PARTE III.
TRANSACCIONES

Tema 3. Transacciones en MySQL

3.1 Introducción....2
3.2 Niveles de aislamiento....3
3.3 Instrucciones de manejo de transacciones....9
3.4 Transacciones y bloqueos....13
3.4.1 Deadlock. Cursores de actualización....15
3.5 Estrategias de bloqueo....19
3.6 Aspectos a tener en cuenta a la hora de utilizar transacciones....21

Anexo. Manejo de transacciones desde otras aplicaciones

1. Desde PHP....21
2. Desde Visual Basic Express 2005....22

TEMA 3.
Transacciones en MySQL

Alberto Carrera Martín

3.1 Introducción

TRANSACCIÓN en SQL: Conjunto de instrucciones SQL, agrupadas lógicamente, que o bien se ejecutan todas sobre la base de datos o bien no se ejecuta ninguna.
Una transferencia bancaria entre dos cuentas supone un ejemplo claro para ilustrar el concepto de transacción. La transferencia se compone de dos operaciones (instrucciones):

1) Descontar de la libreta origen a transferir la cantidad fijada.
2) Aumentar el saldo de la libreta destino con el importe de la cantidad transferida de la cuenta origen.

Está claro que la transacción bancaria anterior no se puede quedar "a medias". O bien se aplican las dos operaciones lógicas que componen la transacción o bien no se realiza ninguna por mantener una **consistencia** contable. Si sólo llegara a realizarse una operación de las dos anteriores, esta se desharía.

Las transacciones sobre la base de datos deben ajustarse al principio **ACID** (**A**TOMIC + **C**ONSISTENT + **I**SOLATED + **D**URABLE), lo que implica que una transacción debe ser:

- **ATÓMICA** (ATOMIC): Todas las operaciones que componen la transacción deben aplicarse sobre la base de datos o no aplicarse ninguna. Una transacción es indivisible.

- **CONSISTENTE** (CONSISTENT): Se mantiene la consistencia de los datos antes y después de ejecutar la transacción.

- **AISLADA** (ISOLATED): Cuando muchas transacciones se pueden llevar a cabo simultáneamente por uno o varios usuarios, la realización de una no debe afectar jamás a las otras y por tanto no llevar a una situación de error.

- **PERMANENTE** (DURABLE): Una vez que la transacción ha sido confirmada (COMMIT) y por tanto los cambios de la base de datos guardados, éstos deben perdurar en el tiempo aun cuando el gestor de base de datos o el propio equipo fallen.

La capacidad transaccional de MySQL depende del sistema de almacenamiento empleado. Los dos más importantes son el MyISAM y el InnoDB.

MyISAM: No admite transacciones. Utilizada para aplicaciones que en su mayoría son de sólo lectura. Si las aplicaciones casi no necesitan el empleo de las transacciones, es la forma de almacenamiento a utilizar pues el rendimiento es óptimo. Si el número de

transacciones y accesos concurrentes para modificar los datos de la base de datos es importante, entonces es mejor utilizar el siguiente modelo de almacenamiento.

InnoDB: El modelo de almacenamiento de transacción segura de MySQL más popular. Soporta transacciones ACID así como bloqueos a nivel de fila y concurrencia (a costa de sacrificar un poco la velocidad de proceso). Es el tipo predeterminado de modelo de almacenamiento si se ha instalado el gestor de bases de datos utilizando el asistente y por tanto no se ha especificado a la hora de crear las tablas en la sentencia CREATE TABLE. Puedes comprobar dicho modelo de almacenamiento con la instrucción SHOW CREATE TABLE alumnos y ver la columna CREATE TABLE:

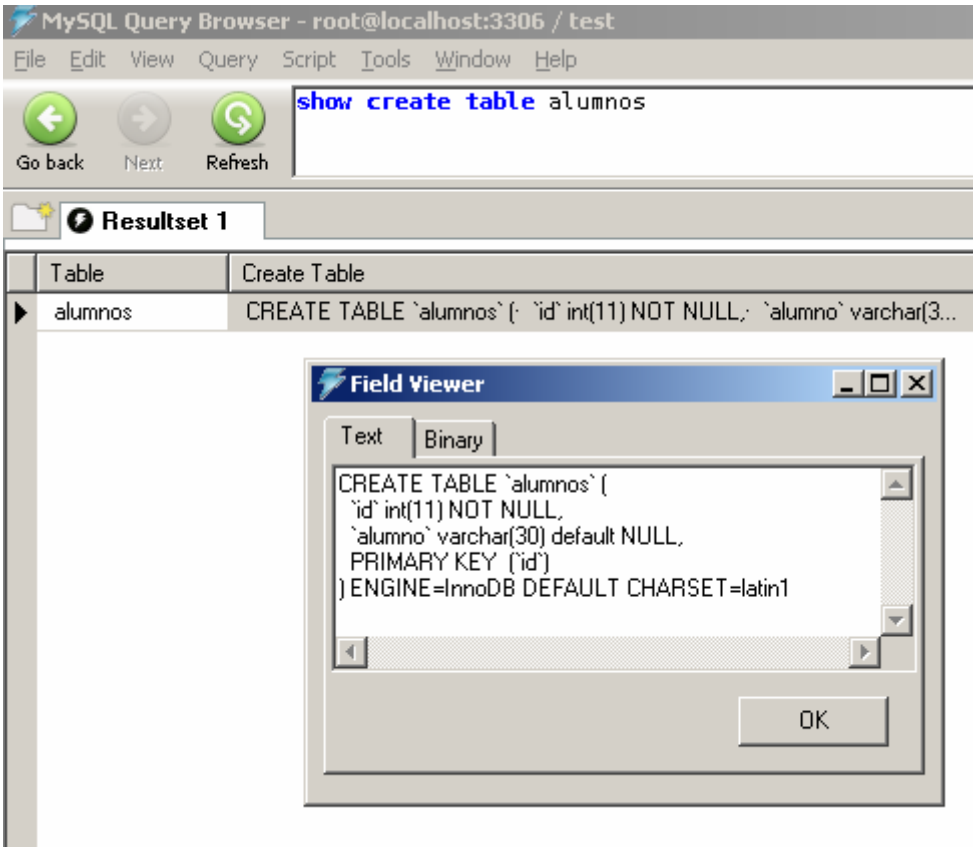

Ilustración 1. Almacenamiento InnoDB

3.2 Niveles de aislamiento

Una sesión de base de datos es una conexión única a la base de datos que comienza cuando se entra en el sistema (login) y termina cuando se desconecta de ella.

