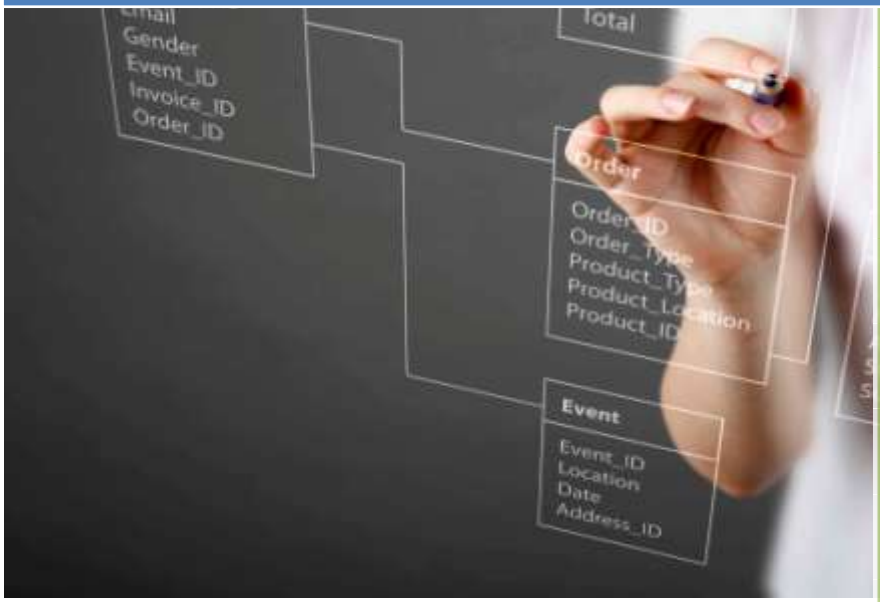
Base de Datos

6. Instrucciones para la Manipulación de Datos (DML)



Curso 2022-2023

Tabla de contenido

1. INTRODUCCIÓN

Mediante la sentencia podemos consultar datos, pero, ¿cómo los podemos manipular, definir y controlar?

El lenguaje SQL aporta una serie de instrucciones con las que se pueden realizar las acciones siguientes:

- La manipulación de los datos (instrucciones **DML** que nos deben permitir efectuar altas, bajas y modificaciones).
- La definición de datos (instrucciones **DDL** que nos deben permitir crear, modificar y eliminar las tablas, los índices y las vistas).
- El control de datos (instrucciones **DCL** que nos deben permitir gestionar los usuarios y sus privilegios).

Acrónimos

Recordemos los acrónimos para los diferentes apartados del lenguaje SQL: **CL** (lenguaje de consulta); **DML** (lenguaje de manipulación de datos); **DDL** (lenguaje de definición de datos); **DCL** (lenguaje de control de datos).

El lenguaje SQL proporciona un conjunto de instrucciones, reducido pero muy potente, para manipular los datos, dentro del cual se debe distinguir entre dos tipos de instrucciones:

- Las instrucciones que permiten ejecutar la manipulación de los datos, y que se reducen a tres: para la introducción de nuevas filas, UPDATE para la modificación de filas, y para el borrado de filas.
- Las instrucciones para el control de transacciones, que deben permitir asegurar que un conjunto de operaciones de manipulación de datos se ejecute con éxito en su totalidad o, en caso de problema, se aborte totalmente o hasta un determinado punto en el tiempo.

Antes de introducirnos en el estudio de las instrucciones ***INSERT***, ***UPDATE*** y ***DELETE***, es necesario conocer cómo el SGBD gestiona las instrucciones de inserción, eliminación y modificación que podamos ejecutar, ya que hay dos posibilidades de funcionamiento:

- Que queden automáticamente validadas y no haya posibilidad de tirar atrás. En este caso, los efectos de toda instrucción de actualización de datos que tenga éxito son automáticamente accesibles desde el resto de conexiones de la base de datos.
- Que queden en una cola de instrucciones, que permite tirar atrás. En este caso, se dice que las instrucciones de la cola están pendientes de validación, y el usuario debe ejecutar, cuando lo cree conveniente, una instrucción para validarlas (llamada *COMMIT*) o una instrucción para tirar atrás (llamada ROLLBACK). Más adelante ampliaremos estos dos comandos.

Este funcionamiento implica que los efectos de las instrucciones pendientes de validación no se ven por el resto de conexiones de la base de datos, pero sí son accesibles desde la conexión donde se han efectuado. Al ejecutar la ***COMMIT***, todas las conexiones acceden a los efectos de las instrucciones validadas. En caso de ejecutar

ROLLBACK, las instrucciones desaparecen de la cola y ninguna conexión (ni la propia ni el resto) no accede a los efectos correspondientes, es decir, es como si nunca hubieran existido.

