Manipulación de Datos

Tabla de Contenidos

1.- Introducción

En anteriores unidades se ha repasado ampliamente las maneras de acceder a la información contenida en la base de datos y extraer información útil para consultas o para aplicaciones.

Sin embargo, dicha información no está sólo para ser consultada, ya que, para que existan datos es necesario que primero se hayan añadido a la base de datos. Asimismo, la información de una base de datos debe poder ser modificada, tanto para corregir errores como por cambios en los datos, que deben quedar reflejados. Por último, a veces es necesario eliminar información que ya no se necesita o no está actualizada.

En esta unidad se verán los mecanismos que se emplean en las bases de datos relacionales para realizar estas operaciones. Además se hablará de la problemática del acceso concurrente a los datos de la base de datos y de los mecanismos ofrecidos para que el acceso concurrente se realice con las garantías necesarias para evitar pérdidas de información o la aparición de inconsistencias.

2.- Añadir filas a una tabla

Para añadir filas a una tabla se utiliza la sentencia `INSERT`. Hay que tener en cuenta que la inserción se puede realizar *sobre una sola tabla*. Si se desean insertar filas en tablas distintas hay que utilizar una sentencia por tabla.

Existen dos formas de funcionamiento de `INSERT`. En una de ellas se especifica en la sentencia los datos a insertar en la tabla y en la otra se obtienen datos desde una consulta y se insertan como filas en una tabla. En las siguientes secciones se detallan estas dos formas.

Es importante tener en cuenta que la inserción de una nueva fila puede fallar aunque los datos que contenga sean perfectamente válidos. Esto puede ser debido a que la nueva fila viola alguna restricción de integridad establecida sobre la tabla en la que se inserta. Más específicamente:

- No se pueden introducir valores duplicados en una clave primaria. Si ya existe un registro con el mismo valor de la clave primaria, la nueva fila no se podrá insertar.
- No se pueden introducir valores duplicados en una columna o columnas con la restricción `UNIQUE`, por la misma razón que la anterior.
- Si la tabla tiene claves ajenas, el sistema comprueba que los valores de dichas claves ajenas en la fila a insertar se corresponden con valores existentes en la tabla o tablas referenciadas. Si no se encuentra correspondencia, la fila no puede ser insertada.

2.1.- Indicando los datos directamente

La forma de INSERT para insertar una fila proporcionando los datos directamente es:

```
INSERT INTO tabla (campo1, campo2, ..., campoN)
VALUES (valor1, valor2, ...., valorN);
```

donde,

- **tabla**

  Es la tabla en la que se va a realizar la inserción

- **campo1, campo2, ..., campoN**

  Nombres de los campos en los que se va a proporcionar un valor para la fila a insertar.

- **valor1, valor2, ..., valorN**

  Valores que se proporcionan para los campos de nombre campo1, campo2,…,campoN, respectivamente.

Adicionalmente, hay que tener en cuenta las siguientes cuestiones sobre el funcionamiento de INSERT:

- Si en la lista de nombres de campos no aparecen todos los campos de la tabla, en aquellos que no aparezcan se insertará el valor por defecto para el campo correspondiente.

  El valor por defecto podrá ser:

  - El indicado por el modificador DEFAULT del campo al crear la tabla.
  - Si no se indicó valor por defecto y el campo no permite NULL, se insertará el valor por defecto para el tipo del campo, determinado por el SGBD.
  - Si no se indicó valor por defecto y el campo permite NULL, se insertará el valor NULL.

- Los valores para los distintos campos se pueden proporcionar utilizando constantes o expresiones que utilicen funciones u operadores. El sistema intentará convertir el valor proporcionado o calculado al tipo de datos de la columna en que se va a almacenar. Si no puede realizar dicha conversión por la razón que sea, la sentencia producirá un error.

