Unidad 11. Manipulación de datos.

JOSÉ JUAN SÁNCHEZ HERNÁNDEZ

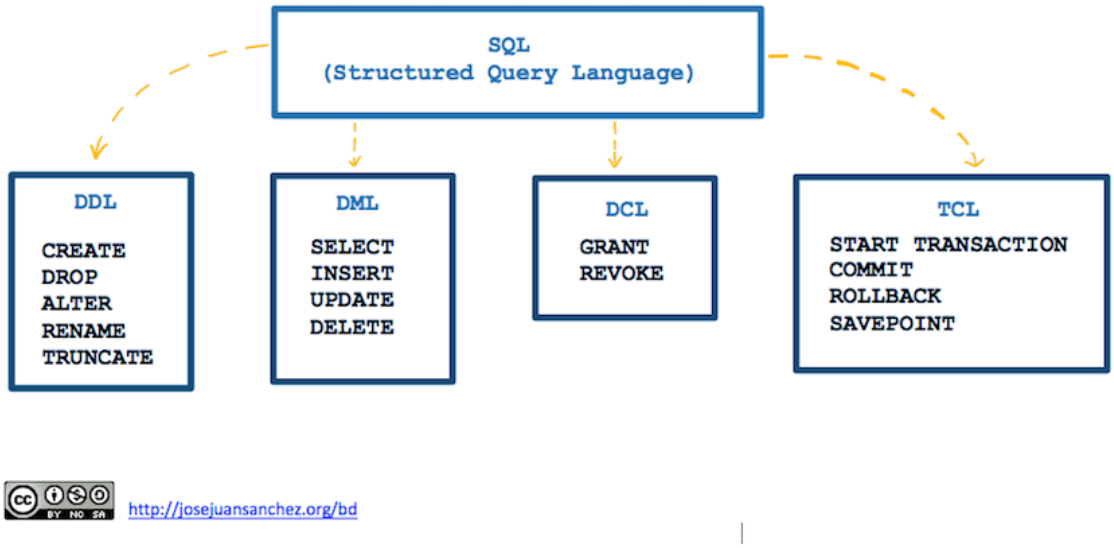
IES Celia Viñas (Almería) - 2020/2021

Índice general

Capítulo 1

Tratamiento de los datos

1.1 El lenguaje DML de SQL



El **DML** (*Data Manipulation Language*) es la parte de SQL dedicada a la manipulación de los datos. Las sentencias **DML** son las siguientes:

- `SELECT`: se utiliza para realizar consultas y extraer información de la base de datos.
- `INSERT`: se utiliza para insertar registros en las tablas de la base de datos.
- `UPDATE`: se utiliza para actualizar los registros de una tabla.
- `DELETE`: se utiliza para eliminar registros de una tabla.

En este tema nos vamos a centrar en el uso de las sentencias `INSERT`, `UPDATE` y `DELETE`.

Capítulo 2

La sentencia `INSERT`

2.1 Sintaxis de la sentencia `INSERT`

Según la documentación oficial de MySQL esta es la sintaxis de la sentencia `INSERT` en MySQL:

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [(col_name [, col_name] ...)]
    {VALUES | VALUE} (value_list) [, (value_list)] ...
    [ON DUPLICATE KEY UPDATE assignment_list]

INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    SET assignment_list
    [ON DUPLICATE KEY UPDATE assignment_list]

INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [(col_name [, col_name] ...)]
    SELECT ...
    [ON DUPLICATE KEY UPDATE assignment_list]

value:
    {expr | DEFAULT}

value_list:
    value [, value] ...

assignment:
```

```
    col_name = value

assignment_list:
    assignment [, assignment] ...
```

2.2 La sentencia INSERT y SELECT

```
INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [(col_name [, col_name] ...)]
    SELECT ...
    [ON DUPLICATE KEY UPDATE assignment_list]

value:
    {expr | DEFAULT}

assignment:
    col_name = value

assignment_list:
    assignment [, assignment] ...
```

Capítulo 3

La sentencia UPDATE

Según la documentación oficial de MySQL esta es la sintaxis de la sentencia `UPDATE` en MySQL:

```
UPDATE [LOW_PRIORITY] [IGNORE] table_reference
    SET assignment_list
    [WHERE where_condition]
    [ORDER BY ...]
    [LIMIT row_count]

value:
    {expr | DEFAULT}

assignment:
    col_name = value

assignment_list:
    assignment [, assignment] ...
```

Capítulo 4

La sentencia DELETE

Según la documentación oficial de MySQL esta es la sintaxis de la sentencia `DELETE` en MySQL:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [WHERE where_condition]
    [ORDER BY ...]
    [LIMIT row_count]
```

Capítulo 5

Borrado y modificación de datos con integridad referencial

- `ON DELETE` y `ON UPDATE`: Nos permiten indicar el efecto que provoca el borrado o la actualización de los datos que están referenciados por claves ajenas. Las opciones que podemos especificar son las siguientes:
  - `RESTRICT`: Impide que se puedan actualizar o eliminar las filas que tienen valores referenciados por claves ajenas. Es la opción por defecto en MySQL.
  - `CASCADE`: Permite actualizar o eliminar las filas que tienen valores referenciados por claves ajenas.
  - `SET NULL`: Asigna el valor `NULL` a las filas que tienen valores referenciados por claves ajenas.
  - `NO ACTION`: Es una palabra clave del estándar SQL. En MySQL es equivalente a `RESTRICT`.
  - `SET DEFAULT`: No es posible utilizar esta opción cuando trabajamos con el motor de almacenamiento **InnoDB**. Puedes encontrar más información en la documentación oficial de MySQL.

Capítulo 6

Transacciones

6.1 Definición

Una transacción SQL es un conjunto de sentencias SQL que se ejecutan formando una unidad lógica de trabajo (*LUW* del inglés *Logic Unit of Work*), es decir, 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, o un `ROLLBACK` para deshacerlos.

MySQL nos permite realizar transacciones en sus tablas si hacemos uso del motor de almacenamiento **InnoDB** (*MyISAM* no permite el uso de transacciones).

El uso de transacciones nos permite realizar operaciones de forma segura y recuperar datos si se produce algún fallo en el servidor durante la transacción, pero por otro lado las transacciones pueden aumentar el tiempo de ejecución de las instrucciones.

Las transacciones deben cumplir las cuatro propiedades **ACID**.

6.2 Propiedades ACID

Las propiedades **ACID** garantizan que las transacciones se puedan realizar en una base de datos de forma segura. 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 o no se ejecuta ninguna.
- **Consistencia**: Esta propiedad asegura que después de una transacción la base de datos estará en un estado válido y consistente.
- **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.

- **Durabilidad**: Esta propiedad quiere decir que los cambios que realiza una transacción sobre la base de datos son permanentes.

6.3 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 o explíticas.

Podemos consultar el valor actual de `AUTOCOMMIT` haciendo:

```
SELECT @@AUTOCOMMIT;
```

Para desactivar la variable `AUTOCOMMIT` hacemos:

```
SET AUTOCOMMIT = 0;
```

Si hacemos esto siempre tendríamos una transacción abierta y los cambios sólo se aplicarían en la base de datos ejecutando la sentencia `COMMIT` de forma explícita.

Para activar la variable `AUTOCOMMIT` hacemos:

```
SET AUTOCOMMIT = 1;
```

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

Se recomienda la lectura del siguiente documento **SQL Transactions**.

6.4 START TRANSACTION, COMMIT y ROLLBACK

Los pasos para realizar una transacción en MySQL son los siguientes:

1. Indicar que vamos a realizar una transacción con la sentencia `START TRANSACTION`, `BEGIN` o `BEGIN WORK`.
2. Realizar las operaciones de manipulación de datos sobre la base datos (insertar, actualizar o borrar filas).
3. Si las operaciones se han realizado correctamente y queremos que los cambios se apliquen de forma permanente sobre la base de datos usaremos la sentencia `COMMIT`. Sin embargo, si durante las operaciones ocurre algún error y no queremos aplicar los cambios realizados podemos deshacerlos con la setencia `ROLLBACK`.