Cada vez que abrimos el programa cliente *MySQL Query Browser* estamos abriendo una sesión (podemos tener varias instancias de este programa abiertas → varias sesiones). Para averiguar el identificador de conexión que nos asigna el servidor, consultaremos la función que nos lo proporciona:

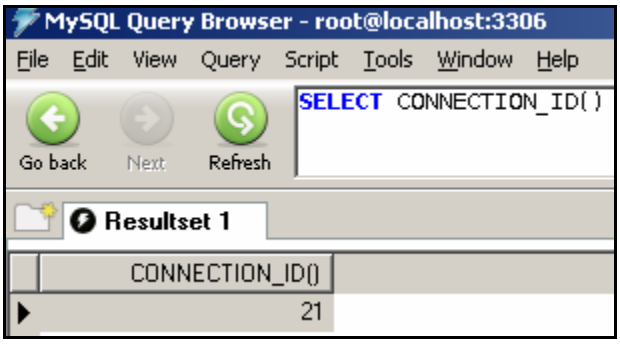

Ilustración 2. Identificador de conexión I

La sesión termina cuando cerramos el programa *MySQL Query Browser*.

También en alguna ocasión hemos creado alguna sesión para trabajar con la base de datos desde la ventana de línea de comandos de Windows:

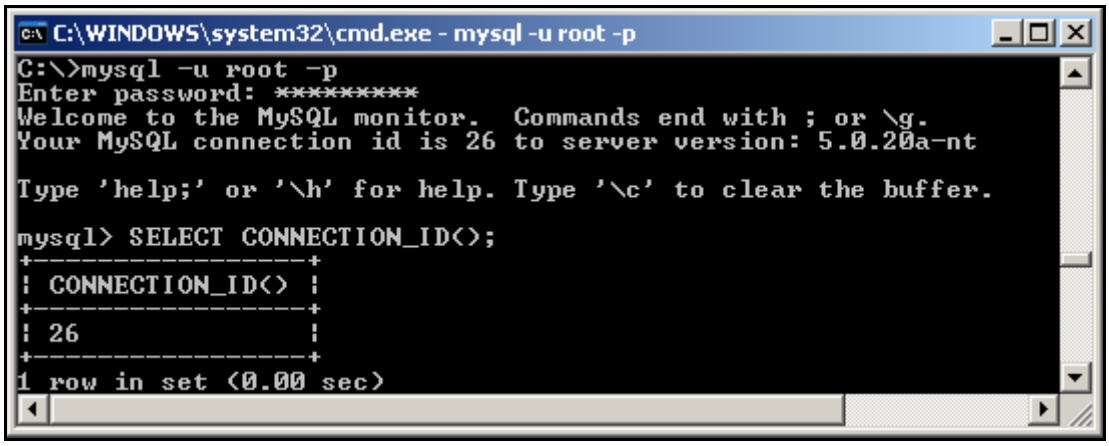

Ilustración 3 Identificador de conexión II

En el momento de conectarnos de esta manera a MySQL se visualiza el identificador de conexión (4ª línea en la ilustración 3) que éste nos asigna. La conexión dura hasta que la abandonemos (comando EXIT).

Cada sesión trabaja en su propia zona de memoria pudiendo llegar incluso a bloquear los datos de la base de datos con los que trabaja.
Los niveles de aislamiento determinan la manera en que las transacciones de una sesión pueden afectar a los datos recuperados o accedidos por otra sesión. Hay por tanto dos conceptos interrelacionados: por una lado la concurrencia (varios sesiones realizando transacciones al mismo tiempo) y por otro el grado de consistencia de los datos. Determinan también el grado en que las transacciones se ajustan al principio ACID. Cuanto mayor es el grado de aislamiento, menor es el número de transacciones que se pueden realizar concurrentemente pero también es menor la posibilidad de que interfieran las transacciones. Por otro lado, cuanto menor es el grado de aislamiento, mayor es el número de transacciones que se pueden realizar concurrentemente pero el riesgo de conflicto entre transacciones es elevado.

El estándar ANSI define 4 modelos de aislamiento, soportados todos ellos por el sistema de almacenamiento **InnoDB** de MySQL:

- **Lectura no confirmada** (también conocido como lectura sucia): Es el nivel más bajo. Permite que una transacción pueda leer filas que todavía no han sido confirmadas (COMMIT). El hecho de que utilizando este nivel se aumente el rendimiento del proceso no justifica que pueda permitirse que un usuario lea datos modificados por otro usuario que todavía puede deshacer dichas modificaciones.

- **Lectura confirmada**. Sólo se permite lectura de datos que han sido confirmados. Si un usuario está ejecutando un procedimiento que recorre las filas recuperadas de una tabla mediante una sentencia SELECT, si otro usuario en ese momento realiza una modificación de la tabla, el primero no verá las modificaciones realizadas por este último.

- **Lectura repetible**. Es el nivel por defecto. Las lecturas repetidas de la misma fila por la misma transacción dan los mismos resultados. Ningún cambio hecho en la base de datos por otros usuarios será visto por la transacción lanzada hasta que esta se confirme o deshaga, es decir, si se repite dentro de una transacción una instrucción SELECT, esta devolverá siempre los mismos resultados (excepto cuando dentro de la misma transacción pudieran realizarse cambios en esa filas recuperadas por la SELECT).

