BBDD

Bases de Datos – Manipulación de Datos

IES Ciudad Escolar
Desarrollo de Aplicaciones Multiplataforma

Objetivos

- Utilizar las sentencias facilitadas por el sistema gestor para la inserción, borrado, y actualización de la información.
- Utilizar diferentes modelos de transacciones y de sentencias SQL.
- Identificar los efectos de las distintas políticas de bloqueo de registros.

MySQL®

Lenguaje SQL

El lenguaje de programación **SQL es el lenguaje fundamental de los SGBD** relacionales y los elementos que lo componen son:

a) **DML (*Data Manipulation Language*)**: es el lenguaje que consulta o manipula los datos ya existentes de nuestra BD.

b) **DDL (*Data Definition Language*)**: permite la **definición, modificación y eliminación de las estructuras básicas** (BD, tablas, etc.) en un SGBD.

c) **DCL (*Data Control Language*)**: administra a los usuarios de la BD, concediendo o denegando los permisos oportunos.

d) **TCL (*Transaction Control Language*)**: lenguaje que controla el procesamiento de las transacciones de la BD.

Transacciones

- De forma introductoria, podemos decir que las **transacciones** son sentencias SQL, agrupadas lógicamente, que o bien se ejecutan todas sobre la base de datos o bien no se ejecuta ninguna.

- Una transacción SQL finaliza con un COMMIT, para aceptar todos los cambios que la transacción ha realizado en la base de datos, o un ROLLBACK para deshacerlos.

- Las transacciones permiten mantener la consistencia de los datos, asegurando que nunca se ejecutan parcialmente las sentencias incluidas en dichas transacciones.

- Un ejemplo típico de transacciones es una transacción bancaria donde una cantidad de dinero es transferida de una cuenta origen a otra destino y que supondrá las siguientes sentencias:

```
UPDATE cuenta SET balance = balance - cantidad
WHERE cliente = persona1;
UPDATE cuentas SET balance = balance + cantidad
WHERE cliente = persona2;
```

¿Qué sucede si "se cae" el sistema después de "sacar" el dinero de la cuenta origen sin haberlo "añadido" a la cuenta destino?

- MySQL 8 usa por defecto el motor de almacenamiento **InnoDB** que soporta el uso de transacciones.

```
SHOW engines;
```

Los motores de almacenamiento son módulos de software que utilizan los SGBDs para crear, leer y actualizar datos de una BD.

- En MySQL – InnoDB, todas las sentencias SQL se ejecutan de forma transaccional. Por defecto, la variable autocommit está activada, lo que significa que cada sentencia SQL que se ejecute, conforma su propia transacción y que MySQL realiza un COMMIT (confirmación) automáticamente tras la ejecución de cada sentencia por lo que imposibilita la realización de un ROLLBACK (descartar cambios).

Transacciones

- Las transacciones deben cumplir las cuatro propiedades **ACID:**
  - **Atomicidad**: Las transacciones son indivisibles (o se ejecutan todas sus sentencias o ninguna).
  - **Consistencia**: Al finalizar una transacción, la BD estará en un estado válido y consistente.
  - **Aislamiento**: Cada transacción está aislada del resto de transacciones y el acceso a los datos en cada una, se hace de forma exclusiva (si una transacción accede a los datos que están siendo tratados por otra, no podrá hacerlo hasta que la primera se haya completado).
  - **Durabilidad**: Los cambios que se realizan en una transacción sobre una BD son permanentes.

- Podemos activar o desactivar el modo **autocommit**: `SET autocommit = {0 | 1}`

- Podemos saber el estado del **autocommit**:

```
SHOW VARIABLES WHERE variable_name='autocommit';
SELECT @@autocommit;
```

- Si el modo **autocommit** está desactivo (SET autocommit = 0), la sesión siempre tiene abierta una transacción y hasta realizar el COMMIT o ROLLBACK, no dará comienzo una nueva transacción. Además, en caso de cerrarse la sesión sin hacer COMMIT, MySQL realizará un ROLLBACK automáticamente.