A continuación se muestra la sintaxis que aparece en la documentación oficial para realizar transacciones en MySQL.

```
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 1:

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

Ejemplo 2:

```
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;
```

Ejemplo 3:

```
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. Consultamos el estado actual de las cuentas
SELECT *
FROM cuentas;

-- 2. Suponga 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;

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

-- 4. Suponga 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;

-- 5. ¿Qué devolverá esta consulta?
SELECT *
FROM cuentas;

-- 6. Suponga 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;

-- 7. ¿Qué devolverá esta consulta?
SELECT *
FROM cuentas;
```

Puede encontrar más información en la documentación oficial.

6.5 SAVEPOINT, ROLLBACK TO SAVEPOINT y RELEASE SAVEPOINT

Si trabajamos con tablas **InnoDB** en MySQL también es posible hacer uso de las sentencias: `SAVEPOINT`, `ROLLBACK TO SAVEPOINT` y `RELEASE SAVEPOINT`.

- `SAVEPOINT`: Nos permite establecer un punto de recuperación dentro de la transacción, utilizando un identificacor. Si en una transacción existen dos `SAVEPOINT` con el mismo nombre sólo se tendrá en cuenta el último que se ha definido.
- `ROLLBACK TO SAVEPOINT`: Nos permite hacer un `ROLLBACK` deshaciendo sólo las instrucciones que se hana ejecutado hasta el `SAVEPOINT` que se indique.
- `RELEASE SAVEPOINT`: Elimina un `SAVEPOINT`.

A continuación se muestra la sintaxis que aparece en la documentación oficial para crear `SAVEPOINT`.

```
SAVEPOINT identifier
ROLLBACK [WORK] TO [SAVEPOINT] identifier
RELEASE SAVEPOINT identifier
```

Ejemplo:

```
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');

-- 1. Comprobamos las filas que existen en la tabla
SELECT *
FROM producto;

-- 2. 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;

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

6.6 Acceso concurrente a los datos

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`.
- ***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**.
- ***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.

Ejemplos

6.6.1 Ejemplo de *Dirty Read* (Lectura sucia)

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

6.6.2 Ejemplo de *Non-Repeatable Read* (Lectura No Repetible)

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;`	

6.6.3 Ejemplo de *Phantom Read* (Lectura fantasma)

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

6.7 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 que controlan el nivel de bloqueo durante el acceso a los datos. El estándar ANSI/ISO de SQL (SQL92) define cuatro niveles de aislamiento.

- ***Read Uncommitted***. En este nivel no se realiza ningún bloqueo, por lo tanto, permite que sucedan los tres problemas
- ***Read Committed***. En este caso 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*.
- ***Repeatable Read***. En este nivel ningún registro leído con un `SELECT` puede ser modificado en otra transacción, por lo tanto, sólo puede suceder el problema del *Phantom Read*.
- ***Serializable***. En este caso las transacciones se ejecutan unas detrás de otras, sin que exista la posibilidad de concurencia.

El nivel de aislamiento que utiliza **InnoDB** por defecto es ***Repeatable Read***.

La siguiente tabla muestra los problemas de lectura que pueden ocurrir en cada uno de los modos de aislamiento.

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	-	-	-

Podemos consultar el nivel de aislamiento que estamos utilizando, consultando el contenido de la variable global y de sesión `@@transaction_isolation`.

```
-- Variable global
SELECT @@GLOBAL.transaction_isolation;
```

```
-- Variable de sesión
SELECT @@SESSION.transaction_isolation;
```

También podemos consultar el contenido de la variable de sesión sin utilizar la palabra reservada `SESSION`.

```
SELECT @@transaction_isolation;
```

6.7.1 Ejemplo: Evaluación de los 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. Para simular los dos usuarios vamos a iniciar dos terminales para conectarnos a un servidor MySQL. Desde el **terminal A** vamos a ejecutar las siguientes sentencias 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: Observe que la transacción que estamos ejectando en el terminal A todavía no ha finalizado, porque no hemos ejecutado `COMMIT` ni `ROLLBACK`.

Ahora desde el **terminal B** ejecute las siguientes sentencias SQL:

```
-- 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;
```

Ahora ejecute `ROLLBACK` en el **terminal A** para finalizar la transacción que estaba sin finalizar.