Estos posibles funcionamientos forman parte de la gestión de transacciones que proporciona el SGBD y que hay que estudiar con más detenimiento. A la hora, sin embargo, de ejecutar instrucciones ***INSERT***, ***UPDATE*** y ***DELETE*** debemos conocer el funcionamiento del SGBD para poder actuar en consecuencia.

Así, por ejemplo, un SGBD MySQL funciona con validación automática tras cada instrucción de actualización de datos a menos que se indique lo contrario y, en cambio, un SGBD Oracle funciona con la cola de instrucciones pendientes de confirmación o rechazo que debe indicar el usuario.

En cambio, en MySQL, si se quiere desactivar la opción de ***autocommit*** que hay por defecto, habrá que ejecutar la siguiente instrucción:

```
SET autocommit=0;
```

2. SENTENCIA INSERT

La sentencia **INSERT** es la instrucción proporcionada por el lenguaje SQL para **insertar** nuevas filas en las tablas.

Admite dos sintaxis:

1. Los valores a insertar deben insertarse se explicitan en la misma instrucción en la cláusula : ***values***

```
INSERT INTO <nom_tabla>[(col1, col2…)]
VALUES (val1, val2...);
```

2. Los valores que deben insertarse se consiguen por medio de una sentencia: ***SELECT***

```
INSERT INTO <nom_tabla> [(col1, col2...)]
SELECT...;
```

En todo caso, se pueden especificar las columnas de la tabla que se deben rellenar y el orden en que se suministran los diferentes valores. En caso de que no se especifiquen las columnas, el SQL entiende que los valores se suministran para todas las columnas de la tabla y, además, en el orden en que están definidos en la tabla.

La lista de valores de la cláusula y la lista de resultados de la sentencia SELECT deben coincidir en número, tipo y orden con la lista de columnas a cumplimentar.

Ejemplo 1 de sentencia INSERT

En el esquema empresa, se solicita insertar el departamento 50 de nombre 'INFORMÁTICA'.

La posible sentencia para conseguir el objetivo es esta:

```
INSERT INTO departamento (dept_no, dnom) VALUES (50, 'INFORMÁTICA');
```

Si ejecutamos una consulta para comprobar el contenido actual de la tabla, encontraremos la nueva fila sin localidad asignada (NULL). El SGBD ha permitido dejar la *localidad* (LOC) con valor NULO porque lo tiene permitido así, como se puede ver en el descriptor de la tabla

```
SQL> desc dept;
Name                 Null     Type
-------------------- -------- ----------------------------
DEPT_NO              NOT NULL NUMBER(2)
DNOM                 NOT NULL VARCHAR2(14)
LOC                           VARCHAR2(14)
3 rows selected
```

Ejemplo 2 de sentencia INSERT

En el esquema sanidad, se pide dar de alta al doctor de código 100 y nombre 'BARRUFET D.'.

La solución parece que podría ser esta:

```
INSERT INTO doctor (doctor_no, nombre)
VALUES (100, 'BARRUFET D.');
```

Al ejecutar esta sentencia, el SGBDR da un error.

Lo cierto es que la tabla no admite valores nulos en la columna *hospital_cod*, ya que esta columna forma parte de la clave primaria. Miremos el descriptor de la tabla :

```
SQL> desc doctor;
Name                 Null     Type
-------------------- -------- ---------------
HOSPITAL_COD         NOT NULL NUMBER(2)
DOCTOR_NO            NOT NULL NUMBER(3)
NOMBRE               NOT NULL VARCHAR2(13)
ESPECIALIDAD         NOT NULL VARCHAR2(16)
4 rows selected
```

Aparte de la columna *hospital_cod*, también deberíamos dar un valor en la columna *especialidad*, ya que tampoco admite valores nulos.

Recordemos que, en nuestro esquema sanidad, la columna *especialidad* es una cadena que no tiene ningún tipo de restricción definida ni es clave foránea de ninguna Tabla en la que haya todas las especialidades posibles. Por lo tanto, si queremos saber qué especialidades hay para escribir la del doctor que queremos insertar, idénticamente a las ya introducidas en caso de que hubiera algún doctor con la misma especialidad del que queremos insertar, hacemos lo siguiente:

```
SQL> select distinct especialidad from doctor;

ESPECIALIDAD

Urologia

Pediatria

Cardiologia

Neurologia

Ginecologia

Psiquiatria

6 rows selected
```

Suponemos que el doctor 'BARRUFET D.' es psiquiatra. Como ya hay algún doctor con la especialidad 'Psiquiatría', correspondería hacer la inserción utilizando la misma grafía para la especialidad. Además, suponemos que queremos dar de alta al doctor en el hospital 66.

```
INSERT INTO doctor (doctor_no, nombre, hospital_cod,
especialidad)
VALUES (100, 'BARRUFET D.', 66, 'Psiquiatria');
```

Esta vez, el SGBD también se nos queja con otro tipo de error: ha fallado la referencia a la clave foránea.

