**Unidad 7. Manipulación de datos.**

Apuntes de BD para DAM

Curso 2023/2024

# Índice

1. [Tratamiento de los datos](#_bookmark0) 1
   1. [El lenguaje DML de SQL](#_bookmark1) 1
2. [La sentencia INSERT](#_bookmark2) 2
   1. [Sintaxis de la sentencia INSERT](#_bookmark3) 2
   2. [La sentencia INSERT y SELECT](#_bookmark4) 2
3. [La sentencia UPDATE](#_bookmark5) 4
4. [La sentencia DELETE](#_bookmark6) 5
5. [Borrado y modificación de datos con integridad referencial](#_bookmark7) 6
6. [Transacciones](#_bookmark8) 7
   1. [Definición](#_bookmark9) 7
   2. [Propiedades ACID](#_bookmark10) 7
   3. [AUTOCOMMIT](#_bookmark11) 8
   4. [START TRANSACTION, COMMIT y ROLLBACK](#_bookmark12) 8
   5. [SAVEPOINT, ROLLBACK TO SAVEPOINT y RELEASE SAVEPOINT](#_bookmark13) 10
   6. [Acceso concurrente a los datos](#_bookmark14) 11
      1. [Ejemplo de Dirty Read (Lectura sucia)](#_bookmark15) 12
      2. [Ejemplo de Non‑Repeatable Read (Lectura No Repetible)](#_bookmark16) 12
      3. [Ejemplo de Phantom Read (Lectura fantasma)](#_bookmark17) 12
   7. [Niveles de aislamiento](#_bookmark18) 12
      1. [Ejemplo: Evaluación de los niveles de aislamiento ante el problema *Dirty Read*](#_bookmark19)13
      2. [Ejemplo: Evaluación de los niveles de aislamiento ante el problema Non‑Repeatable Read.](#_bookmark20) 15
      3. [Ejemplo: Evaluación de los niveles de aislamiento ante el problema *Phantom Read*](#_bookmark21)16
      4. [Ejemplo](#_bookmark22) 17
   8. [Cómo realizar transacciones con procedimientos almacenados](#_bookmark23) 18
7. [Ejercicios prácticos](#_bookmark24) 20
   1. [Tienda de informática](#_bookmark25) 20
   2. [Empleados](#_bookmark26) 20
   3. [Jardinería](#_bookmark27) 21
8. [Ejercicios prácticos de transacciones](#_bookmark28) 23
9. [Ejercicios de teoría](#_bookmark29) 26

i

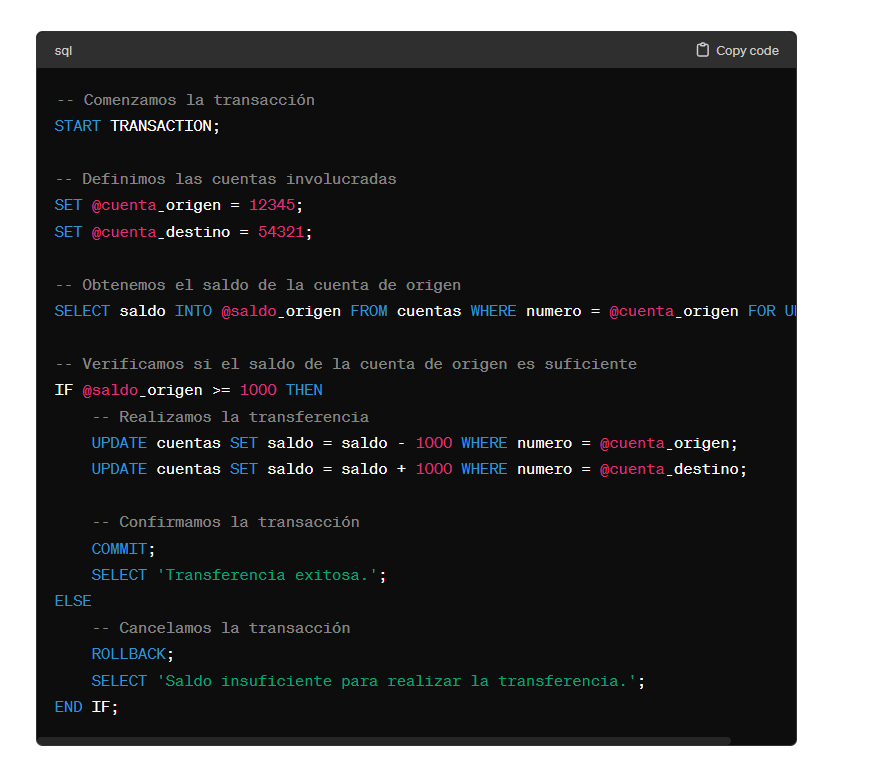
Unidad 7. Manipulación de datos. Curso 2023/2024