- Si, para un campo que aparece en la lista de nombres se quiere dar el valor por defecto de dicho campo, de forma explícita, se puede utilizar el valor especial DEFAULT para indicar que se desea almacenar el valor por defecto de ese campo.

- Si para un campo que aparece en la lista de nombres se quiere dar el valor NULL, bastará con indicarlo utilizando NULL como valor.

EJEMPLO

Supongamos que tenemos una tabla con la siguiente definición:

```
CREATE TABLE empleado (
    dni CHAR(9) NOT NULL,
    nombre VARCHAR(200) NOT NULL,
    sueldo NUMERIC(6,2) NOT NULL,
    complementos NUMERIC(6,2) DEFAULT 0 NOT NULL,
    fecha_nacimiento DATE NOT NULL,
    telefono CHAR(9) NULL,
    CONSTRAINT PRIMARY KEY (dni)
);
```

y con los datos:

empleado					
dni	**nombre**	**sueldo**	**complementos**	**fecha_nacimiento**	**telefono**
11111111A	José López	1232,12	576,88	1987-03-21	600123123
22222222B	Ana Sánchez	1656,87	123,77	1972-05-12	NULL
33333333C	Julia Ramírez	2333,76	777,22	1982-08-24	600333222

Para insertar al nuevo empleado, `Jaime Reyes`, con `DNI 44444444D`, `sueldo 987,67 euros`, `complementos 127,23 euros`, `fecha de nacimiento 23/07/1997` y con `teléfono 666888888`, el comando a ejecutar sería:

```
INSERT INTO empleado
(dni, nombre, sueldo, complementos, fecha_nacimiento, telefono)
VALUES
('44444444D', 'Jaime Reyes', 987.67, 127.23, '1997-07-23', '666888888');
```

Si se hubiera dado el caso de que `Jaime Reyes`, inicialmente no va a cobrar complementos, la inserción podría haberse realizado:

```
INSERT INTO empleado
(dni, nombre, sueldo, complementos, fecha_nacimiento, telefono)
VALUES
('44444444D', 'Jaime Reyes', 987.67, 0, '1997-07-23', '666888888');
```

O, dado, que el valor por defecto de complementos es 0, se podría haber hecho de la manera alternativa:

```
INSERT INTO empleado (dni, nombre, sueldo, fecha_nacimiento, telefono)
VALUES ('44444444D', 'Jaime Reyes', 987.67, '1997-07-23', '666888888');
```

Obteniéndose en ambos casos la misma inserción.

Si el caso hubiera sido que `Jaime Reyes` no dispusiera de teléfono, la inserción debería haberse hecho:

```
INSERT INTO empleado
(dni, nombre, sueldo, complementos, fecha_nacimiento, telefono)
VALUES
('44444444D', 'Jaime Reyes', 987.67, 127.23, '1997-07-23', NULL);
```

O bien

```
INSERT INTO empleado
(dni, nombre, sueldo, complementos, fecha_nacimiento)
VALUES
('44444444D', 'Jaime Reyes', 987.67, 127.23, '1997-07-23');
```

2.2.- Utilizando subconsultas

A veces se quiere hacer permanente información que se obtiene mediante consultas. Para ello debería haber un mecanismo que permita almacenar datos provenientes de una o más tabla de la base de datos en otra. Este mecanismo es la segunda variación de la sentencia `INSERT`. En esta variación, la sintaxis es:

```
INSERT INTO tabla (campo1, campo2, ..., campoN) sentencia_select;
```

donde,

- **`tabla`**
  Es la tabla en la que se va a realizar la inserción.
- **`campo1, campo2, ..., campoN`**
  Son los campos a los que se le va a proporcionar un valor.
- **`sentencia_select`**
  Es una sentencia `SELECT`, como las ya descritas en temas anteriores que es la que va a proporcionar los valores para insertar en la tabla.

Funciona como una sentencia `INSERT` en la que se indican los datos directamente pero utilizando una sentencia `SELECT` para obtener los mismos.