El error nos informa de que una restricción de integridad definida en la Tabla ha intentado ser violada y, por tanto, la instrucción no ha finalizado con éxito. El SGBD nos pasa dos informaciones para que tengamos pistas de dónde está el problema:

- Nos da una descripción breve del problema (no se puede añadir una fila hija -child row-), la cual nos da a entender que se trata de un error de clave foránea, es decir, que no existe el código en la tabla referenciada.
- Nos dice la restricción que ha fallado `().sanidad.doctor, CONSTRAINT … FOREIGN KEY (HOSPITA_COD) …`

El SGBD tiene toda la razón. Recordemos que la columna *hospital_cod* de la tabla *DOCTOR* es clave foránea de la tabla *HOSPITAL*. Esto quiere decir que cualquier inserción en la Tabla *DOCTOR* debe ser para hospitales existentes en la Tabla *HOSPITAL*, y eso no pasa con el hospital 66, como se puede ver al consultar los hospitales existentes:

```
SQL> SELECT * FROM hospital;

+--------------+------------+----------------------+----------+-----------+
| HOSPITAL_COD | NOMBRE     | DIRECCION            | TELEFONO | NUM_CAMAS |
+--------------+------------+----------------------+----------+-----------+
|           13 | Provincial | O Donell 50          | 964-4264 |        88 |
|           18 | General    | Atocha s/n           | 595-3111 |        63 |
|           22 | La Paz     | Castellana 1000      | 923-5411 |       162 |
|           45 | San Carlos | Ciudad Universitaria | 597-1500 |        92 |
+--------------+------------+----------------------+----------+-----------+
4 rows in set (0.092 sec)
```

Así pues, o nos hemos equivocado de hospital o debemos dar de alta previamente el hospital 66. Suponemos que es el segundo caso y que, por tanto, tenemos que dar de alta el hospital 66:

```
insert into hospital (hospital_cod, nombre, direccion)
values (66,'General','De la font, 13');
```

El SGBD nos acepta la instrucción. Fijémonos que hemos informado del código de hospital, del nombre y de la dirección. Miremos el descriptor de la tabla HOSPITAL:

```
SQL> desc hospital;
+--------------+----------------------+------+-----+---------+-------+
| Field        | Type                 | Null | Key | Default | Extra |
+--------------+----------------------+------+-----+---------+-------+
| HOSPITAL COD | tinyint(2)           | NO   | PRI | NULL    |       |
| NOMBRE       | varchar(10)          | NO   | MUL | NULL    |       |
| DIRECCION    | varchar(20)          | YES  |     | NULL    |       |
| TELEFONO     | varchar(8)           | YES  |     | NULL    |       |
| NUM CAMAS    | smallint(3) unsigned | YES  |     | 0       |       |
+--------------+----------------------+------+-----+---------+-------+
5 rows selected
```

En él vemos cinco campos, de los cuales sólo los dos primeros tienen marcada la obligatoriedad de valor. Por tanto, no se nos ha quejado porque no hayamos indicado el teléfono del hospital ni la cantidad de camas que tiene el hospital ya que estos pueden ser nulos por definición.

Comprobamos la información que ahora hay en la tabla:

```
SQL> SELECT * FROM hospital;
+--------------+------------+----------------------+----------+-----------+
| HOSPITAL_COD | NOMBRE     | DIRECCION            | TELEFONO | NUM_CAMAS |
+--------------+------------+----------------------+----------+-----------+
|           13 | Provincial | O Donell 50          | 964-4264 |        88 |
|           18 | General    | Atocha s/n           | 595-3111 |        63 |
|           22 | La Paz     | Castellana 1000      | 923-5411 |       162 |
|           45 | San Carlos | Ciudad Universitaria | 597-1500 |        92 |
+--------------+------------+----------------------+----------+-----------+
4 rows in set (0.001 sec)
```

¡Sorpresa! Para el nuevo hospital, la columna *telefono* no tiene valor (valor NULL), pero la columna *num_camas* tiene el valor 0. ¿De dónde ha salido? Esto se debe a que la columna *num_camas* de la tabla tiene definido el valor por defecto (0) que el SGBD utiliza para rellenar la columna *num_camas* cuando se produce una inserción en la tabla sin indicar valor para esta columna.