#### [Referencias](#_bookmark30) 27

Las transacciones son una parte esencial de las bases de datos relacionales. Una transacción es una secuencia de operaciones que se ejecutan como una unidad atómica, lo que significa que se ejecutan todas juntas o ninguna. Aquí te explico con más detalle qué son y para qué sirven:

1. **¿Qué son las transacciones?**:
   * Una transacción es un conjunto de operaciones (ya sean lecturas o escrituras) que se realizan en la base de datos.
   * Se considera una unidad atómica, lo que significa que todas las operaciones se ejecutan completamente o ninguna se ejecuta en absoluto.
   * Las transacciones aseguran la integridad de los datos y mantienen la coherencia de la base de datos, incluso en situaciones de error o fallo del sistema.
2. **Propiedades ACID**:
   * **Atomicidad**: Una transacción es atómica, es decir, o se ejecutan todas las operaciones con éxito o ninguna.
   * **Consistencia**: La base de datos se mantiene en un estado consistente antes y después de la transacción.
   * **Aislamiento**: Las transacciones se ejecutan independientemente de otras transacciones en curso, como si fueran las únicas operaciones en la base de datos.
   * **Durabilidad**: Los cambios realizados por una transacción exitosa se guardan de manera permanente en la base de datos y no se deshacen en caso de fallo del sistema.
3. **¿Para qué sirven las transacciones?**:
   * **Mantenimiento de la integridad de los datos**: Las transacciones garantizan que los datos se actualicen de manera coherente y segura.
   * **Conservación de la consistencia de la base de datos**: Las transacciones aseguran que la base de datos esté en un estado consistente antes y después de las operaciones.
   * **Manejo de errores y fallos**: Las transacciones permiten revertir los cambios en caso de error o fallo del sistema, manteniendo la integridad de los datos.
   * **Soporte para operaciones complejas**: Las transacciones permiten realizar operaciones complejas que involucran múltiples pasos, como transferencias de fondos bancarios o reservas de inventario.

En resumen, las transacciones en las bases de datos son fundamentales para garantizar la integridad, consistencia y durabilidad de los datos, así como para manejar situaciones de error o fallo del sistema de manera segura.

****

**For Update – donde nose ve toda la linea…**

Este es un ejemplo básico que demuestra el uso de transacciones para garantizar la integridad de los datos en una operación compleja como una transferencia de fondos. Explicación paso a paso:

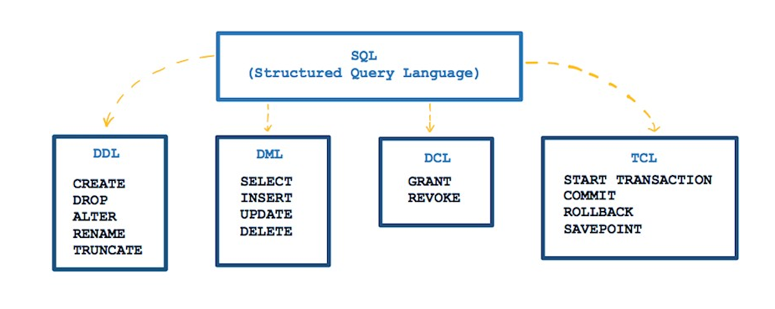
1. **Inicio de la transacción**: **START TRANSACTION;** inicia la transacción. A partir de aquí, todas las operaciones se considerarán parte de la misma transacción.
2. **Obtención del saldo de la cuenta de origen**: Se obtiene el saldo de la cuenta de origen y se guarda en una variable **@saldo\_origen**. Se utiliza **FOR UPDATE** para bloquear la fila y evitar que otros procesos la modifiquen mientras se realiza la transacción.
3. **Verificación del saldo**: Se verifica si el saldo de la cuenta de origen es suficiente para realizar la transferencia.
4. **Transferencia de fondos**: Si hay suficiente saldo en la cuenta de origen, se actualizan los saldos de las cuentas de origen y destino.
5. **Confirmación o cancelación de la transacción**:
   * Si la transferencia se realiza con éxito, se confirma la transacción con **COMMIT;**.
   * Si no hay suficiente saldo en la cuenta de origen, se cancela la transacción con **ROLLBACK;**.
6. **Mensajes de salida**: Se emiten mensajes indicando si la transferencia se realizó correctamente o si hubo un problema de saldo insuficiente.