La sentencia SELECT tiene unas restricciones:

- El número de columnas del resultado debe coincidir con el número de campos de la lista de campos de la sentencia INSERT.
- Los valores devueltos por la consulta para cada columna deben de poder convertirse al tipo de datos de la columna de destino. Si no se puede realizar una conversión, se producirá un error.

EJEMPLO

Supongamos que se ha decidido, por parte de la empresa, el eliminar los complementos y utilizar únicamente un salario. Para ello se he creado una nueva tabla para almacenar los datos de los empleados con la siguiente estructura:

```
CREATE TABLE empleado_nuevo (
    dni CHAR(9) NOT NULL,
    nombre VARCHAR(200) NOT NULL,
    salario NUMERIC(6,2) NOT NULL,
    fecha_nacimiento DATE NOT NULL,
    telefono CHAR(9) NULL,
    CONSTRAINT PRIMARY KEY (dni)
);
```

Como se puede comprobar es muy similar a la actual pero desaparece el campo `complementos` y el campo `sueldo` se ha renombrado a `salario`.

Si se quieren traspasar los datos de la tabla antigua a la nueva se podría hacer con la siguiente sentencia:

```
INSERT INTO empleado_nuevo
(dni, nombre, salario, fecha_nacimiento, telefono)
SELECT dni, nombre, sueldo + complementos, fecha_nacimiento, telefono
FROM empleado;
```

2.3.- Copiar Tablas

Se puede realizar una copia de una tabla utilizando la sintaxis:

```
SELECT INTO tabla_nueva FROM tabla_antigua;
```

donde,

- **`tabla_nueva`**

  Es la nueva tabla a crear. No debe existir ya una tabla con ese nombre o se producirá un error.
- **`tabla_antigua`**

  Es la tabla a copiar.

Esta sentencia creará una nueva tabla igual que la original (estructura y datos).

NOTA

Esta sintaxis no funciona en MariaDB / MySQL. Se puede hacer lo mismo utilizando la sintaxis:

```
CREATE TABLE tabla_nueva SELECT * FROM tabla_antigua;
```

3.- Modificar filas de una tabla

La modificación de los datos de una tabla se realiza a través de la sentencia UPDATE, con la siguiente sintaxis:

```
UPDATE tabla
SET campo1 = expresion1, campo2 = expresion2, ..., campoN = expresionN
[WHERE condicion];
```

donde,

- **tabla**

  Es la tabla en la que está la fila o filas que se quieren actualizar (se pueden actualizar más de una fila con una sola sentencia UPDATE).
- **campoX**

  Es un campo cuyo valor se quiere modificar.
- **expresionX**

  Es el nuevo valor que se va a asignar al campo.
- **condicion**

  Es una condición sobre los campos de cada fila de la tabla tabla. Aquellas filas que cumplan la condición *antes de ser actualizadas* se actualizaran. Aquellas que no la cumplan permanecerán sin cambios.

La sentencia funciona de la siguiente manera:

- Si se proporciona una cláusula WHERE se procesa toda la tabla y se comprueba la condición para cada fila de la tabla. Aquellas filas en las que la condición proporcione un valor VERDADERO serán actualizadas. Las filas en las que la condición proporcione un valor FALSO **no** serán actualizadas.
- Si no se proporciona una cláusula WHERE **todas las filas de la tabla serán actualizadas**. Es por esto por lo que hay que tener especial cuidado a la hora de utilizar UPDATE sin una cláusula WHERE.
- Para crear la condición de la cláusula WHERE se pueden utilizar la misma sintaxis que para las cláusula WHERE de la sentencia SELECT, incluidas subconsultas. Sin embargo, hay que tener en cuenta que para evaluar la condición se utilizan los valores que están actualmente almacenados en la tabla, o sea, ***los valores existentes antes de realizar la actualización***.