- **Secuenciable**. Mayor nivel de aislamiento. Las transacciones se aíslan completamente dando la impresión de que se ejecutan secuencialmente, una detrás de otra. Para conseguir esto, los sistemas gestores de bases de datos bloquean cada fila leída para que otras sesiones no puedan modificar estos datos hasta que la transacción finalice. El bloqueo dura hasta que la transacción es confirmada o deshecha. Este nivel disminuye mucho el rendimiento del sistema y puede provocar situaciones de "abrazo mortal" como se verá más adelante.

El nivel de aislamiento por defecto se podría cambiar (no aconsejable):

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

Los siguientes ejemplos nos ayudarán a comprender el manejo de transacciones:

Nota: Recordar que se mantiene siempre la consistencia en lectura → Los datos que ven las transacciones son los últimos que se validaron.

Antes de comenzar estos ejemplos ejecuta el *procedimiento1* ya visto en temas anteriores y que crea la tabla alumnos con 5 alumnos:

Abre dos sesiones nuevas de cliente de línea de comandos de MySQL (lo puedes hacer mediante la opción de menú *Tools / MySQL Command Line Client*).

id	alumno
1	alumno 1
2	alumno 2
3	alumno 3
4	alumno 4
5	alumno 5

Ilustración 4. Tabla alumnos

En este momento tendrás 3 sesiones abiertas, una del programa *MySQL Query Browser* y dos clientes de línea de comando de MySQL (ilustración 5). Puedes comprobar el identificador de cada sesión con la función vista anteriormente (SELECT CONNECTION_ID();):

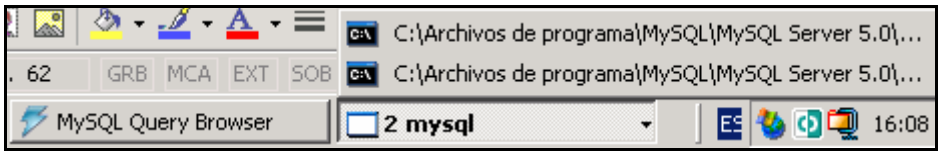

Ilustración 5. Sesiones abiertas en la barra de tareas

Utilizaremos para la práctica las dos sesiones de clientes de línea de comando MySQL.

Como los números de identificación de conexión serán distintos a los que utilices, llamaremos **A** a la conexión de cliente de línea de comando de identificador más bajo y **B** a la otra conexión de cliente cuyo identificador es más alto.

Dejaremos la propiedad **autocommit** a 0 en las dos sesiones abiertas por lo que los cambios en la base de datos no se almacenan después de cada instrucción sino cuando se confirman explícitamente con la instrucción COMMIT.

```
A
> SET autocommit = 0;
> USE TEST
```

```
B
> SET autocommit = 0;
> USE TEST
```

```
A
> INSERT INTO alumnos VALUES (6, 'alumno6');
> SELECT * FROM alumnos WHERE id=6;
+----+---------+
| id | alumno  |
+----+---------+
| 6  | alumno6 |
+----+---------+
```

```
B
> SELECT * FROM alumnos WHERE id=6;
Empty set (0.00 sec)
```

```
A
> COMMIT;
```

```
B
> SELECT * FROM alumnos WHERE id=6;
Empty set (0.00 sec)
> COMMIT;
Query OK, 0 rows affected (0.00 sec)
> SELECT * FROM alumnos WHERE id=6;
+----+---------+
| id | alumno  |
+----+---------+
| 6  | alumno6 |
+----+---------+
1 row in set (0.00 sec)
```

```
A
> UPDATE alumnos SET alumno = 'Alberto Carrera' WHERE id=1;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
> SELECT * FROM ALUMNOS WHERE id=1;
+----+-----------------+
| id | alumno          |
+----+-----------------+
| 1  | Alberto Carrera |
+----+-----------------+
1 row in set (0.00 sec)
```

```
B
> SELECT * FROM ALUMNOS WHERE id=1;
+----+----------+
| id | alumno   |
+----+----------+
| 1  | alumno 1 |
+----+----------+
1 row in set (0.00 sec)

> UPDATE alumnos SET alumno = 'Raquel Carrera' WHERE id = 2;
Query OK, 1 row affected (0.02 sec)
Rows matched: 1  Changed: 1  Warnings: 0

l> SELECT * FROM ALUMNOS WHERE id=2;
+----+----------------+
| id | alumno         |
+----+----------------+
| 2  | Raquel Carrera |
+----+----------------+
1 row in set (0.00 sec)
```

```
A
mysql> SELECT * FROM ALUMNOS WHERE id=2;
+----+----------+
| id | alumno   |
+----+----------+
| 2  | alumno 2 |
+----+----------+
1 row in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM ALUMNOS;
+----+-----------------+
| id | alumno          |
+----+-----------------+
| 1  | Alberto Carrera |
| 2  | alumno 2        |
| 3  | alumno 3        |
| 4  | alumno 4        |
| 5  | alumno 5        |
| 6  | alumno6         |
+----+-----------------+
6 rows in set (0.00 sec)
```

```
B
mysql> SELECT * FROM ALUMNOS;
+----+----------------+
| id | alumno         |
+----+----------------+
| 1  | alumno 1       |
| 2  | Raquel Carrera |
| 3  | alumno 3       |
| 4  | alumno 4       |
| 5  | alumno 5       |
| 6  | alumno6        |
+----+----------------+
6 rows in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.02 sec)

mysql> SELECT * FROM ALUMNOS;
+----+-----------------+
| id | alumno          |
+----+-----------------+
| 1  | Alberto Carrera |
| 2  | Raquel Carrera  |
| 3  | alumno 3        |
| 4  | alumno 4        |
| 5  | alumno 5        |
| 6  | alumno6         |
+----+-----------------+
6 rows in set (0.00 sec)

mysql> SELECT * FROM ALUMNOS;
+----+------------------+
| id | alumno           |
+----+------------------+
| 1  | Alberto Carrera  |
| 2  | alumno 2         |
| 3  | alumno 3         |
| 4  | alumno 4         |
| 5  | alumno 5         |
| 6  | alumno6          |
+----+------------------+
6 rows in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM ALUMNOS;
+----+------------------+
| id | alumno           |
+----+------------------+
| 1  | Alberto Carrera  |
| 2  | Raquel Carrera   |
| 3  | alumno 3         |
| 4  | alumno 4         |
| 5  | alumno 5         |
| 6  | alumno6          |
+----+------------------+
6 rows in set (0.00 sec)
```

Nota, como no se ha utilizado la instrucción de comienzo de transacción START TRANSACTION la transacción ha empezado de manera implícita.

