

Manipulación de datos, transacciones.

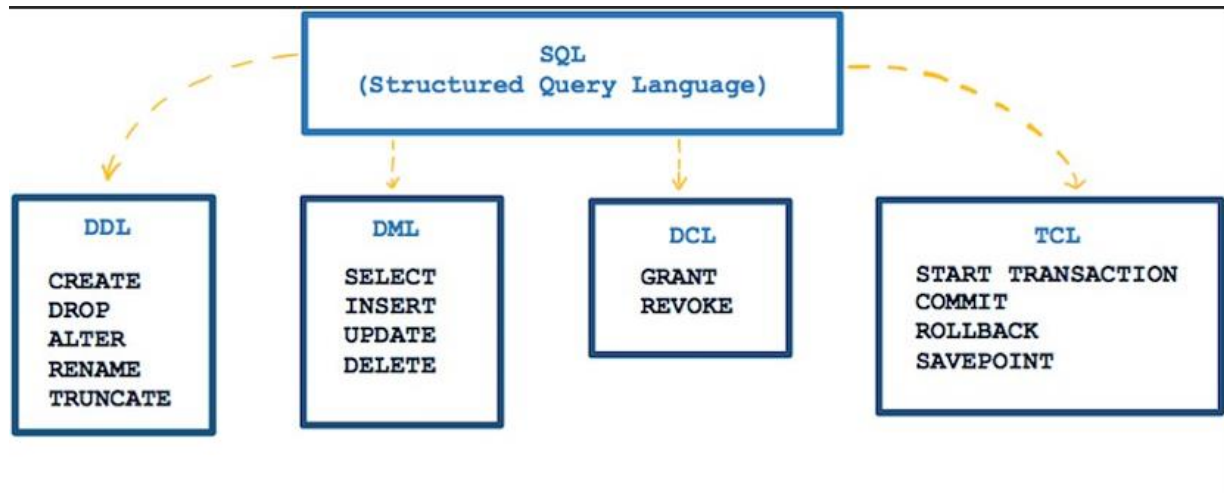
Apuntes BD

Índice

1 Tratamiento de los datos	1
1.1 El lenguaje DML de SQL.....	1
2 La sentencia INSERT	2
2.1 Sintaxis de la sentencia INSERT.....	2
2.2 La sentencia INSERT y SELECT.....	2
3 La sentencia UPDATE	4
4 La sentencia DELETE	5
5 Borrado y modificación de datos con integridad referencial	6
6 Transacciones	7
6.1 Definición.....	7
6.2 Propiedades ACID	7
6.3 AUTOCOMMIT	8
6.4 START TRANSACTION, COMMIT y ROLLBACK	8
6.5 SAVEPOINT, ROLLBACK TO SAVEPOINT y RELEASE SAVEPOINT	10
6.6 Acceso concurrente a los datos	11
6.6.1 Ejemplo de <i>Dirty Read</i> (Lectura sucia)	12
6.6.2 Ejemplo de <i>Non-Repeatable Read</i> (Lectura No Repetible).....	12
6.6.3 Ejemplo de <i>Phantom Read</i> (Lectura fantasma)	12
6.7 Niveles de aislamiento	12
6.7.1 Ejemplo: Evaluación de los niveles de aislamiento ante el problema <i>Dirty Read</i>	13
6.7.2 Ejemplo: Evaluación de los niveles de aislamiento ante el problema <i>Non-Repeatable Read</i> . ..	15
6.7.3 Ejemplo: Evaluación de los niveles de aislamiento ante el problema <i>Phantom Read</i>	16
6.7.4 Ejemplo.....	17
6.8 Cómo realizar transacciones con procedimiento almacenados	18

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**.

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:

```
1  INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
2      [INTO] tbl_name
3      [PARTITION (partition_name [, partition_name] ...)]
4      [(col_name [, col_name] ...)]
5      {VALUES | VALUE} (value_list) [, (value_list)] ...
6      [ON DUPLICATE KEY UPDATE assignment_list]
7
8  INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
9      [INTO] tbl_name
10     [PARTITION (partition_name [, partition_name] ...)]
11     SET assignment_list
12     [ON DUPLICATE KEY UPDATE assignment_list]
13
14  INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
15      [INTO] tbl_name
16      [PARTITION (partition_name [, partition_name] ...)]
17      [(col_name [, col_name] ...)]
18      SELECT ...
19      [ON DUPLICATE KEY UPDATE assignment_list]
20
21  value:
22      {expr | DEFAULT}
23
24  value_list:
25      value [, value] ...
26
27  assignment:
28      col_name = value
29
30  assignment_list:
31      assignment [, assignment] ...
```

