacciones implícitas (no es posible deshacer cambios, `AUTOCOMMIT = 1`) o explícitas (se pueden deshacer `AUTOCOMMIT = 0`).

Podemos consultar el valor actual de `AUTOCOMMIT` haciendo:

```
SELECT @@AUTOCOMMIT;
```

Para desactivar la variable `AUTOCOMMIT` hacemos:

```
SET AUTOCOMMIT = 0;
```

Para activar la variable `AUTOCOMMIT` hacemos:

```
SET AUTOCOMMIT = 1;
```

Para poder trabajar con transacciones en MySQL es necesario utilizar InnoDB.

Si se quiere deshabilitar el modo autocommit para una serie única de comandos, es cuando se usa el comando:

```
START TRANSACTION;
```

Con **`START TRANSACTION`**, autocommit permanece deshabilitado hasta el final de la transacción con **`COMMIT`** o **`ROLLBACK`**.

Sentencias que no se pueden deshacer.

Algunos comandos no pueden deshacerse. En general, esto incluye comandos del lenguaje de definición de datos (DDL), tales como los que crean y borran bases de datos, los que crean, borran o alteran tablas o rutinas almacenadas.

Cada uno de los comandos siguientes (y cualquier sinónimo de los mismos) terminan una transacción implícitamente, como si hubiera realizado un **`COMMIT`** después de ejecutar el comando:

`ALTER TABLE, CREATE TABLE, CREATE INDEX, CREATE DATATABLE, DROP DATABASE, DROP INDEX, DROP TABLE, RENAME TABLE, SET AUTOCOMMIT=1, START TRANSACTION, TRUNCATE TABLE,` etc..

Aunque puedas utilizar estas sentencias en las transacciones, es bueno que no incluyas tales comandos. Si inserta una de estas sentencias en una transacción, y luego un comando posterior falla, el efecto global de la transacción no puede deshacerse mediante un comando **`ROLLBACK`**.

Ejemplo de: START TRANSACTION, COMMIT y ROLLBACK

```
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;

-- 1. ¿Qué devolverá esta consulta?
SELECT * FROM cliente;

SET AUTOCOMMIT=0;

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

-- 2. ¿Qué devolverá esta consulta?
SELECT * FROM cliente;

ROLLBACK;

-- 3. ¿Qué devolverá esta consulta?
SELECT * FROM cliente;
```

10.7. Cómo realizar transacciones con procedimientos almacenados.

Podemos utilizar el manejo de errores para decidir si hacemos `ROLLBACK` de una transacción en un *Store Procedure*. En el siguiente ejemplo vamos a capturar los errores que se produzcan de tipo `SQLEXCEPTION`.

Ejemplo:

```
DELIMITER $$

CREATE PROCEDURE transaccion_en_mysql(usuario_id INT, libro_id INT)

BEGIN

  DECLARE EXIT HANDLER FOR SQLEXCEPTION  -- OCURRE UN ERROR
```

```
    BEGIN
        ROLLBACK;
    END;
  START TRANSACTION;
    -- Sentencias SQL
  INSERT INTO prestamos (id_libro, id_usuario) VALUES(libro_id, usuario_id);
  UPDATE libros SET stock = stock - 1 WHERE id_libro = libro_id;
  COMMIT;
END $$
DELIMITER;
```

Este procedimiento se llamará cada vez que en esa biblioteca se preste un libro. Partiendo de una situación estable y consistente debe realizar las siguientes acciones:

- Añadir a la tabla “prestamos” el identificador del libro prestado y el del lector que se lo lleva.
- Restar en la tabla “libros” 1 en el stock del libro prestado.

Ambas acciones componen el proceso de préstamo. ¿Qué ocurre si el código del libro no existiese? Pues, se anota el préstamo en la tabla de “prestamos” pero saltaría un error al actualizar “libros”, quedando la transacción a medias y por tanto la base de datos inconsistente.

Ese error provoca una llamada a manejador de errores (`EXIT HANDLER FOR SQLEXCEPTION`) que ejecuta un rollback volviendo a dejar las tablas consistentes.