Esta transacción garantiza que la transferencia de fondos se realice de manera segura y que los saldos de las cuentas se actualicen correctamente, manteniendo la integridad de los datos incluso en caso de error.

Principio del formulario

ii

## 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.

## Sintaxis de la sentencia INSERT

Según la [documentación oficial de MySQL](https://dev.mysql.com/doc/refman/5.7/en/insert.html) 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

1. **INSERT** [LOW\_PRIORITY | DELAYED | HIGH\_PRIORITY] [IGNORE]
2. [**INTO**] tbl\_name
3. [PARTITION (partition\_name [, partition\_name] ...)]
4. **SET** assignment\_list
5. [**ON** DUPLICATE **KEY UPDATE** assignment\_list]

13

1. **INSERT** [LOW\_PRIORITY | HIGH\_PRIORITY] [IGNORE]
2. [**INTO**] tbl\_name
3. [PARTITION (partition\_name [, partition\_name] ...)]
4. [(col\_name [, col\_name] ...)]
5. **SELECT** ...
6. [**ON** DUPLICATE **KEY UPDATE** assignment\_list]

20

1. **value**:
2. {expr | **DEFAULT**}

23

1. value\_list:
2. **value** [, **value**] ...

26

1. assignment:
2. col\_name = **value**

29

1. assignment\_list:
2. assignment [, assignment] ...

## La sentencia INSERT y SELECT

1 **INSERT** [LOW\_PRIORITY | HIGH\_PRIORITY] [IGNORE]

1. [**INTO**] tbl\_name
2. [PARTITION (partition\_name [, partition\_name] ...)]
3. [(col\_name [, col\_name] ...)]

Unidad 11. Manipulación de datos. Curso 2023/2024

1. **SELECT** ...
2. [**ON** DUPLICATE **KEY UPDATE** assignment\_list]

7

1. **value**:
2. {expr | **DEFAULT**}

10

1. assignment:
2. col\_name = **value**

13

1. assignment\_list:
2. assignment [, assignment] ...

Según la [documentación oficial de MySQL](https://dev.mysql.com/doc/refman/5.7/en/update.html) 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

1. **value**:
2. {expr | **DEFAULT**}

9

1. assignment:
2. col\_name = **value**

12

1. assignment\_list:
2. assignment [, assignment] ...

Según la [documentación oficial de MySQL](https://dev.mysql.com/doc/refman/5.7/en/delete.html) esta es la sintaxis de la sentencia DELETE en MySQL:

1 **DELETE** [LOW\_PRIORITY] [QUICK] [IGNORE] **FROM** tbl\_name

1. [PARTITION (partition\_name [, partition\_name] ...)]
2. [**WHERE** where\_condition]
3. [**ORDER BY** ...]
4. [**LIMIT** row\_count]

* 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 almacena‑ miento **InnoDB**. Puedes encontrar más información en la [documentación oficial de MySQL](https://dev.mysql.com/doc/refman/5.7/en/create-table-foreign-keys.html).

## 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.**

## Propiedades ACID

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

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 sen‑ tencias 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.

## 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**](http://myy.haaga-helia.fi/~dbms/dbtechnet/download/SQL-Transactions_handbook_SP.pdf).

## 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 sentencia ROLLBACK.

A continuación, se muestra la sintaxis que aparece en la [documentación oficial para realizar transacciones en](https://dev.mysql.com/doc/refman/8.0/en/commit.html) [MySQL](https://dev.mysql.com/doc/refman/8.0/en/commit.html).