Ahora parece que ya podemos insertar a nuestro doctor 'BARRUFET D'.:

```
INSERT INTO doctor (doctor_no, cognom, hospital_cod,
especialitat)
VALUES (100, 'BARRUFET D.', 66, 'Psiquiatria');
```

No nos olvidemos de registrar los cambios con la instrucción o de hacer, si tenemos el *autocommit* desactivado.

Ejemplo 3 de sentencia INSERT

Antes de empezar, desactivaremos el autocommit que tiene configurado por defecto MySQL para poder practicar:

```
SET AUTOCOMMIT=0;
```

En el esquema empresa, se quiere insertar el pedido identificado por el número 1.000, con fecha de pedido el 1 de septiembre de 2000 y para el cliente 500.

Quizás debemos conocer, en primer lugar, el descriptor de la tabla:

```
SQL> DESC pedido;
+----------------+----------------------+------+-----+---------+-------+
| Field          | Type                 | Null | Key | Default | Extra |
+----------------+----------------------+------+-----+---------+-------+
| PEDIDO_NUMERO  | smallint(4) unsigned | NO   | PRI | NULL    |       |
| PEDIDO_FECHA   | date                 | YES  | MUL | NULL    |       |
| PEDIDO_TIPO    | char(1)              | YES  |     | NULL    |       |
| CLIENTE_CODIGO | int(6) unsigned      | NO   | MUL | NULL    |       |
| FECHA_ENVIO    | date                 | YES  | MUL | NULL    |       |
| TOTAL          | decimal(8,2) unsigned | YES  |     | NULL    |       |
+----------------+----------------------+------+-----+---------+-------+
```

Fijémonos que tenemos la información correspondiente a todos los campos obligatorios. Por lo tanto, podemos ejecutar lo siguiente:

```
INSERT INTO pedido (PEDIDO_NUMERO, PEDIDO_FECHA,
CLIENTE_CODIGO)
VALUES (1000,'2000/09/01',500);
```

El SGBD nos reporta el error de restricción de integridad sobre la clave foránea. Y, por descontado, el SGBD vuelve a tener razón, ya que en el esquema *empresa* la tabla tiene una restricción de clave foránea en la columna cliente_codigo.

Si consultamos el contenido de la tabla, veremos que no hay ningún cliente con código 500. Por ello, el SGBD ha dado un error. Suponemos que era un error nuestro y la orden

correspondía al cliente 109 (que sí existe en la tabla CLIENTE). Esta vez la siguiente instrucción no nos da ningún problema.

```
INSERT INTO pedido (PEDIDO_NUMERO, PEDIDO_FECHA, CLIENTE_CODIGO)
VALUES (1000,'2000/09/01',109);
```

Podemos comprobar cómo ha quedado insertada la orden:

```
SELECT * FROM PEDIDO WHERE PEDIDO_NUMERO=1000;
+---------------+--------------+-------------+----------------+-------------+-------+
| PEDIDO_NUMERO | PEDIDO_FECHA | PEDIDO_TIPO | CLIENTE_CODIGO | FECHA_ENVIO | TOTAL |
+---------------+--------------+-------------+----------------+-------------+-------+
|          1000 | 2000-09-01   | NULL        |            109 | NULL        |  NULL |
+---------------+--------------+-------------+----------------+-------------+-------+
1 row in set (0.009 sec)
```

Hacemos *rollback* para tirar atrás la inserción efectuada y así poder comprobar que también la podríamos hacer de diferentes maneras. Recordemos que no es obligatorio indicar las columnas para las que se introducen los valores. En este caso, el SGBD espera todas las columnas de la tabla en el orden en que están definidas en la tabla. Así pues, podemos hacer lo siguiente:

```
INSERT INTO PEDIDO
VALUES (1000,'2000/09/01',NULL,109,NULL,NULL);
```

Hacemos *rollback* para probar otra posibilidad. Fijémonos que el SGBD también nos deja introducir un precio total de orden cualquiera:

```
ROLLBACK;
INSERT INTO PEDIDO
VALUES (1000,DATE '2000-09-01',NULL,109,NULL,9999);
COMMIT;
```

El SGBDR ha aceptado esta sentencia y ha insertado la fila correspondiente. Sin embargo, hemos introducido un importe total que no se corresponde con la realidad, ya que no hay ninguna línea de detalle. Es decir, el valor 9999 no es válido! Los SGBD proporcionan mecanismos (*disparadores*) para controlar este tipo de incoherencias de los datos.

Disparadores

Un ***disparador*** (tambien llamado ***trigger***) es un conjunto de instrucciones que se ejecutan automáticamente ante un evento determinado. Así, podemos controlar que, al insertar, borrar o modificar filas de detalle de una orden, el importe total de la orden se actualice automáticamente.