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

Desde el **terminal B** vuelva a ejecutar esta sentencia:

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

¿Qué es lo que ha sucedido? Repita el ejercicio utilizando los otros niveles de aislamiento (`READ COMMITTED`, `REPEATABLE READ` y `SERIALIZABLE`). Tendrá que configurar el nivel de aislamiento que va a utilizar durante la sesión con las siguientes sentencias:

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

6.7.2 Ejemplo: Evaluación de los 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. Para simular los dos usuarios vamos a iniciar dos terminales para conectarnos a un servidor MySQL. Desde el **terminal A** vamos a ejecutar las siguientes sentencias 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;
```

NOTA: Observe que la transacción que estamos ejectando en el terminal A todavía no ha finalizado, porque no hemos ejecutado `COMMIT` ni `ROLLBACK`.

Ahora desde el **terminal B** ejecute las siguientes sentencias SQL:

```
-- 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;
```

Ahora volvemos a ejecutar en el **terminal A** la misma consulta que ejecutamos al inicio de la transacción.

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

¿Qué es lo que ha sucedido? Repita el ejercicio utilizando los otros niveles de aislamiento (`READ COMMITTED`, `REPEATABLE READ` y `SERIALIZABLE`). Tendrá que configurar el nivel de aislamiento que va a utilizar durante la sesión con las siguientes sentencias:

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

6.7.3 Ejemplo: Evaluación de los 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. Para simular los dos usuarios vamos a iniciar dos terminales para conectarnos a un servidor MySQL. Desde el **terminal A** vamos a ejecutar las siguientes sentencias 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 SUM(saldo) FROM cuentas;
```

NOTA: Observe que la transacción que estamos ejectando en el terminal A todavía no ha finalizado, porque no hemos ejecutado `COMMIT` ni `ROLLBACK`.

Ahora desde el **terminal B** ejecute las siguientes sentencias SQL:

```
-- 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;
```

Ahora volvemos a ejecutar en el **terminal A** la misma consulta que ejecutamos al inicio de la transacción.

```
-- 4. Volvemos a ejecutar la misma sentencia para observar los datos que existen
    en la tabla cuentas
SELECT SUM(saldo) FROM cuentas;
```

¿Qué es lo que ha sucedido? Repita el ejercicio utilizando los otros niveles de aislamiento (`READ COMMITTED`, `REPEATABLE READ` y `SERIALIZABLE`). Tendrá que configurar el nivel de aislamiento que va a utilizar durante la sesión con las siguientes sentencias:

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

6.8 Políticas de bloqueo

Cuando una transacción accede a los datos lo hace de forma exclusiva, de modo que una transacción no podrá acceder a los datos que están siendo utilizados por una transacción hasta que ésta haya terminado.

El bloqueo de los datos se puede realizar a nivel de:

- Base de datos.
- Tabla.
- Fila.
- Columna.

InnoDB realiza por defecto un bloqueo a nivel de fila.

6.8.1 Ejemplo

En este ejemplo vamos a simular que hay dos usuarios que quieren acceder de forma concurrente a los mismos datos de una tabla. Para simular los dos usuarios vamos a iniciar dos terminales para conectarnos a un servidor MySQL. Desde el **terminal A** vamos a ejecutar las siguientes sentencias 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;
```

NOTA: Observe que la transacción no que estamos ejectando en el terminal A todavía no ha finalizado, porque no hemos ejecutado `COMMIT` ni `ROLLBACK`.

Ahora desde el **terminal B** ejecute las siguientes sentencias 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 terminal A
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
```

¿Qué es lo que ha ocurrido en el **terminal B**? ¿Puedo acceder a los datos para consultaros? ¿Y para modificarlos? ¿Puedo modificar desde el **terminal B** una cuenta bancaria que no esté siendo utilizada por la transacción del **terminal A**?

Ahora ejecute `COMMIT` en el **terminal A** para finalizar la transacción que estaba sin finalizar. ¿Qué es lo que ha sucedido?

6.9 Cómo realizar transacciones con procedimientos almacenados

Podemos utilizar el manejo de errores para decidir si hacemos `ROLLBACK` de una transacción. En el siguiente ejemplo vamos a capturar los errores que se produzcan de tipo `SQLEXCEPTION` y `SQLWARNING`.