3.3 Instrucciones de manejo de transacciones

- **START TRANSACTION**: Comienzo de una nueva transacción. Si ya existe una iniciada, esta última finaliza con confirmación de datos (COMMIT). Cuando comienza una nueva transacción, la propiedad **autocommit** automáticamente pasa a estado 0 (OFF apagado) hasta que finaliza la transacción.

- **COMMIT**: Termina la transacción guardando en la base de datos todos los cambios realizados por la transacción. Cualquier tipo de bloqueo que se mantuviera durante la transacción queda liberado.

- **ROLLBACK**: Termina la transacción deshaciendo todos los cambios que hubiera realizado sobre la base de datos. Libera los bloqueos que hubiera realizado la transacción.

- **SAVEPOINT** ***punto de salvaguarda***: Crea un punto de salvaguarda al que se puede retroceder mediante la instrucción ROLLBACK TO SAVEPOINT.

- **ROLLBACK TO SAVEPOINT** ***punto de salvaguarda***. Realiza un ROLLBACK de todas las sentencias ejecutadas desde que se creó el punto de salvaguarda.

- **SET TRANSACTION**: Permite cambiar el nivel de aislamiento de la transacción como se ha visto anteriormente.

- **LOCK TABLES**: Permite bloquear explícitamente una o varias tablas. A la vez cierra todas las transacciones abiertas.

Por defecto, MySQL efectúa un COMMIT implícito después de ejecutar cada instrucción DML SQL por lo que los cambios se guardan después de cada instrucción.

La propiedad (variable) **autocommit** que controla este comportamiento está puesta a 1.

Ilustración 6. Propiedad autocommit

Para poder trabajar con transacciones:

1º) Cambiar la propiedad **autocommit** para que no se haga un COMMIT automáticamente después de cada instrucción SQL:

```
SET autocommit=0
```

De todas formas, aunque es un buen método de trabajo en este y otros gestores, es redundante esta primera acción pues como se ha indicado la instrucción START TRANSACTION ya deja a 0 el valor de la propiedad anterior.

2º) Indicar el comienzo de la transacción con la instrucción START TRANSACTION

Nota: Independientemente del valor de la propiedad anterior, todas las sentencias DDL del lenguaje: ALTER, CREATE, DROP … tienen el COMMIT o confirmación de manera implícita o automática.

Un ejemplo sencillo de todo lo visto anteriormente:

```
2  DELIMITER $$
3  DROP PROCEDURE IF EXISTS transac1 $$
4  CREATE PROCEDURE transac1 (p_id INT, p_alumno VARCHAR(30))
5  MODIFIES SQL DATA
6  BEGIN
7      SET autocommit = 0;
8      START TRANSACTION;
9      INSERT INTO alumnos VALUES(p_id,p_alumno);
10     COMMIT;
11 END $$
```

Procedimiento 1 transac1

La llamada al procedimiento:

```
CALL transac1(7,'Mario A. Carrera')
```

insertará al alumno Mario.

En cambio:

```
CALL transac1(7, 'Carmen Bailin')
```

dará error de clave duplicada. Como el error no es tratado al no existir manejador de errores, la instrucción 10 no llegará a ejecutarse pero la transacción finalizará sin haber realizado la inserción.

Otro ejemplo de código y ejecución:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS transac2 $$
CREATE PROCEDURE transac2 (p_id INT, p_alumno VARCHAR(30),
                           OUT p_error_num INT, OUT p_error_text VARCHAR(100))
MODIFIES SQL DATA
BEGIN
    DECLARE clave_repetida_error CONDITION FOR 1062;
    DECLARE clave_nula_error CONDITION FOR 1048;
    DECLARE CONTINUE HANDLER FOR clave_repetida_error
        BEGIN
          SET p_error_num=1062;
          SET p_error_text='Clave duplicada';
        END;
    DECLARE CONTINUE HANDLER FOR clave_nula_error
        BEGIN
          SET p_error_num=1048;
          SET p_error_text='Clave nula';
         END;
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
        BEGIN
          SET p_error_num=-1;
          SET p_error_text='Ocurrió un error';
         END;
    SET p_error_num=0;
    SET autocommit = 0;
    START TRANSACTION;
    INSERT INTO alumnos VALUES(p_id,p_alumno);
    COMMIT;
    IF p_error_num=0 THEN
         SET p_error_text='Alta de alumno realizada';
    END IF;
END $$
```

Procedimiento 2 transac2

```
mysql> USE TEST
Database changed
mysql> call transac2(7, 'Blanca Bailin', @n_error, @texto_error);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @n_error, @texto_error;
+----------+-----------------+
| @n_error | @texto_error    |
+----------+-----------------+
| 1062     | Clave duplicada |
+----------+-----------------+
1 row in set (0.00 sec)

mysql> call transac2(8, 'Irene Yera', @n_error, @texto_error);
Query OK, 0 rows affected (0.03 sec)

mysql> SELECT @n_error, @texto_error;
+----------+--------------------------+
| @n_error | @texto_error             |
+----------+--------------------------+
| 0        | Alta de alumno realizada |
+----------+--------------------------+
1 row in set (0.00 sec)