- Los pasos para ejecutar una transacción explícitamente en MySQL son:

1. `START TRANSACTION;`
2. Realizar las operaciones DML sobre la BD (altas, bajas o modificaciones).
3. { `COMMIT;` | `ROLLBACK;` } en función de si queremos dar persistencia a los datos en la BD o si por el contrario queremos deshacer los cambios y que no se lleve ninguno a BD.

https://dev.mysql.com/doc/refman/8.0/en/innodb-autocommit-commit-rollback.html

Acceso concurrente a los datos

- En MySQL, los accesos concurrentes están autorizados. Por lo tanto, es posible que una petición lea un conjunto de filas, mientras que otra petición cambia algunas de estas filas. Por ello, es indispensable que cada conexión MySQL indique a las demás conexiones cuáles son los recursos que no deben ser modificados. Esta notificación se efectúa mediante la incorporación de **bloqueos**.

- Cuando dos transacciones distintas intentan acceder concurrentemente a los mismos datos pueden ocurrir los siguientes problemas:

  - **Dirty Read** (Lectura sucia). Sucede cuando una segunda transacción lee datos que están siendo modificados por una transacción antes de que haga COMMIT o ROLLBACK.

Transacción 1	Transacción 2
`UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;`	
	`SELECT saldo FROM cuentas WHERE id = 1;`
`ROLLBACK`	

  - **Non-Repeatable Read** (Lectura No Repetible). Se produce cuando una transacción consulta el mismo dato dos veces durante la ejecución de la transacción y la segunda vez encuentra que el valor del dato ha sido modificado por otra transacción.

Transacción 1	Transacción 2
`SELECT saldo FROM cuentas WHERE id = 1;`	
	`UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;`
`SELECT saldo FROM cuentas WHERE id = 1;`	

  - **Phantom Read** (Lectura fantasma). Este error ocurre cuando una transacción ejecuta dos veces una consulta que devuelve un conjunto de filas y en la segunda ejecución de la consulta aparecen nuevas filas en el conjunto que no existían cuando se inició lo transacción.

Transacción 1	Transacción 2
`SELECT SUM(saldo)FROM cuentas;`	
	`INSERT INTO cuentas VALUES (4, 3000);`
`SELECT SUM(saldo)FROM cuentas;`	

Acceso concurrente a los datos

- Para evitar que sucedan esos problemas de acceso concurrente, se pueden establecer diferentes **niveles de aislamiento** que controlan el nivel de bloqueo durante el acceso a los datos:

    - **Read Uncommitted**. No se realiza ningún bloqueo.
    - **Read Committed**. Impide las lecturas de datos sucios impidiendo leer valores de datos modificados, pero aún no confirmados por otras transacciones.
    - **Repeatable Read**. Incluye READ COMMITTED y adicionalmente fuerza a que ninguna otra transacción pueda modificar o eliminar los datos leídos por la transacción actual hasta que esta se confirme.
    - **Serializable.** La más restrictiva. Todas las lecturas de la transacción solo únicamente ven datos confirmados antes de comenzar dicha transacción, y nunca ven los cambios de la transacción concurrente confirmados durante la ejecución de la transacción.

Nivel	**Dirty Read (Lectura sucia)**	**Non-Repeatable Read (Lectura No Repetible)**	**Phantom Read (Lectura fantasma)**
Read Uncommitted	Es posible	Es posible	Es posible
Read Committed	-	Es posible	Es posible
Repeatable Read	-	-	Es posible
Serializable	-	-	-

- El nivel de aislamiento que utiliza **InnoDB** por defecto es **Repeatable Read.** SELECT @@transaction_isolation;

SET SESSION TRANSACTION ISOLATION LEVEL {READ COMMITTED | READ UNCOMMITTED | REPEATABLE READ | SERIALIZABLE};

Políticas de bloqueo

- Como usuarios tenemos formas de establecer bloqueos sobre los datos que consultamos o modificamos para impedir que otros usuarios puedan realizar modificaciones al mismo tiempo.