1. START **TRANSACTION**
2. [transaction\_characteristic [, transaction\_characteristic] ...]

3

1. transaction\_characteristic: {
2. WITH CONSISTENT SNAPSHOT
3. | **READ WRITE**
4. | **READ ONLY**

8 }

9

1. **BEGIN** [WORK]
2. **COMMIT** [WORK] [**AND** [**NO**] CHAIN] [[**NO**] RELEASE]
3. **ROLLBACK** [WORK] [**AND** [**NO**] CHAIN] [[**NO**] RELEASE]
4. **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

1. **CREATE TABLE** cliente (
2. id **INT** UNSIGNED **PRIMARY KEY**,
3. nombre **CHAR** (20)

8 );

9

1. START **TRANSACTION**;
2. **INSERT INTO** cliente **VALUES** (1, 'Pepe');
3. **COMMIT**;

13

1. -- 1. ?`Qué devolverá esta consulta?
2. **SELECT** \*
3. **FROM** cliente;

17

1. **SET** AUTOCOMMIT=0;
2. **INSERT INTO** cliente **VALUES** (2, 'Maria');
3. **INSERT INTO** cliente **VALUES** (20, 'Juan');
4. **DELETE FROM** cliente **WHERE** nombre = 'Pepe';

22

1. -- 2. ?`Qué devolverá esta consulta?
2. **SELECT** \*
3. **FROM** cliente;

26

27 **ROLLBACK**;

28

1. -- 3. ?`Qué devolverá esta consulta?
2. **SELECT** \*
3. **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 (

1. id **INTEGER** UNSIGNED **PRIMARY KEY**,
2. saldo **DECIMAL**(11,2) **CHECK** (saldo >= 0)

8 );

9

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

13

1. -- 1. Consultamos el estado actual de las cuentas
2. **SELECT** \*
3. **FROM** cuentas;

17

1. -- 2. Suponga que queremos realizar una transferencia de dinero entre dos cuentas bancarias con la siguiente transacción:
2. START **TRANSACTION**;
3. **UPDATE** cuentas **SET** saldo = saldo - 100 **WHERE** id = 1;
4. **UPDATE** cuentas **SET** saldo = saldo + 100 **WHERE** id = 2;
5. **COMMIT**;

23

1. -- 3. ?`Qué devolverá esta consulta?
2. **SELECT** \*
3. **FROM** cuentas;

27

1. -- 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:
2. START **TRANSACTION**;
3. **UPDATE** cuentas **SET** saldo = saldo - 100 **WHERE** id = 9999;
4. **UPDATE** cuentas **SET** saldo = saldo + 100 **WHERE** id = 2;
5. **COMMIT**;

33

1. -- 5. ?`Qué devolverá esta consulta?
2. **SELECT** \*
3. **FROM** cuentas;

37

1. -- 6. Suponga que queremos realizar una transferencia de dinero entre dos cuentas bancarias con la siguiente transacción y la cuenta origen no tiene saldo:
2. START **TRANSACTION**;
3. **UPDATE** cuentas **SET** saldo = saldo - 100 **WHERE** id = 3;
4. **UPDATE** cuentas **SET** saldo = saldo + 100 **WHERE** id = 2;
5. **COMMIT**;

43

1. -- 7. ?`Qué devolverá esta consulta?
2. **SELECT** \*
3. **FROM** cuentas;

Puede encontrar más información en la [documentación oficial](https://dev.mysql.com/doc/refman/5.6/en/commit.html).

## 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](https://dev.mysql.com/doc/refman/8.0/en/savepoint.html).

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

1. **CREATE TABLE** producto (
2. id **INT** UNSIGNED AUTO\_INCREMENT **PRIMARY KEY**,
3. nombre **VARCHAR**(100) **NOT NULL**,
4. precio DOUBLE

9 );

10

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

14

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

18

1. -- 2. Ejecutamos una transacción que incluye un SAVEPOINT
2. START **TRANSACTION**;
3. **INSERT INTO** producto (id, nombre) **VALUES** (4, 'Cuarto');
4. SAVEPOINT sp1;
5. **INSERT INTO** producto (id, nombre) **VALUES** (5, 'Cinco');
6. **INSERT INTO** producto (id, nombre) **VALUES** (6, 'Seis');
7. **ROLLBACK** TO sp1;

26

1. -- 3. ?`Qué devolverá esta consulta?
2. **SELECT** \*
3. **FROM** producto;

## 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 modifi‑ cados 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

* + 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;

* + 1. **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;

* + 1. **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;

## 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 tran‑ sacción, 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 aisla‑ miento.