/* Borrado de la tabla alumnos*/

mysql> call transac2(9, 'Ana Escalada', @n_error, @texto_error);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @n_error, @texto_error;
+----------+------------------+
| @n_error | @texto_error     |
+----------+------------------+
| -1       | Ocurri├| un error |
+----------+------------------+
1 row in set (0.00 sec)
```

Antes de ejecutar el tercer ejemplo asegúrate que tienes la tabla:

```
CREATE TABLE AUDITA (mensaje VARCHAR(200))
```

```
DELIMITER $$
DROP PROCEDURE IF EXISTS transac3 $$
CREATE PROCEDURE transac3 (    p_numce INTEGER,
                               p_nomce VARCHAR(25),
                               p_domicilio VARCHAR(30),
                               p_numde INTEGER,
                               p_nomde VARCHAR(20),
                               p_presu INTEGER)
MODIFIES SQL DATA
/* Resultado de la ejecución en tabla audita y no en variable OUT */
/* Lanzar antes CREATE TABLE AUDITA (mensaje VARCHAR(200))*/
BEGIN
     DECLARE v_existe_centro INTEGER DEFAULT 0;
     DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
        BEGIN
           INSERT INTO audita VALUES (CONCAT(NOW(),'-Ocurrió un error'));
        END;
     INSERT INTO audita VALUES (CONCAT('Comienzo de la ejecucion: ', NOW()));
     START TRANSACTION;
     SELECT COUNT(*)
         INTO v_existe_centro
     FROM centros
     WHERE numce=p_numce;
     IF v_existe_centro=0 THEN
        INSERT INTO audita VALUES (CONCAT(NOW(),': Dando de alta nuevo Centro: ', p_numce));
        INSERT INTO centros VALUES (p_numce,p_nomce,p_domicilio);
    END IF;

    SAVEPOINT fin_proceso_centro;

    BEGIN
         DECLARE v_departamento_duplicado TINYINT DEFAULT 0;
         DECLARE clave_duplicada_error CONDITION FOR 1062;
         DECLARE CONTINUE HANDLER FOR clave_duplicada_error
                 BEGIN
                    SET v_departamento_duplicado=1;
                    ROLLBACK TO SAVEPOINT fin_proceso_centro;
                 END;
        INSERT INTO audita VALUES (CONCAT(NOW(),': Dando de alta nuevo Departamento: ', p_numde));
        INSERT INTO departamentos (numde, numce, nomde ,presu)
                        VALUES (p_numde, p_numce, p_nomde, p_presu);
        IF v_departamento_duplicado = 1 THEN
              INSERT INTO audita VALUES (CONCAT(NOW(),': Departamento duplicado: ', p_numde));
        END IF;
    END;
    COMMIT;
    INSERT INTO audita VALUES (CONCAT('Fin de la ejecucion: ', NOW()));
END$$
```

Procedimiento 3 transac3

Nota, probar diferentes opciones. El punto de ruptura se utiliza para deshacer grabaciones en la tabla audita. Se deja al alumno averiguar su funcionamiento.

La ejecución de la llamada al procedimiento:

```
CALL transac3 (30, 'Centro 30', 'Los Olivos - Huesca', 140, 'Departamento 140', 40000)
```

Insertará la última fila en la tabla centros y departamentos:

numce	nomce	seas
10	SEDE CENTRAL	C/ ALCALA, 820 - MADRID
20	RELACION CON CLIENTES	C/ ATOCHA, 405 - MADRID
30	Centro 30	Los Olivos - Huesca

Ilustración 7. Tabla centros después de la última inserción

numde	numce	direc	tidir	presu	depde	nomde
▪▪▪▪		▪▪▪▪▪		▪▪▪▪▪▪		
140	30	NULL	NULL	40000	NULL	Departamento 140

Ilustración 8. Tabla departamentos después de la última inserción

y las siguientes entradas en la tabla audita:

mensaje
Comienzo de la ejecucion: 2006-06-14 09:55:25
2006-06-14 09:55:25: Dando de alta nuevo Centro: 30
2006-06-14 09:55:25: Dando de alta nuevo Departamento: 140
Fin de la ejecucion: 2006-06-14 09:55:25

Ilustración 9. Tabla audita después de la última inserción

Si volvemos a ejecutar la misma llamada, se añaden nuevas líneas:

Comienzo de la ejecucion: 2006-06-14 10:23:20
2006-06-14 10:23:20: Departamento duplicado: 140
Fin de la ejecucion: 2006-06-14 10:23:20

Ilustración 10. Tabla audita después de la última operación

3.4 Transacciones y bloqueos

Observa la siguiente secuencia de operaciones que se realizan en dos sucursales bancarias sobre una misma cuenta. La primera columna tiempo (Tº) indica el orden en que se efectúan:

Tº	SUCURSAL A	SUCURSAL B	SALDO INICIAL CUENTA
0			1.000 €
1	Disminuye 200 €		800 €
2		Aumenta 300 €	1.300 €
3	Confirma operación		800 €
4		Confirma operación	1.300 €

La situación anterior no debe aceptarse. Solución:

Tº	SUCURSAL A	SUCURSAL B	SALDO INICIAL CUENTA
0			1.000 €
1	Disminuye 200€		800€
2		La cuenta está bloqueada hasta que la Sucursal A confirme la operación	
3	Confirma operación		800€
4		Aumenta 300 €	1.100 €
5		Confirma operación	1.100 €

En la anterior solución cada transacción de programas, usuarios… espera a que acabe la transacción ya iniciada en un fila y que por tanto está bloqueada (no hay espera si la operación simplemente es una lectura de datos de la fila y no una modificación de los mismos).

Las situaciones en que se producen estos bloqueos son:

- Sentencia UPDATE: Las filas afectadas se bloquean hasta que se confirme o deshaga la transacción.

- Sentencia INSERT: Si existe clave primaria, las filas insertadas quedan bloqueadas para prevenir que otra transacción pueda introducir otra fila con la misma clave.

- Sentencia LOCK TABLES: Bloquea la tabla entera. No es muy eficiente pues reduce la concurrencia.

- Si dentro de una sentencia SELECT se utilizan las clausulas FOR UPDATE o LOCK IN SHARE MODE, todas las filas devueltas por la sentencia SELECT serán bloqueadas:

```
SELECT opciones de la sentencia SELECT
      [FOR UPDATE | LOCK IN SHARE MODE]
```

La cláusula LOCK IN SHARE MODE suele utilizarse para garantizar la integridad referencial. Imagina que estás intentando introducir los datos de un nuevo departamento al que le asignarás un determinado centro. Existe la clave ajena: departamentos.numce → centros.numce. Puede ocurrir que durante el proceso, otra transacción elimine la fila correspondiente del centro en la tabla Centros. Al intentar introducir el departamento aparecerá una situación de error. Por eso, en estos casos, en primer lugar hay que asegurarse de que nadie modificará el Centro:

```
SELECT * FROM CENTROS WHERE NUMCE=XX LOCK IN SHARE MODE
```

Por otro lado la solución anterior no es buena para garantizar la clave única en una tabla. Imagina que cada departamento se obtiene sumando 10 al número último asignado. Si utilizamos la cláusula LOCK IN SHARE MODE puede ocurrir que dos transacciones lean al mismo tiempo el último id asignado, le sumen 10 y a la hora de modificarlo provoquen una situación de clave duplicada o como se ve más adelante una situación de abrazo mortal (**deadlock**). La solución para este caso:

```
SELECT id FROM tablax… FOR UPDATE
UPDATE tablax SET id…
```

en la que primero se intenta leer con intención de modificar, utilizando además un bloqueo FOR UPDATE y luego, si ha sido posible realizar la operación anterior, entonces incrementarlo (UPDATE tablax…).
El nivel de bloqueo es el mismo que para un UPDATE.
Los bloqueos son levantados cuando la transacción se confirma o deshace.

3.4.1 Deadlock. Cursores de actualización

También conocido como abrazo mortal. Esta situación ocurre cuando una transacción A intenta modificar los datos que están siendo modificados por una transacción B y a su vez esta última intenta modificar los datos que están siendo modificados por la primera transacción A.

Tº	SUCURSAL A	SUCURSAL B	SITUACION
1	Modifica saldo cuenta X		La cuenta X queda bloqueada por A
2		Modifica saldo cuenta Y	La cuenta Y queda bloqueada por B
3	Modifica saldo cuenta Y		A queda a la espera pues Y está bloqueada
4		Modifica saldo cuenta X	B queda a la espera pues X está bloqueada

Como consecuencia del abrazo mortal anterior, una de las dos transacciones realizará un ROLLBACK provocando una situación de error.

Volvemos a la situación inicial de la tabla alumnos, 5 filas con 5 alumnos de id's 1 al 5 y de nombres alumno1…alumno5.

Abrimos 2 sesiones A y B:

```
A
mysql> USE TEST;
Database changed
mysql> SET autocommit = 0;
```

```
B
mysql> USE TEST;
Database changed
mysql> SET autocommit = 0;
```

```
A
mysql> UPDATE ALUMNOS SET alumno='Alberto Carrera' WHERE id=1;
/* A ya puede ver el cambio producido pero B seguirá viendo alumno1
como nombre en lugar de Alberto Carrera */
```

```
B
mysql> UPDATE ALUMNOS SET alumno='Raquel Carrera' WHERE id=1;
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
mysql> _
```

Ilustración 11. Tiempo de espera superado

```
Después de un tiempo de espera de B, aparece el error de la
ilustración anterior, al intentar modificar una fila que está
siendo modificada y todavía no ha sido confirmada o deshecha por
A.

mysql> UPDATE ALUMNOS SET alumno='Mario Carrera' WHERE id=2;
```

```
A
mysql>UPDATE ALUMNOS SET alumno='Carmen Bailin' WHERE id=2;
```

```
B
Si ejecutamos rápidamente:
mysql>UPDATE ALUMNOS SET alumno='Raquel Carrera' WHERE id=1;
```

```
mysql> UPDATE ALUMNOS SET alumno='Raque Carrera' WHERE id=1;
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting trans
action
mysql>
```

Ilustración 12. Abrazo mortal

A

```
mysql> SELECT * FROM ALUMNOS;
+----+-----------------+
| id | alumno          |
+----+-----------------+
| 1  | Alberto Carrera |
| 2  | Carmen Bailin   |
| 3  | alumno 3        |
| 4  | alumno 4        |
| 5  | alumno 5        |
+----+-----------------+
5 rows in set (0.00 sec)
```

Ilustración 13. Tabla alumnos vista por el usuario A

B

```
mysql> SELECT * FROM ALUMNOS;
+----+----------+
| id | alumno   |
+----+----------+
| 1  | alumno 1 |
| 2  | alumno 2 |
| 3  | alumno 3 |
| 4  | alumno 4 |
| 5  | alumno 5 |
+----+----------+
```

Ilustración 14. Tabla alumnos vista por el usuario B

```
A
mysql>COMMIT;
```

```
B
Sigue viendo la pantalla anterior última (ilustración 14).

mysql>COMMIT;

Ve la pantalla penúltima (ilustración 13)
```

Para probar el cuarto ejemplo hay que lanzar antes otra vez el procedimiento *procedimiento1* para crear de nuevo la tabla *alumnos*.