Ejemplo 4 de sentencia INSERT

En primer lugar, volvemos a activar la opción de autocommit para que sea más cómoda el trabajo.

```
SET AUTOCOMMIT=1;
```

Como detalle del pedido 1000 insertada en el ejemplo anterior, en el esquema empresa se quieren insertar las mismas líneas que contiene el pedido 620.

En este caso, ejecutaremos una instrucción tomando como valores que hay que insertar los que nos da el resultado de una sentencia :

```
INSERT INTO detalle
SELECT 1000, detalle_numero, producto_numero, precio_venta, cantidad, importe
FROM detalle
WHERE pedido_numero =620;
```

En esta instrucción, hemos seleccionado las filas de detalle del orden 620 y las hemos insertado como filas de detalle del pedido 1000. Debemos ser conscientes de que el importe total del pedido 1000 sigue siendo, sin embargo, incorrecto.

Como ya hemos comentado, hemos utilizado una sentencia para insertar valores en una tabla. Es una coincidencia que ambas sentencias actúen sobre la misma Tabla .

Al no indicar, en la sentencia INSERT, las columnas en que deben insertarse los valores, ha sido necesario construir la sentencia SELECT de modo que las columnas de la cláusula coincidieran, en orden, con las columnas de la tabla en que debe efectuarse la inserción. Además, como para todas las filas del pedido 620 había que indicar 1000 como número de pedido, la cláusula ha incorporado la constante 1000 como valor para la primera columna.

3. SENTENCIA UPDATE

La sentencia es la instrucción proporcionada por el lenguaje SQL para modificar filas que hay en las tablas.

Su sintaxis es esta:

```
UPDATE <nom_TABLA>
SET col1=val1, col2=val2, col3=val3...
[WHERE <condición>];
```

La cláusula optativa selecciona las filas a actualizar. En caso de inexistencia, se actualizan todas las filas de la tabla. `where`

La cláusula indica las columnas a actualizar y el valor con el que se actualizan. `set`

El valor de actualización de una columna puede ser el resultado obtenido por una sentencia SELECT que recupera una única fila:

```
UPDATE <nom_TABLA>
SET col1=(SELECT exp1 FROM ... ),
SET col2=(SELECT exp2 FROM ... ),
SET col3=val3,
...
[WHERE <condición>];
```

En tales situaciones, la sentencia es una subconsulta de la sentencia que puede utilizar valores de las columnas de la fila que se está modificando en la sentencia.

Como a veces es posible que haya que actualizar los valores de más de una columna a partir de diferentes resultados de una misma sentencia, no sería nada eficiente ejecutar varias veces la misma sentencia para actualizar más de una columna. Por tanto, la sentencia también admite la siguiente sintaxis:

```
UPDATE <nom_tabla>
SET (col1, col2)=(SELECT exp1, exp2 FROM
... ),
SET col3=val3,
...
[WHERE <condición>];
```

Ejemplo 1 de sentencia UPDATE

En el esquema empresa, se quiere modificar la localidad de los departamentos de manera que queden todos los caracteres con minúsculas.

La instrucción para resolver la solicitud puede ser esta:

```
UPDATE departamento
SET loc = LOWER(loc);
```

Ahora se quiere modificar la localidad de los departamentos de manera que queden con la inicial en mayúscula y el resto de letras con minúsculas.

La instrucción para resolver la solicitud puede ser esta:

```
UPDATE departamento
```

```
SET
loc=concat(UPPER(LEFT(loc,1)),RIGHT(LOWER(loc),LENGTH(loc)-
1));
```

Ejemplo 2 de sentencia UPDATE

En el esquema empresa, se quiere actualizar el importe total real del pedido 1000 a partir de los importes de las diferentes líneas de detalle que forman el pedido.

```
UPDATE pedido c
SET total = (SELECT SUM(importe) FROM detalle
     WHERE pedido_numero=c.pedido_numero)
WHERE pedido_numero=1000;
```

Ahora podemos comprobar la corrección de la información que hay en la base de datos sobre el pedido 1000:

```
SQL> SELECT * FROM detalle WHERE pedido_numero=1000;
+---------------+----------------+-----------------+--------------+----------+---------+
| PEDIDO_NUMERO | DETALLE_NUMERO | PRODUCTO_NUMERO | PRECIO_VENTA | CANTIDAD | IMPORTE |
+---------------+----------------+-----------------+--------------+----------+---------+
|          1000 |              1 |          100860 |        35.00 |       10 |  350.00 |
|          1000 |              2 |          200376 |         2.40 |     1000 | 2400.00 |
|          1000 |              3 |          102130 |         3.40 |      500 | 1700.00 |
+---------------+----------------+-----------------+--------------+----------+---------+
SQL> SELECT * FROM pedido WHERE pedido_numero=1000;

+---------------+--------------+-------------+----------------+-------------+---------+
| PEDIDO_NUMERO | PEDIDO_FECHA | PEDIDO_TIPO | CLIENTE_CODIGO | FECHA_ENVIO | TOTAL   |
+---------------+--------------+-------------+----------------+-------------+---------+
|          1000 | 2000-09-01   | NULL        |            109 | NULL        | 4450.00 |
+---------------+--------------+-------------+----------------+-------------+---------+
1 row in set (0.001 sec)
```