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;
3. -- Variable de sesión
4. **SELECT** @@SESSION.transaction\_isolation;

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

1 **SELECT** @@transaction\_isolation;

### 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

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

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 transfeair dinero entre dos cuentas START **TRANSACTION**;

**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

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

9

1. -- 3. Iniciamos una transacción y observamos los datos que existen en la tabla cuentas
2. START **TRANSACTION**;
3. **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 du‑ rante 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;

### 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

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

**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. -- 1. Seleccionamos la base de datos
2. USE test;

3

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

9

1. -- 3. Iniciamos una transacción y atualizamos los datos de la tabla cuentas
2. START **TRANSACTION**;

12 **UPDATE** cuentas **SET** saldo = saldo - 100 **WHERE** id = 1;

13

1. -- 4. Finalizamos la transacción con COMMIT
2. **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 du‑ rante 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;

### 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

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

**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. -- 1. Seleccionamos la base de datos
2. USE test;

3

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

9

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

13

1. -- 4. Finalizamos la transacción con COMMIT
2. **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 du‑ rante 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

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.

### 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

1. **CREATE TABLE** cuentas (
2. id **INTEGER** UNSIGNED **PRIMARY KEY**,
3. saldo **DECIMAL**(11,2) **CHECK** (saldo >= 0)

8 );

9

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

13

1. -- 1. Ejecutamos una transacción para transfereir dinero entre dos cuentas
2. START **TRANSACTION**;
3. **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. -- 1. Seleccionamos la base de datos
2. USE test;

3

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

7

1. -- 3. Intentamos actualizar el saldo de una de las cuentas que está siendo utilizada en la transacción del terminal A
2. **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?

## 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:

1. DELIMITER $$
2. **CREATE** PROCEDURE transaccion\_en\_mysql()
3. **BEGIN**

4 DECLARE EXIT HANDLER FOR SQLEXCEPTION

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

**END**

$$

**BEGIN**

-- ERROR

**ROLLBACK**; **END**;

DECLARE EXIT HANDLER FOR SQLWARNING

**BEGIN**

-- WARNING

**ROLLBACK**; **END**;

START **TRANSACTION**;

-- Sentencias SQL

**COMMIT**;

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

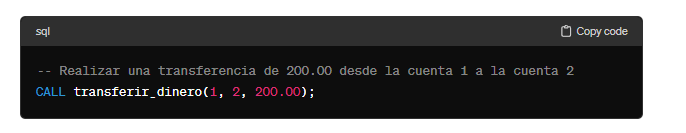
1. START **TRANSACTION**;
2. -- Sentencias SQL
3. **COMMIT**;
4. **END**

14 $$



Este procedimiento almacenado llamado **transferir\_dinero** toma tres parámetros: **cuenta\_origen**, **cuenta\_destino** y **cantidad**. Realiza una transferencia de dinero entre dos cuentas. Comienza iniciando una transacción con **START TRANSACTION**. Luego, obtiene los saldos de las cuentas involucradas y verifica si hay suficiente saldo en la cuenta de origen. Si hay suficiente saldo, se actualizan los saldos de las cuentas y se confirma la transacción con **COMMIT**. Si no hay suficiente saldo, se cancela la transacción y se revierten los cambios con **ROLLBACK**.

Ahora, para usar este procedimiento almacenado, puedes llamarlo con los valores adecuados:



# Ejercicios prácticos

## 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 re‑ gistros 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\_productosque tenga las siguientes columnas:

nombre\_fabricante, nombre\_producto y precio.

* + 1. Elimina el fabricante Asus. ¿Es posible eliminarlo? Si no fuese posible, ¿qué cambios debería realizar para que fuese posible borrarlo?
    2. Elimina el fabricante Xiaomi. ¿Es posible eliminarlo? Si no fuese posible, ¿qué cambios debería realizar para que fuese posible borrarlo?
    3. 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?
    4. 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?
    5. Actualiza el precio de todos los productos sumándole 5 € al precio actual.
    6. Elimina todas las impresoras que tienen un precio menor de 200 €.

## 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 todas 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 debe‑ ría 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, sola‑ mente a aquellos departamentos que tienen un presupuesto menor que 20000 €.
    12. Realiza una **transacción** que elimine todos los empleados que no tienen un departamento asociado.