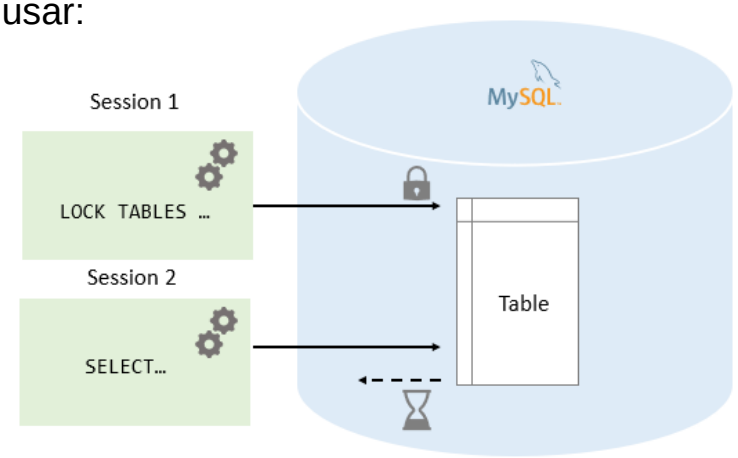
- Existen dos tipos de bloqueos: los **bloqueos de lectura**, que permiten a otras conexiones leer los mismos datos pero nadie puede modificarlos, y los **bloqueos de escritura**, que prohíben a todas las demás conexiones leer o escribir pero sí al que crea el bloqueo. Los primeros pueden ser adquiridos por varias sesiones mientras que los segundos sólo pueden ser adquiridos por una única sesión.

- El bloqueo de los datos se puede realizar a nivel de: base de datos, tabla, fila o columna (**InnoDB** realiza bloqueo de fila por defecto).

- Para bloquear/desbloquear tablas explícitamente podemos usar:

Operativa recomendada durante bloqueo tablas

```
SET autocommit = 0;
LOCK TABLE nom_tabla1 READ;
LOCK TABLE nom_tabla2 WRITE;
… (cambios en esas tablas)…
commit;
UNLOCK TABLES;
```



- En InnoDB no es habitual bloquear tablas completas sino determinados registros. Podemos establecer dos tipos de **bloqueos a nivel de registro**: *shared* o *exclusive* (análogos a los bloqueos de lectura y escritura de las tablas:

SELECT … FOR SHARE/UPDATE.

SELECT * FROM performance_schema.data_locks;

Acceso concurrente: Interbloqueos

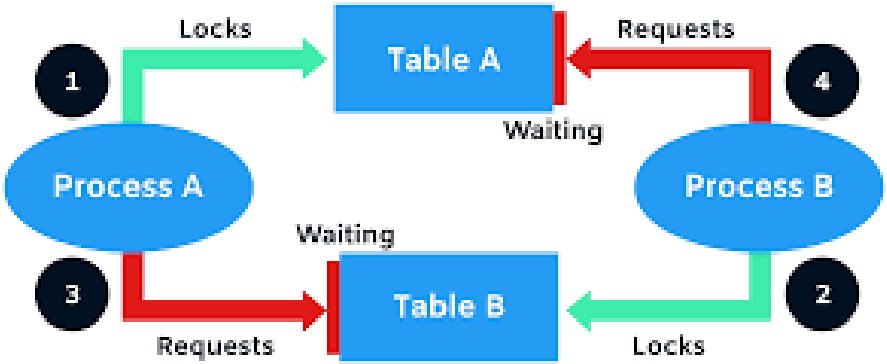
- El uso de bloqueos puede ocasionar antes o después interbloqueos.

- Los interbloqueos o "**deadlocks**" en BDs ocurren cuando dos o más transacciones se mantienen bloqueadas mutuamente debido a que solicitan bloqueos sobre los mismos recursos en un orden diferente, creando así un ciclo de dependencias.

Por ejemplo, tenemos dos transacciones que se ejecutan en la tabla de orders:

Transacción #1
START TRANSACTION;
UPDATE film SET rental_rate= 3.9 WHERE film_id = 1;
UPDATE film SET rental_rate = 6.9 WHERE film_ id = 2;
COMMIT;

Transacción #2
START TRANSACTION;
UPDATE film SET rental_rate = 6.9 WHERE film_ id = 2;
UPDATE film SET rental_rate= 3.9 WHERE film_id = 1;
COMMIT;



Podría pasar que cada transacción ejecutará su primera sentencia actualizando un registro, bloqueándolo durante el proceso. Cada transacción intentará actualizar su segundo registro, pero descubrirá que ya está bloqueado. Las dos transacciones esperarán para siempre a que se completen, a menos que algo intervenga para romper el interbloqueo.

- Para resolver este problema, los sistemas de bases de datos implementan varias formas de **detección de interbloqueos** y **tiempos de espera**. InnoDB detecta las dependencias circulares y fuerza el rollback de una o más transacciones para evitarlo.