4. SENTENCIA DELETE

La sentencia es la instrucción proporcionada por el lenguaje SQL para borrar filas existentes que hay en las tablas.

Su sintaxis es esta:

```
DELETE FROM <nom_tabla>
[WHERE <condición>];
```

La cláusula optativa selecciona las filas a eliminar. En su caso, se eliminan todas las filas de la Tabla.

Ejemplo de sentencia DELETE

En el esquema empresa, se quiere eliminar el pedido 1000.

La instrucción parece que podría ser esta:

```
DELETE FROM pedido WHERE pedido_numero=1000;
```

Al ejecutar esta sentencia, sin embargo, nos encontramos con un error que nos indica que no se puede eliminar una fila padre (a parent row).

El motivo es que la columna *pedido_numero* de la tabla DETALLE es clave foránea de la Tabla, lo que imposibilita eliminar una cabecera de pedido si hay líneas de detalle correspondientes. Estas se eliminarían de manera automática si hubiera definida la eliminación en cascada, pero no es el caso. Así pues, habrá que hacer lo siguiente:

```
DELETE FROM detalle WHERE pedido_numero=1000;
DELETE FROM pedido WHERE pedido_numero=1000;
```

5. Control de Transacciones y Concurrencias

Una **transacción** es una secuencia de instrucciones SQL que SGBD gestiona como una unidad. Las sentencias **`COMMIT`** y **`ROLLBACK`** permiten indicar un fin de transacción

Transacciones en MySQL

Las transacciones en MySQL sólo tienen sentido bajo el motor de almacenamiento **InnoDB**, que es el único motor **transaccional** de MySQL. Recuerde que los demás sistemas de almacenamiento son no transaccionales y, por tanto, cada instrucción que se ejecuta es independiente y funciona, siempre, de forma **autoconmitiva**.

Ejemplo de transacción: operación en el cajero automático

Una transacción típica es una operación en un cajero automático, por ejemplo: si vamos a un cajero automático a sacar dinero de una cuenta bancaria esperamos que si la operación termina bien (y obtenemos el dinero extraído) se refleje esta operación en la cuenta, y, en cambio, si ha habido algún error y el sistema no ha podido darnos el dinero, esperamos que no se refleje esta extracción en nuestra cuenta bancaria. La operación, pues, debe considerarse como una unidad y acabe bien (**commit**) , pero que si acaba mal (**rollback**) todo quede como estaba inicialmente antes de empezar.

Una transacción habitualmente comienza en la primera sentencia SQL que se produce después de establecer conexión en la base de datos, después de una sentencia **COMMIT** o después de una sentencia **ROLLBACK**.

Una transacción finaliza con la sentencia **COMMIT**, con la sentencia **ROLLBACK** o con la **desconexión** (intencionada o no) de la base de datos.

Los cambios realizados en la base de datos en el transcurso de una transacción sólo son visibles para el usuario que los ejecuta. Al ejecutar una **COMMIT**, los cambios realizados en la base de datos pasan a ser permanentes y, por tanto, visibles para todos los usuarios.

Si una transacción finaliza con **ROLLBACK**, se deshacen todos los cambios realizados en la base de datos por las sentencias de la transacción.

Recordemos que MySQL tiene el **autocommit definido por defecto**, por lo que se efectúa un **COMMIT** automático después de cada sentencia SQL de manipulación de datos. Para desactivarlo es necesario ejecutar:

```
SET autocommit = 0 ;
```

Para volver a activar el sistema de autocommit:

```
SET autocommit = 1 ;
```

Hay que tener en cuenta que una transacción sólo tiene sentido si no está definido el **autocommit**.

El funcionamiento de transacciones no es el mismo en todos los SGBD y, por tanto, habrá que averiguar el tipo de gestión que proporciona antes de querer trabajar.

5.1. Sentencia START TRANSACTION en MySQL

START TRANSACTION define explícitamente el inicio de una transacción. Por tanto, el código que haya entre **START TRANSACTION** y **COMMIT** o **ROLLBACK** formará la transacción.