Ejemplo:

```
DELIMITER $$
CREATE PROCEDURE transaccion_en_mysql()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        -- ERROR
        ROLLBACK;
    END;

    DECLARE EXIT HANDLER FOR SQLWARNING
    BEGIN
        -- WARNING
```

```
        ROLLBACK;
    END;

    START TRANSACTION;
    -- Sentencias SQL
    COMMIT;
END
$$
```

En lugar de tener un manejador para cada tipo de error, podemos tener uno común para todos los casos.

```
DELIMITER $$
CREATE PROCEDURE transaccion_en_mysql()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING
    BEGIN
        -- ERROR, WARNING
        ROLLBACK;
    END;

    START TRANSACTION;
    -- Sentencias SQL
    COMMIT;
END
$$
```

Capítulo 7

Ejercicios prácticos

7.1 Tienda de informática

Realice las siguientes operaciones sobre la base de datos `tienda`.

1. Inserta un nuevo fabricante indicando su `código` y su `nombre`.
2. Inserta un nuevo fabricante indicando solamente su `nombre`.
3. Inserta un nuevo producto asociado a uno de los nuevos fabricantes. La sentencia de inserción debe incluir: `código`, `nombre`, `precio` y `código_fabricante`.
4. Inserta un nuevo producto asociado a uno de los nuevos fabricantes. La sentencia de inserción debe incluir: `nombre`, `precio` y `código_fabricante`.
5. Crea una nueva tabla con el nombre `fabricante_productos` que tenga las siguientes columnas: `nombre_fabricante`, `nombre_producto` y `precio`. Una vez creada la tabla inserta todos los registros de la base de datos `tienda` en esta tabla haciendo uso de única operación de inserción.
6. Crea una vista con el nombre `vista_fabricante_productos`que tenga las siguientes columnas: `nombre_fabricante`, `nombre_producto` y `precio`.
7. Elimina el fabricante `Asus`. ¿Es posible eliminarlo? Si no fuese posible, ¿qué cambios debería realizar para que fuese posible borrarlo?
8. Elimina el fabricante `Xiaomi`. ¿Es posible eliminarlo? Si no fuese posible, ¿qué cambios debería realizar para que fuese posible borrarlo?
9. Actualiza el código del fabricante `Lenovo` y asígnale el valor 20. ¿Es posible actualizarlo? Si no fuese posible, ¿qué cambios debería realizar para que fuese actualizarlo?
10. Actualiza el código del fabricante `Huawei` y asígnale el valor 30. ¿Es posible actualizarlo? Si no fuese posible, ¿qué cambios debería realizar para que fuese actualizarlo?
11. Actualiza el precio de todos los productos sumándole 5 € al precio actual.
12. Elimina todas las impresoras que tienen un precio menor de 200 €.

7.2 Empleados

Realice las siguientes operaciones sobre la base de datos `empleados`.

1. Inserta un nuevo departamento indicando su `código`, `nombre` y `presupuesto`.
2. Inserta un nuevo departamento indicando su `nombre` y `presupuesto`.
3. Inserta un nuevo departamento indicando su `código`, `nombre`, `presupuesto` y `gastos`.
4. Inserta un nuevo empleado asociado a uno de los nuevos departamentos. La sentencia de inserción debe incluir: `código`, `nif`, `nombre`, `apellido1`, `apellido2` y `codigo_departamento`.
5. Inserta un nuevo empleado asociado a uno de los nuevos departamentos. La sentencia de inserción debe incluir: `nif`, `nombre`, `apellido1`, `apellido2` y `codigo_departamento`.
6. Crea una nueva tabla con el nombre `departamento_backup` que tenga las mismas columnas que la tabla `departamento`. Una vez creada copia todos las filas de tabla `departamento` en `departamento_backup` .
7. Elimina el departamento `Proyectos`. ¿Es posible eliminarlo? Si no fuese posible, ¿qué cambios debería realizar para que fuese posible borrarlo?
8. Elimina el departamento `Desarrollo`. ¿Es posible eliminarlo? Si no fuese posible, ¿qué cambios debería realizar para que fuese posible borrarlo?
9. Actualiza el código del departamento `Recursos Humanos` y asígnale el valor 30. ¿Es posible actualizarlo? Si no fuese posible, ¿qué cambios debería realizar para que fuese actualizarlo?
10. Actualiza el código del departamento `Publicidad` y asígnale el valor 40. ¿Es posible actualizarlo? Si no fuese posible, ¿qué cambios debería realizar para que fuese actualizarlo?
11. Actualiza el presupuesto de los departamentos sumándole 50000 € al valor del presupuesto actual, solamente a aquellos departamentos que tienen un presupuesto menor que 20000 €.
12. Reliza una **transacción** que elimine todas los empleados que no tienen un departamento asociado.