2.2 La sentencia INSERT y SELECT

```
1  INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
2      [INTO] tbl_name
3      [PARTITION (partition_name [, partition_name] ...)]
4      [(col_name [, col_name] ...)]
5
6      SELECT ...
7      [ON DUPLICATE KEY UPDATE assignment_list]
8
9  value:
10     {expr | DEFAULT}
11
12  assignment:
13     col_name = value
14
15  assignment_list:
16     assignment [, assignment] ...
```

3 La sentencia UPDATE

Según la [documentación oficial de MySQL](#) esta es la sintaxis de la sentencia UPDATE en MySQL:

```
1  UPDATE [LOW_PRIORITY] [IGNORE] table_reference
2      SET assignment_list
3      [WHERE where_condition]
4      [ORDER BY ...]
5      [LIMIT row_count]
6
7  value:
8     {expr | DEFAULT}
9
10  assignment:
11     col_name = value
12
13  assignment_list:
14     assignment [, assignment] ...
```

4 La sentencia DELETE

Según la [documentación oficial de MySQL](#) esta es la sintaxis de la sentencia DELETE en MySQL:

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

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](#).

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 las 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 explícitas.

Podemos consultar el valor actual de `AUTOCOMMIT` haciendo:

```
1 SELECT @@AUTOCOMMIT;
```

Para desactivar la variable `AUTOCOMMIT` hacemos:

```
1 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:

```
1 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 de 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 sentencia `ROLLBACK`.

A continuación se muestra la sintaxis que aparece en la [documentación oficial para realizar transacciones en MySQL](#).

```
1 START TRANSACTION
2     [transaction_characteristic [, transaction_characteristic] ...]
3
4 transaction_characteristic: {
5     WITH CONSISTENT SNAPSHOT
6 | READ WRITE
7 | READ ONLY
```



```
8 }
9
10 BEGIN [WORK]
11 COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
12 ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
13 SET autocommit = {0 | 1}
```

Ejemplo 1:

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

Ejemplo 2:

```
1 DROP DATABASE IF EXISTS test;
2 CREATE DATABASE test CHARACTER SET utf8mb4;
3 USE test;
4
5 CREATE TABLE cliente (
6     id INT UNSIGNED PRIMARY KEY,
7     nombre CHAR (20)
8 );
9
10 START TRANSACTION;
11 INSERT INTO cliente VALUES (1, 'Pepe');
12 COMMIT;
13
14 -- 1. ¿Qué devolverá esta consulta?
15 SELECT *
16 FROM cliente;
17
18 SET AUTOCOMMIT=0;
19 INSERT INTO cliente VALUES (2, 'Maria');
20 INSERT INTO cliente VALUES (20, 'Juan');
21 DELETE FROM cliente WHERE nombre = 'Pepe';
22
23 -- 2. ¿Qué devolverá esta consulta?
24 SELECT *
25 FROM cliente;
26
27 ROLLBACK;
28
29 -- 3. ¿Qué devolverá esta consulta?
30 SELECT *
31 FROM cliente;
```

Ejemplo 3:

```
1 DROP DATABASE IF EXISTS test;
2 CREATE DATABASE test CHARACTER SET utf8mb4;
3 USE test;
4
5 CREATE TABLE cuentas (
6     id INTEGER UNSIGNED PRIMARY KEY,
7     saldo DECIMAL(11,2) CHECK (saldo >= 0)
```

```
8 );
9
10 INSERT INTO cuentas VALUES (1, 1000);
11 INSERT INTO cuentas VALUES (2, 2000);
12 INSERT INTO cuentas VALUES (3, 0);
13
14 -- 1. Consultamos el estado actual de las cuentas
15 SELECT *
16 FROM cuentas;
17
18 -- 2. Suponga que queremos realizar una transferencia de dinero entre dos
    cuentas bancarias con la siguiente transacción:
19 START TRANSACTION;
20 UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
21 UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;
22 COMMIT;
23
24 -- 3. ¿Qué devolverá esta consulta?
25 SELECT *
26 FROM cuentas;
27
28 -- 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:
29 START TRANSACTION;
30 UPDATE cuentas SET saldo = saldo - 100 WHERE id = 9999;
31 UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;
32 COMMIT;
33
34 -- 5. ¿Qué devolverá esta consulta?
35 SELECT *
36 FROM cuentas;
37
38 -- 6. Suponga que queremos realizar una transferencia de dinero entre dos
    cuentas bancarias con la siguiente transacción y la cuenta origen no tiene
    saldo:
39 START TRANSACTION;
40 UPDATE cuentas SET saldo = saldo - 100 WHERE id = 3;
41 UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;
42 COMMIT;
43
44 -- 7. ¿Qué devolverá esta consulta?
45 SELECT *
46 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

identificador. 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 hayan 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](#).