Iniciar una transacción implica un bloqueo de tablas (**LOCK TABLES**), así como la finalización de la transacción provoca el desbloqueo de las tablas (**UNLOCK TABLES**).

En MySQL **start transaction** es sinónimo de **begin** y, también, de **begin work**. Y la sintaxis para **start transaction** es la siguiente::

```
START TRANSACTION [WITH CONSISTENT SNAPSHOT] | BEGIN [WORK]
```

La opción **WITH CONSISTENT SNAPSHOT** inicia una transacción que permite lecturas consistentes de los datos.

5.2. Sentencias COMMIT y ROLLBACK en MySQL

COMMIT define explícitamente la finalización esperada de una transacción. La sentencia **ROLLBACK** define la finalización errónea de una transacción.

ROLLBACK ignora las acciones que tuvieron lugar dentro de la transacción; muy deseable cuando una instrucción UPDATE / DELETE hace algo no intencionado.

La sintaxis de **commit** y **rollback** en MySQL es la siguiente:

```
COMMIT [ WORK ] [ AND [ NO ] CHAIN | [ NO ] RELEASE ]

ROLLBACK [ WORK ] [ AND [ NO ] CHAIN | [ NO ] RELEASE ]
```

La opción **AND CHAIN** provoca el inicio de una nueva transacción que empezará apenas acabe la actual.

La opción **REALEASE** causará la desconexión de la sesión actual.

Cuando se hace un **rollback** es posible que el sistema procese de forma lenta las operaciones, ya que uno **rollback** es una instrucción lenta. Si se desea visualizar el conjunto de procesos que se ejecutan se puede ejecutar el comando **SHOW PROCESSLIST** y visualizar los procesos que se están ***deshaciendo*** debido al **rollback**.

SGBDR *MySQL* realiza una **COMMIT** implícito antes de ejecutar cualquier sentencia **DDL** (lenguaje de definición de datos) o **DCL** (lenguaje de control de datos), o al ejecutar una desconexión que no haya estado precedida de un error. Por tanto, no tiene sentido incluir este tipo de sentencias dentro de las transacciones.

5.3. Sentencias SAVEPOINT y ROLLBACK TO SAVEPOINT en MySQL

Existe la posibilidad de marcar puntos de control (**savepoints**) en medio de una transacción, de modo que si se efectúa **ROLLBACK** éste pueda ser total (toda la transacción) o hasta uno de los puntos de control de la transacción.

La instrucción **SAVEPOINT** permite crear puntos de control. Su sintaxis es ésta:

```
SAVEPOINT < nombre_punto_control > ;
```

La sentencia **ROLLBACK** para deshacer los cambios hasta un determinado punto de control tiene esta sintaxis:

```
ROLLBACK [ WORK ] TO [ SAVEPOINT ] < nombre_punto_control > ;
```

Si en una transacción se crea un punto de control con el mismo nombre que un punto de control que ya existe, éste queda sustituido por el nuevo.

Si se quiere eliminar el punto de control sin ejecutar un **commit** ni un **rollback** podemos ejecutar la siguiente instrucción:

```
RELEASE SAVEPOINT < nombre_punto_control >
```

Ejemplo de utilización de puntos de control

Consideramos la siguiente situación:

SQL > instrucción_A;

SQL > SAVEPOINT PB;

SQL > instrucción_B;

SQL > SAVEPOINT PC;

SQL > instrucción C;

SQL > instrucción_consulta_1;

SQL > ROLLBACK TO PC;

SQL > instrucción_consulta_2;

SQL > ROLLBACK ; o commit;

La instrucción de consulta 1 ve los cambios efectuados por las instrucciones A, B y C, pero lo ROLLBACK TO PC deshace los cambios producidos desde el punto de control PC, por lo que la instrucción de consulta 2 sólo ve los cambios efectuados por las instrucciones A y B (los cambios por C han desaparecido), y el último ROLLBACK deshace todos los cambios efectuados por A y B, mientras que el último COMMIT los dejaría como permanentes.

5.4. Sentencias LOCK TABLES y UNLOCK TABLES

Concurrencia

La concurrencia en la ejecución de procesos implica la ejecución simultánea de diversas acciones, lo que habitualmente tiene como consecuencia el acceso simultáneo a unos datos comunes, que habrá que tener en cuenta a la hora de diseñar los procesos individuales a fin de evitar inconsistencia en los datos.

Con el fin de prevenir la modificación de ciertas tablas y vistas en algunos momentos, cuando se requiere acceso exclusivo a las mismas, en sesiones paralelas (o concurrentes), es posible bloquear el acceso a las tablas.

El bloqueo de tablas (**LOCK TABLES**) protege contra accesos inapropiados de lecturas o escrituras de otras sesiones.