7.3 Jardinería

Realice las siguientes operaciones sobre la base de datos `jardineria`.

1. Inserta una nueva oficina en `Almería`.
2. Inserta un empleado para la oficina de `Almería` que sea representante de ventas.
3. Inserta un cliente que tenga como representante de ventas el empleado que hemos creado en el paso anterior.
4. Inserte un pedido para el cliente que acabamos de crear, que contenga al menos dos productos.
5. Actualiza el código del cliente que hemos creado en el paso anterior y averigua si hubo cambios en las tablas relacionadas.
6. Borra el cliente y averigua si hubo cambios en las tablas relacionadas.

7. Elimina los clientes que no hayan realizado ningún pedido.
8. Incrementa en un 20% el precio de los productos que no tengan pedidos.
9. Borra los pagos del cliente con menor límite de crédito.
10. Establece a 0 el límite de crédito del cliente que menos unidades pedidas tenga del producto `OR-179`.
11. Modifica la tabla `detalle_pedido` para insertar un campo numérico llamado `iva`. Mediante una transacción, establece el valor de ese campo a `18` para aquellos registros cuyo pedido tenga fecha a partir de Enero de 2009. A continuación actualiza el resto de pedidos estableciendo el `iva` al 21.
12. Modifica la tabla `detalle_pedido` para incorporar un campo numérico llamado `total_linea` y actualiza todos sus registros para calcular su valor con la fórmula:

```
total_linea = precio_unidad*cantidad * (1 + (iva/100));
```

```
ALTER TABLE detalle_pedido
ADD COLUMN total_linea NUMERIC(15,2) NOT NULL DEFAULT 0;

UPDATE detalle_pedido
SET total_linea = cantidad*precio_unidad*(1+(iva/100));

SELECT cantidad, precio_unidad,
     cantidad*precio_unidad*0.21,
       cantidad*precio_unidad*1.21,
       cantidad*precio_unidad*(1+(iva/100))
FROM detalle_pedido
WHERE iva = 21;

SELECT *
FROM detalle_pedido;
```

13. Borra el cliente que menor límite de crédito tenga. ¿Es posible borrarlo solo con una consulta? ¿Por qué?
14. Inserta una oficina con sede en `Granada` y tres empleados que sean representantes de ventas.
15. Inserta tres clientes que tengan como representantes de ventas los empleados que hemos creado en el paso anterior.
16. Realiza una **transacción** que inserte un pedido para cada uno de los clientes. Cada pedido debe incluir dos productos.
17. Borra uno de los clientes y comprueba si hubo cambios en las tablas relacionadas. Si no hubo cambios, modifica las tablas necesarias estableciendo la clave foránea con la cláusula `ON DELETE CASCADE`.
18. Realiza una **transacción** que realice los pagos de los pedidos que han realizado los clientes del ejercicio anterior.

Capítulo 8

Ejercicios prácticos de transacciones

1. Ejecuta las siguientes instrucciones y resuelve las cuestiones que se plantean en cada paso.

```
SET AUTOCOMMIT = 0;
SELECT @@AUTOCOMMIT;

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');


-- 1. Comprueba que las filas se han insertado en la tabla de forma correcta.
SELECT *
FROM producto;
```

Ahora vamos a simular que perdemos la conexión con el servidor antes de que la transacción sea completada (Observa que hemos ejecutado `SET AUTOCOMMIT = 0`). Para simular que perdemos la conexión desde *MySQL Workbench* hay que cerrar la pestaña de conexión con el servidor. Si estás conectado al servidor desde la consola de MySQL sólo tienes que ejecutar el comando `EXIT`.