## 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 diferentes.
    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 11679.
    11. Modifica la tabla detalle\_pedido para insertar un campo numérico llamado iva. Mediante una tran‑ sacción, 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 ac‑ tualiza todos sus registros para calcular su valor con la fórmula:

1 total\_linea = precio\_unidad\*cantidad \* (1 + (iva/100));

* + 1. Borra el cliente que menor límite de crédito tenga. ¿Es posible borrarlo solo con una consulta? ¿Por qué?
    2. Inserta una oficina con sede en Granada y tres empleados que sean representantes de ventas.
    3. Inserta tres clientes que tengan como representantes de ventas los empleados que hemos creado en el paso anterior.
    4. Realiza una **transacción** que inserte un pedido para cada uno de los clientes. Cada pedido debe incluir dos productos.
    5. 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.
    6. Realiza una **transacción** que realice los pagos de los pedidos que han realizado los clientes del ejercicio anterior.

# Ejercicios prácticos de transacciones

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

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

**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 completa‑ da (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:

1 USE test;

2

1. -- ?`Qué devolverá esta consulta?
2. **SELECT** \*
3. **FROM** producto;
4. Ejecuta las siguientes instrucciones y resuelve las cuestiones que se plantean en cada paso.
5. **SET** AUTOCOMMIT = 1;
6. **SELECT** @@AUTOCOMMIT;

3

1. **DROP** DATABASE IF **EXISTS** test;
2. **CREATE** DATABASE test **CHARACTER SET** utf8mb4;
3. USE test;

7

1. **CREATE TABLE** producto (
2. id **INT** UNSIGNED AUTO\_INCREMENT **PRIMARY KEY**,
3. nombre **VARCHAR**(100) **NOT NULL**,
4. precio DOUBLE

12 );

13

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

17

18

1. -- 1. ?`Qué devolverá esta consulta?
2. **SELECT** \*
3. **FROM** producto;

22

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

25

1. -- 3. ?`Qué devolverá esta consulta? Justifique su respuesta.
2. **SELECT** \*
3. **FROM** producto;

29

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

35

1. -- 5. ?`Qué devolverá esta consulta? Justifique su respuesta.
2. **SELECT** \* **FROM** producto;

38

1. -- 6. Ejecutamos la siguiente transacción
2. **INSERT INTO** producto (id, nombre) **VALUES** (5, 'Quinto');
3. **ROLLBACK**;

42

1. -- 7. ?`Qué devolverá esta consulta? Justifique su respuesta.
2. **SELECT** \* **FROM** producto;

45

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

49

50 **DELETE FROM** producto **WHERE** id > 0;

51

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

54

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

59

1. -- 11. Hacemos un ROLLBACK
2. **ROLLBACK**;

62

63 -- 12. ?`Qué devolverá esta consulta? Justifique su respuesta.

64 **SELECT** \* **FROM** producto;

65

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

73

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

# 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 rea‑ lizan 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:

1 **CREATE TABLE** cuentas (

1. id **INTEGER** UNSIGNED **PRIMARY KEY**,
2. saldo **DECIMAL**(11,2) **CHECK** (saldo >= 0)

4 );

Suponga que queremos realizar una transferencia de dinero entre dos cuentas bancarias con la siguiente tran‑ sacción:

1. START **TRANSACTION**;
2. **UPDATE** cuentas **SET** saldo = saldo - 100 **WHERE** id = 20;
3. **UPDATE** cuentas **SET** saldo = saldo + 100 **WHERE** id = 30;
4. **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?

# 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.](http://myy.haaga-helia.fi/~dbms/dbtechnet/download/SQL-Transactions_handbook_SP.pdf)
* [Transacción (informática)](https://es.wikipedia.org/wiki/Transacci%C3%B3n_(inform%C3%A1tica)). Wikipedia.
* [ACID](https://es.wikipedia.org/wiki/ACID). Wikipedia.
* [Aislamiento (ACID)](https://es.wikipedia.org/wiki/Aislamiento_(ACID)). Wikipedia.
* [Materiales de la Familia Profesional Informática y Comunicaciones de la Junta de Andalucía](https://educacionadistancia.juntadeandalucia.es/aulavirtual/course/index.php?categoryid=36).
* [Vídeo. Cómo no vender la misma entrada a dos personas diferentes](https://www.youtube.com/watch?v=mNaX4NfsM0c). José Muñoz. T3chFest 2023.
* **Designing Data‑Intensive Applications**. (Chapter 7. Transactions). O’Reilly. Martin Kleppmann