La sintaxis para bloquear algunas tablas o vistas, e impedir que otros accesos puedan cambiar simultáneamente los datos, es la siguiente:

```
LOCK TABLES < nombre_tabla1 > [ [ AS ] < alias1 > ] READ | WRITE
 [ , < nombre_tabla1 > [ [ AS ] < alias1 > ] READ | WRITE ] ...
```

La opción **read** permite leer sobre la tabla, pero no escribir en ella. La opción **write** permite que la sesión que ejecuta el bloqueo pueda escribir sobre la tabla, pero el resto de sesiones sólo la puedan leer, hasta que termine el bloqueo.

En ocasiones, los bloqueos se utilizan para simular transacciones (en motores de almacenamiento que no sean transaccionales, por ejemplo) o bien para conseguir acceso más rápido a la hora de actualizar las tablas.

Cuando se ejecuta **LOCK TABLES** se hace un **COMMIT** implícito, por tanto, si había alguna transacción abierta, ésta termina. Si termina la conexión (normal o anormalmente) antes de desbloquear las tablas, automáticamente se desbloquean las tablas.

Es posible, también, en MySQL bloquear **todas las tablas de todas las bases de datos** de SGBD, para hacer, por ejemplo, copias de seguridad. La sentencia que lo permite es **FLUSH TABLES WITH READ LOCK**.

Para desbloquear las tablas (todas las que estuvieran bloqueadas) es necesario ejecutar la sentencia **UNLOCK TABLES**.

5.4.1. Funcionamiento de los bloqueos

Cuando se crea un bloqueo para acceder a una tabla, dentro de esta zona de bloqueo no se puede acceder a otras tablas (a excepción de las tablas del diccionario de SGBD -**information_schema**-) hasta que no finalice el bloqueo. Por ejemplo:

```
mysql > LOCK TABLES t1 READ ;
mysql > SELECT COUNT ( * ) FROM t1;
+ ----------+
| COUNT ( * ) |
+ ----------+
```

```
|       3 |
+ ----------+
mysql > SELECT COUNT ( * ) FROM t2;
ERROR 1100 ( HY000 ) : TABLE 't2' was NOT locked WITH LOCK TABLES
```

No se puede acceder más de una vez a la tabla bloqueada. Si se necesita acceder dos veces a la misma tabla, es necesario definir un sobrenombre para el segundo acceso a la hora de realizar el bloqueo.

```
mysql > LOCK TABLE t WRITE , t AS t1 READ ;
mysql > INSERT INTO t SELECT * FROM t;
ERROR 1100 : TABLE 't' was NOT locked WITH LOCK TABLES
mysql > INSERT INTO t SELECT * FROM t AS t1;
```

Si se bloquea una tabla especificando un sobrenombre, debe hacerse referencia con este sobrenombre. Provoca un error acceder directamente con su nombre:

```
mysql > LOCK TABLE t AS myalias READ ;
mysql > SELECT * FROM t;
ERROR 1100 : TABLE 't' was NOT locked WITH LOCK TABLES
mysql > SELECT * FROM t AS myalias;
```

Si se quiere acceder a una tabla (bloqueada) con un sobrenombre, es necesario definir el sobrenombre en el momento de establecer el bloqueo:

```
mysql > LOCK TABLE t READ ;
mysql > SELECT * FROM t AS myalias;
ERROR 1100 : TABLE 'myalias' was NOT locked WITH LOCK TABLES
```

5.5. Sentencia SET TRANSACTION

MySQL permite configurar el tipo de transacción con la sentencia **SET TRANSACTION**.

La sintaxis para configurar las transacciones es:

```
SET [ GLOBAL | SESSION ] TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }
```

Esta sentencia permite definir determinados parámetros para la transacción en curso o para la siguiente transacción que se abrirá. Las características que se definen pueden afectar globalmente (**GLOBAL**) o afectar a la sesión en curso (**SESSION**).

- **READ UNCOMMITTED**: se permite acceder a los datos de las tablas, aunque no se haya realizado un **commit**. Por tanto, es posible acceder a datos no consistentes (**dirty read**).
- **READ COMMITTED**: sólo se permite acceder a datos que se hayan aceptado (**commit)** .
- **REPEATABLE READ** (opción por defecto): permite acceder a los datos de manera consistente dentro de las transacciones, de modo que todas las lecturas de los datos, dentro de una transacción de tipos **REPEATABLE READ**, permitirán obtener los datos como al inicio de la transacción, aunque ya hubieran cambiado.
- **SERIALIZABLE:** permite acceder a los datos de forma consistente en cualquier lectura de los datos, aunque no nos encontramos dentro de una transacción.