Volvemos a conectar con el servidor y ejecutamos las siguientes instrucciones:

```
USE test;

-- ¿Qué devolverá esta consulta?
SELECT *
FROM producto;
```

2. Ejecuta las siguientes instrucciones y resuelve las cuestiones que se plantean en cada paso.

```
SET AUTOCOMMIT = 1;
SELECT @@AUTOCOMMIT;

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');


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

-- 2. Vamos a intentar deshacer la transacción actual
ROLLBACK;

-- 3.  ¿Qué devolverá esta consulta? Justifique su respuesta.
SELECT *
FROM producto;

-- 4. Ejecutamos la siguiente transacción
START TRANSACTION;
INSERT INTO producto (id, nombre) VALUES (4, 'Cuarto');
SELECT * FROM producto;
ROLLBACK;

-- 5.  ¿Qué devolverá esta consulta? Justifique su respuesta.
SELECT * FROM producto;

-- 6. Ejecutamos la siguiente transacción
```

```
INSERT INTO producto (id, nombre) VALUES (5, 'Quinto');
ROLLBACK;

-- 7.  ¿Qué devolverá esta consulta? Justifique su respuesta.
SELECT * FROM producto;

-- 8. Desactivamos el modo AUTOCOMMIT y borramos el contenido de la tabla
SET AUTOCOMMIT = 0;
SELECT @@AUTOCOMMIT;

DELETE FROM producto WHERE id > 0;

-- 9. Comprobamos que la tabla esta vacia
SELECT * FROM producto;

-- 10. Insertamos dos filas nuevas
INSERT INTO producto (id, nombre) VALUES (6, 'Sexto');
INSERT INTO producto (id, nombre) VALUES (7, 'Séptimo');
SELECT * FROM producto;

-- 11. Hacemos un ROLLBACK
ROLLBACK;

-- 12.  ¿Qué devolverá esta consulta? Justifique su respuesta.
SELECT * FROM producto;

-- 13. Ejecutamos la siguiente transacción
SET AUTOCOMMIT = 0;
START TRANSACTION;
CREATE TABLE fabricante (id INT UNSIGNED);
INSERT INTO fabricante (id) VALUES (1);
SELECT * FROM fabricante;
ROLLBACK;

-- 14. ¿Se puede hacer ROLLBACK de instrucciones de tipo DDL (CREATE, ALTER, DROP,
     RENAME y TRUNCATE)?
```

Capítulo 9

Ejercicios de teoría

1. ¿Qué son las propiedades *ACID*?
2. ¿Cuáles son los tres problemas de concurrencia en el acceso a datos que pueden suceder cuando se realizan transacciones? Ponga un ejemplo para cada uno de ellos.
3. Cuando se trabaja con transacciones, el SGBD puede bloquear conjuntos de datos para evitar o permitir que sucedan los problemas de concurrencia comentados en el ejercicio anterior. ¿Cuáles son los cuatro niveles de aislamiento que se pueden solicitar al SGBD?
4. ¿Cuál es el nivel de aislamiento que se usa por defecto en las tablas **InnoDB** de MySQL?
5. ¿Es posible realizar transacciones sobre tablas **MyISAM** de MySQL?
6. ¿Qué diferencias existen entre los motores **InnoDB** y **MyISAM** de MySQL?
7. Considera que tenemos una tabla donde almacenamos información sobre cuentas bancarias definida de la siguiente manera:

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

Suponga 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 = 20;
UPDATE cuentas SET saldo = saldo + 100 WHERE id = 30;
COMMIT;
```

- ¿Qué ocurriría si el sistema falla o si se pierde la conexión entre el cliente y el servidor después de realizar la primera sentencia `UPDATE`?

- ¿Qué ocurriría si no existiese alguna de las dos cuentas (`id = 20` y `id = 30`)?
- ¿Qué ocurriría en el caso de que la primera sentencia `UPDATE` falle porque hay menos de 100 € en la cuenta y no se cumpla la restricción del `CHECK` establecida en la tabla?

Capítulo 10

Referencias

- **Bases de Datos**. 2ª Edición. Grupo editorial Garceta. Iván López Montalbán, Manuel de Castro Vázquez y John Ospino Rivas.
- **Gestión de Bases de Datos**. 2ª Edición. Ra-Ma. Luis Hueso Ibáñez.
- **SQL Transactions**. Martti Laiho, Dimitris A. Dervos, Kari Silpiö. DBTech VET Teachers project.
- Transacción (informática). Wikipedia.
- ACID. Wikipedia.
- Aislamiento (ACID). Wikipedia.
- Materiales de la Familia Profesional Informática y Comunicaciones de la Junta de Andalucía.

Capítulo 11

Licencia