```
1 SAVEPOINT identifier
2 ROLLBACK [WORK] TO [SAVEPOINT] identifier
3 RELEASE SAVEPOINT identifier
```

Ejemplo:

```
1 DROP DATABASE IF EXISTS test;
2 CREATE DATABASE test CHARACTER SET utf8mb4;
3 USE test;
4
5 CREATE TABLE producto (
6     id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
7     nombre VARCHAR(100) NOT NULL,
8     precio DOUBLE
9 );
10
11 INSERT INTO producto (id, nombre) VALUES (1, 'Primero');
12 INSERT INTO producto (id, nombre) VALUES (2, 'Segundo');
13 INSERT INTO producto (id, nombre) VALUES (3, 'Tercero');
14
15 -- 1. Comprobamos las filas que existen en la tabla
16 SELECT *
17 FROM producto;
18
19 -- 2. Ejecutamos una transacción que incluye un SAVEPOINT
20 START TRANSACTION;
21 INSERT INTO producto (id, nombre) VALUES (4, 'Cuarto');
22 SAVEPOINT spl;
23 INSERT INTO producto (id, nombre) VALUES (5, 'Cinco');
24 INSERT INTO producto (id, nombre) VALUES (6, 'Seis');
25 ROLLBACK TO spl;
26
27 -- 3. ¿Qué devolverá esta consulta?
28 SELECT *
29 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 la transacción.

Ejemplos

6.6.1 Ejemplo de Dirty Read (Lectura sucia)

Transacción 1	Transacción 2
<pre>UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;</pre>	<pre>SELECT saldo FROM cuentas WHERE id = 1;</pre>
<pre>ROLLBACK</pre>	

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

Transacción 1	Transacción 2
<pre>SELECT saldo FROM cuentas WHERE id = 1;</pre>	<pre>UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;</pre>
<pre>SELECT saldo FROM cuentas WHERE id = 1;</pre>	

6.6.3 Ejemplo de Phantom Read (Lectura fantasma)

Transacción 1	Transacción 2
<pre>SELECT SUM(saldo) FROM cuentas;</pre>	<pre>INSERT INTO cuentas VALUES (4, 3000);</pre>
<pre>SELECT SUM(saldo) FROM cuentas;</pre>	

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-Repeatable 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 concurrencia.

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`.

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

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

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

```
1 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:

```
1 DROP DATABASE IF EXISTS test;
2 CREATE DATABASE test CHARACTER SET utf8mb4;
```

```
3 USE test;
4
5 CREATE TABLE cuentas (
6     id INTEGER UNSIGNED PRIMARY KEY,
7     saldo DECIMAL(11,2) CHECK (saldo >= 0)
8 );
9
10 INSERT INTO cuentas VALUES (1, 1000);
11 INSERT INTO cuentas VALUES (2, 2000);
12 INSERT INTO cuentas VALUES (3, 0);
13
14 -- 1. Configuramos que en esta sesión vamos a utilizar el nivel de aislamiento
    READ UNCOMMITTED
15 SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
16 -- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
17 -- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
18 -- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
19
20
21 -- 2. Ejecutamos una transacción para transfereir dinero entre dos cuentas
22 START TRANSACTION;
23 UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
```

NOTA: Observe que la transacción que estamos ejecutando 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 -- 1. Seleccionamos la base de datos
2 USE test;
3
4 -- 2. Configuramos que en esta sesión vamos a utilizar el nivel de aislamiento
    READ UNCOMMITTED
5 SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
6 -- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
7 -- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
8 -- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
9
10 -- 3. Iniciamos una transacción y observamos los datos que existen en la tabla
    cuentas
11 START TRANSACTION;
12 SELECT * FROM cuentas WHERE id = 1;
```

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

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

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

```
1 -- 4. Observamos los datos que existen en la tabla cuentas
2 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:

```
1 DROP DATABASE IF EXISTS test;
2 CREATE DATABASE test CHARACTER SET utf8mb4;
3 USE test;
4
5 CREATE TABLE cuentas (
6     id INTEGER UNSIGNED PRIMARY KEY,
7     saldo DECIMAL(11,2) CHECK (saldo >= 0)
8 );
9
10 INSERT INTO cuentas VALUES (1, 1000);
11 INSERT INTO cuentas VALUES (2, 2000);
12 INSERT INTO cuentas VALUES (3, 0);
13
14 -- 1. Configuramos que en esta sesión vamos a utilizar el nivel de aislamiento
    READ UNCOMMITTED
15 SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
16 -- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
17 -- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
18 -- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
19
20
21 -- 2. Ejecutamos una transacción para transfereir dinero entre dos cuentas
22 START TRANSACTION;
23 SELECT * FROM cuentas WHERE id = 1;
```

NOTA: Observe que la transacción que estamos ejecutando 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 -- 1. Seleccionamos la base de datos
2 USE test;
3
4 -- 2. Configuramos que en esta sesión vamos a utilizar el nivel de aislamiento
    READ UNCOMMITTED
5 SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
6 -- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
7 -- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
8 -- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
9
10 -- 3. Iniciamos una transacción y actualizamos los datos de la tabla cuentas
11 START TRANSACTION;
```

```
12 UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
13
14 -- 4. Finalizamos la transacción con COMMIT
15 COMMIT;
```