```
DELIMITER $$
DROP PROCEDURE IF EXISTS transac4 $$
CREATE PROCEDURE transac4 (p_id INT, p_alumno VARCHAR(30),
                           OUT p_error_num INT, OUT p_error_text VARCHAR(100))
MODIFIES SQL DATA
BEGIN
    DECLARE clave_repetida_error CONDITION FOR 1062;
    DECLARE clave_nula_error CONDITION FOR 1048;
    DECLARE tiempo_excedido_error CONDITION FOR 1205;
    DECLARE abrazo_mortal_error CONDITION FOR 1213;
    DECLARE CONTINUE HANDLER FOR clave_repetida_error
        BEGIN
          SET p_error_num=1062;
          SET p_error_text='Clave duplicada';
        END;
    DECLARE CONTINUE HANDLER FOR clave_nula_error
        BEGIN
          SET p_error_num=1048;
          SET p_error_text='Clave nula';
         END;
    DECLARE EXIT HANDLER FOR tiempo_excedido_error
         BEGIN
          SET p_error_num=1205;
          SET p_error_text='Tiempo de espera excedido';
         END;
    DECLARE EXIT HANDLER FOR abrazo_mortal_error
         BEGIN
           ROLLBACK;
          SET p_error_num=1213;
          SET p_error_text='Abrazo mortal';
         END;
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
        BEGIN
          SET p_error_num=-1;
          SET p_error_text='Ocurrió un error';
         END;
    SET p_error_num=0;
    -- SET autocommit = 0;
    START TRANSACTION;
    INSERT INTO alumnos VALUES(p_id,p_alumno);
    COMMIT;
    IF p_error_num=0 THEN
         SET p_error_text='Alta de alumno realizada';
    END IF;
END $$
```

Procedimiento 4transac4

```
A
USE TEST;
SET autocommit = 0;
```

```
B
USE TEST;
SET autocommit = 0;
```

```
A
UPDATE alumnos SET alumno= 'Mario Carrera'
WHERE id=1;
```

```
B
CALL transac4 (1, 'Carmen Bailin', @p n error, @p text error);
/* ……Después de un tiempo de espera - 50 segundos en el que parece que se ha quedado
colgado el programa */
Query OK, 0 rows affected (51.55 sec)
SELECT @p n error, @p text error;
+------------+---------------------------+
| @p n error | @p text error             |
+------------+---------------------------+
| 1205       | Tiempo de espera excedido |
+------------+---------------------------+
```

A continuación una solución para evitar el DEADLOCK: Cursores de actualización (CURSOR FOR UPDATE)

```
DELIMITER $$
DROP PROCEDURE IF EXISTS transac5 $$
CREATE PROCEDURE transac5 (p_id INT, p_alumno VARCHAR(30),
                           OUT p_error_num INT, OUT p_error_text VARCHAR(100))
MODIFIES SQL DATA
BEGIN
    DECLARE v_alumno VARCHAR(30);
    DECLARE alumnos_cursor CURSOR FOR
        SELECT alumno
          FROM alumnos
         WHERE id = p_id
        FOR UPDATE;
    /*                                                                    */
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
        BEGIN
          SET p_error_num=-1;
          SET p_error_text='Ocurrió un error';
         END;
    SET p_error_num=0;
    -- SET autocommit = 0;
    START TRANSACTION;
    OPEN alumnos_cursor;
    FETCH alumnos_cursor INTO v_alumno;
    UPDATE alumnos
      SET alumno= p_alumno
    WHERE id=p_id;
    CLOSE alumnos_cursor;
    COMMIT;
    IF p_error_num=0 THEN
         SET p_error_text='Alta de alumno realizada';
    END IF;
END $$
```

Procedimiento 5 transac5

Antes se finalizar este apartado señalar que, como hemos comprobado al intentar escribir en una fila bloqueada (sin ser deadlock), se ha producido un tiempo de espera de 50 segundos tras el que se ha producido un error de tiempo de espera excedido (1205) y la transacción se ha deshecho.

La variable de sistema **innodb_lock_wait_timeout** es la que almacena el tiempo máximo de espera. Dicha variable puede modificarse en caso de aplicaciones con largas transacciones. También se puede como se ha hecho antes crear un manejador de error.

3.5 Estrategias de bloqueo

Si una transacción tiene necesidad de leer datos que posteriormente se pueden ver afectados por operaciones de manipulación (INSERT, UPDATE, DELETE), hay que tomar precauciones para que otra transacción no pueda modificar unos datos después de que hayan sido leídos por la primera transacción y antes de ser modificados por esta. Ej.

Tº	SUCURSAL A	SUCURSAL B	SITUACION
0			SALDO INICIAL CUENTA X = 1.000 €
1	LEE SALDO CON INTENCIÓN DE DISMINUIRLO EN 900 € SI ES POSIBLE PARA EVITAR DEJAR LA CUENTA EN NÚMEROS ROJOS		1.000€
2		DISMINUYE LA CUENTA EN 500 €	500€
3	COMO EL SALDO PARA LA SUCURSAL A ES DE 1.000€, A CONTINUACIÓN DISMINUYE EL SALDO EN 900€		- 400€

Posibles soluciones:

- **Estrategia de bloqueo pesimista**: El planteamiento para este caso es que las transacciones concurrentes es muy fácil de que ocurran y por tanto hay que estar prevenido. Por tanto habrá que bloquear las filas después de leerlas. Otras transacciones que quieran modificar los datos deberán esperar.

- **Estrategia de bloqueo optimista**: Asume que es muy poco probable que el valor de una fila que acabamos de leer cambie. Si asumimos este planteamiento como mínimo deberemos asegurarnos de que la fila no ha sido modificada y si así ha sido entonces la transacción no debe llevarse a cabo aun pudiéndose realizar.

A continuación un ejemplo de implementación de la estrategia pesimista:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS transac6 $$
CREATE PROCEDURE transac6 (p_id INT, p_alumno VARCHAR(30),
                           OUT p_error_num INT,
                           OUT p_error_text VARCHAR(100))
MODIFIES SQL DATA
BEGIN
    DECLARE v_alumno VARCHAR(30);
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
        BEGIN
          SET p_error_num=-1;
          SET p_error_text='Ocurrió un error';
         END;
    SET p_error_num=0;
    -- SET autocommit = 0;
    START TRANSACTION;
    SELECT alumno
      INTO v_alumno
      FROM alumnos
     WHERE id=p_id
     FOR UPDATE;
    UPDATE alumnos
      SET alumno= p_alumno
    WHERE id=p_id;
    COMMIT;
    IF p_error_num=0 THEN
         SET p_error_text='Alta de alumno realizada';
    END IF;
END $$
```

Procedimiento 6 transac6

Es una estrategia muy segura pues asegura la consistencia entre la lectura (SELECT) y la operación DML (UPDATE), pero limita mucho el rendimiento del sistema al obligar a las transacciones a largas esperas para poder completarse.