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

```
1 -- 4. volvemos a ejecutar la misma sentencia para observar los datos que
  existen en la tabla cuentas
2 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:

```
1 DROP DATABASE IF EXISTS test;
2 CREATE DATABASE test CHARACTER SET utf8mb4;
3 USE test;
4
5 CREATE TABLE cuentas (
6     id INTEGER UNSIGNED PRIMARY KEY,
7     saldo DECIMAL(11,2) CHECK (saldo >= 0)
8 );
9
10 INSERT INTO cuentas VALUES (1, 1000);
11 INSERT INTO cuentas VALUES (2, 2000);
12 INSERT INTO cuentas VALUES (3, 0);
13
14 -- 1. Configuramos que en esta sesión vamos a utilizar el nivel de aislamiento
    READ UNCOMMITTED
15 SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
16 -- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
17 -- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
18 -- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
19
20
21 -- 2. Ejecutamos una transacción para transfereir dinero entre dos cuentas
22 START TRANSACTION;
23 SELECT SUM(saldo) FROM cuentas;
```


NOTA: Observe que la transacción que estamos ejecutando 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 -- 1. seleccionamos la base de datos
2 USE test;
3
4 -- 2. Configuramos que en esta sesión vamos a utilizar el nivel de aislamiento
   READ UNCOMMITTED
5 SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
6 -- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
7 -- SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
8 -- SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
9
10 -- 3. Iniciamos una transacción y actualizamos los datos de la tabla cuentas
11 START TRANSACTION;
12 INSERT INTO cuentas VALUES (4, 3000);
13
14 -- 4. Finalizamos la transacción con COMMIT
15 COMMIT;
```

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

```
1 -- 4. volvemos a ejecutar la misma sentencia para observar los datos que
   existen en la tabla cuentas
2 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;` ## Políticas de bloqueo

Quando 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.

6.7.4 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:

```
1 DROP DATABASE IF EXISTS test;
2 CREATE DATABASE test CHARACTER SET utf8mb4;
3 USE test;
4
5 CREATE TABLE cuentas (
6     id INTEGER UNSIGNED PRIMARY KEY,
7     saldo DECIMAL(11,2) CHECK (saldo >= 0)
8 );
9
10 INSERT INTO cuentas VALUES (1, 1000);
11 INSERT INTO cuentas VALUES (2, 2000);
12 INSERT INTO cuentas VALUES (3, 0);
13
14 -- 1. Ejecutamos una transacción para transfereir dinero entre dos cuentas
15 START TRANSACTION;
16 UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
```

NOTA: Observe que la transacción no que estamos ejecutando 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 -- 1. Seleccionamos la base de datos
2 USE test;
3
4 -- 2. Observamos los datos que existen en la tabla cuentas
5 SELECT *
6 FROM cuentas;
7
8 -- 3. Intentamos actualizar el saldo de una de las cuentas que está siendo
9     utilizada en la transacción del terminal A
10 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 consultarlos? ¿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?

6.8 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 `SQL_EXCEPTION` y `SQL_WARNING`.

Ejemplo:

```
1 DELIMITER $$
2 CREATE PROCEDURE transaccion_en_mysql()
3 BEGIN
4     DECLARE EXIT HANDLER FOR SQLEXCEPTION
5     BEGIN
6         -- ERROR
7         ROLLBACK;
8     END;
9
10    DECLARE EXIT HANDLER FOR SQLWARNING
11    BEGIN
12        -- WARNING
13        ROLLBACK;
14    END;
15
16    START TRANSACTION;
17    -- Sentencias SQL
18    COMMIT;
19 END
20 $$
```

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

```
1 DELIMITER $$
2 CREATE PROCEDURE transaccion_en_mysql()
3 BEGIN
4     DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING
5     BEGIN
6         -- ERROR, WARNING
7         ROLLBACK;
8     END;
9
10    START TRANSACTION;
11    -- Sentencias SQL
12    COMMIT;
13 END
14 $$
```