En la estrategia optimista, como la transacción no bloquea la fila que lee, antes de que realice la modificación sobre la misma debe asegurarse que el valor de la fila no ha cambiado desde que la leyó; en caso de que no se hubieran producido cambios se realizará y confirmará la transacción, en caso contrario no se llevará a cabo.

Para la elección de una estrategia u otra hay que tener en cuenta la concurrencia y robustez: la estrategia pesimista supone menos errores y reintentos mientras que con la optimista se reduce el tiempo de duración de los bloqueos aumentando por tanto la concurrencia y el rendimiento del sistema.
Generalmente suele utilizarse la estrategia pesimista y sólo se recurre a la optimista si la duración de los bloqueos o el número de filas que se bloquean es elevado en la estrategia pesimista. De todas formas el uso de una u otra depende mucho de las características de la aplicación.

3.6 Aspectos a tener en cuenta a la hora de utilizar transacciones

De todo lo visto anteriormente, podemos resumir:

- Siempre debe mantenerse la integridad de la base de datos para garantizar que los datos que almacenan son precisos, fieles a la realidad y lo más correctos posibles.

- La duración de los bloqueos debe ser lo mínimo posible. De igual manera debe ser también mínimo el número de filas a bloquear por las transacciones.

- La operación de ROLLBACK (deshacer) debe evitarse en los casos que se pueda pues es una operación costosa en el tiempo y en utilización de recursos. Por eso, habrá que adelantarse a las situaciones que pueden provocarlo como los intentos de inserción de filas de clave duplicada, realizando en este caso búsquedas antes de las inserciones. Relacionado con este punto, se evitarán en la medida en que se pueda la creación y utilización de puntos de salvaguarda; sólo si el número de situaciones posibles a controlar es elevado, quizás en ese caso sea rentable y aceptable su utilización Indicar también que la operación de confirmación o COMMIT al igual que la de ROLLBACK es una operación costosa pues trae consigo escritura física en disco de los datos de la caché de memoria del gestor de bases de datos. Por eso hay que utilizar esta instrucción de confirmación en los sistemas en tiempo real de un negocio para mantener los datos íntegros pero hay que minimizar su uso en situaciones de carga masiva de datos a través de scripts por ejemplo.

- Considerar siempre la estrategia de bloqueo pesimista salvo en los casos del final del apartado anterior que podrían llegar a causar frustración en el usuario por el bajo rendimiento del sistema.

- En cada programa debe especificarse explícitamente el comienzo y fin de transacción y no asumir que los programas externos son los que validarán o desharán la transacción.

Anexo: Manejo de transacciones desde otras aplicaciones

De la misma manera y con la misma intención con que se ha expuesto el anexo del último apartado del tema 2, a continuación se indica cómo son manejadas las transacciones MySQL desde otras aplicaciones externas a ella.

1 Desde PHP

En el siguiente ejemplo se intenta llevar a cabo una transacción consistente en cambiar el nombre del alumno de matrícula 1. Si todo va bien aparecerá el mensaje “Filas

afectadas: 1 Transacción completada" llevándose a cabo la transacción y validándose mediante la instrucción de la línea 28. Si no puede realizarse es deshecha (línea 23).

```
10 /*.....................................................................
11 $dbh = new mysqli($hostname, $username, $password, $database);
12     /* Comprobar conexión */
13     if (mysqli_connect_errno()) {
14         printf("Conexión fallida: %s\n", mysqli_connect_error());
15         exit ();
16     }
17 $dbh->autocommit(FALSE);
18 $dbh->query("UPDATE alumnos
19                   SET alumno='Mario Carrera'
20               WHERE id=1");
21 if ($dbh->errno)       {
22       printf("Transacción fallida: %s\n",$dbh->error);
23       $dbh->rollback(  );
24    }
25    else       {
26       printf ("Filas afectadas: %d \n",$dbh->affected_rows);
27       printf("Transacción completada\n");
28       $dbh->commit(  );
29    }
30 $dbh->close();
31 ?>
```

Ilustración 15. Manejando transacciones desde PHP

2 Desde Visual Basic Express 2005

Idéntica operación que en el apartado 1 de este anexo:

```
'.................
       Dim MiConexion As New MySqlConnection(cadena_conexion)
       Dim cadena_sql As String = "UPDATE alumnos" + _
                " SET alumno='Mario Carrera' " + _
                " WHERE id=1"
       Dim instruccion As MySqlCommand = New MySqlCommand(cadena_sql, MiConexion)
       MiConexion.Open()
       Dim MiTransaccion As MySqlTransaction = MiConexion.BeginTransaction
       Try
           Dim filas_afectadas As Integer
           filas_afectadas = instruccion.ExecuteNonQuery()
           Console.WriteLine("Filas afectadas: " + filas_afectadas.ToString)
           MiTransaccion.Commit()
           Console.WriteLine("Transacción finalizada")
       Catch Exception As MySqlException
           Console.WriteLine("Error en la transacción: ")
           Console.WriteLine(Exception.Message)
           Try
               MiTransaccion.Rollback()
               Console.WriteLine("Transaction no realizada")
           Catch RollbackException As MySqlException
               Console.WriteLine("Rollback fallido:")
               Console.WriteLine(RollbackException.Message)
           End Try
       End Try
       '.................
```