- Una vez seleccionadas las filas sobre las que se va a realizar la actualización, se recorren una a una y se van aplicando las actualizaciones indicadas por las parejas `campo = valor` indicadas en la clausula `SET`. En cada fila se cambia el valor del campo con `nombre campo` al nuevo valor indicado en la `expresión valor`.
- La expresión que da el nuevo valor puede ser un valor constante (a todas las filas se le va a asignar el mismo valor) o puede ser una expresión que utilice los valores actuales de la fila a actualizar para calcular los nuevos valores. Esta expresión puede utilizar operadores, funciones y subconsultas que devuelvan un único valor (una única columna y una única fila). En la subconsulta se pueden utilizar los valores de los campos de la fila que se está actualizando.

Al realizar actualizaciones, hay que tener especial cuidado con la modificación de columnas que participen en una regla de integridad referencial (`FOREIGN KEY`) o de identidad (`PRIMARY KEY / UNIQUE`) ya que pueden producirse situaciones en las que la actualización provocaría la violación de alguna de estas reglas. En ese caso la sentencia fallaría con un error.

EJEMPLO

Para subir el `sueldo` un `5%` a todos los empleados cuyo `salario neto` (sueldo mas `complementos`) es menor de `1500 euros`:

```
UPDATE empleado SET sueldo = sueldo * 1.05
WHERE (sueldo + complementos) < 1500;
```

Para almacenar el sueldo completo en sueldo y poner el complemento a cero, se utilizaría:

```
UPDATE empleado SET sueldo = sueldo + complementos, complementos = 0;
```

Como se puede ver, no se utiliza `WHERE` por lo que la actualización se realizará para *todos* los empleados.

4.- Eliminar filas de una tabla

La eliminación de filas se realiza mediante la sentencia `DELETE`, con la siguiente sintaxis:

```
DELETE FROM tabla [WHERE condicion];
```

donde,

- **`tabla`**
  Es la tabla en la que se van a eliminar filas.
- **`condicion`**
  Es una condición que se evalúa por cada fila de la tabla.

El funcionamiento es muy sencillo: Se recorre toda la tabla `tabla` y por cada fila se evalúa `condicion`. Si el resultado es `VERDADERO`, la fila se elimina de la tabla. Si el resultado es `FALSO`, la fila permanece. La clausula `WHERE` es opcional y si no se indica significa que se eliminarán **TODAS** las filas.

Como en UPDATE, en la condición se pueden emplear cualquier operador o función que es válido en la clausula WHERE de una sentencia SELECT, incluidas las subconsultas.

EJEMPLO

Para eliminar todos los empleados cuyo salario sea superior a los 3000 euros se emplearía la sentencia:

```
DELETE FROM empleado WHERE (sueldo + complementos) > 3000;
```

O para dejar vacía la tabla empleado se podría utilizar:

```
DELETE FROM empleado;
```

5.- Transacciones

Hasta ahora todas las operaciones sobre las bases de datos se han realizado por parte de un sólo usuario. Sin embargo, los SGBDs pierden utilidad si la información sólo puede ser manipulada por un usuario. Para funcionar a plena potencia se necesita permitir el acceso simultáneo de varios usuarios a los mismos datos (el número de usuarios variará según el SGBD).

Este acceso simultáneo o *concurrente*, lamentablemente presenta algunos problemas que deben ser evaluados y corregidos. Entre los problemas que presenta el acceso concurrente a los datos tenemos:

- **Pérdida de actualización**

  Imaginemos el siguiente escenario. Dos personas, A y B van a sacar dinero de la misma cuenta utilizando dos cajeros automáticos distintos. En la cuenta hay actualmente 100 euros y el usuario A quiere sacar 50 euros y el B 40. El usuario A lee el saldo y el sistema dice 100 euros. El usuario B también lee el saldo y es 100. El usuario B saca el dinero y se actualiza la cuenta que ahora tendrá 100 - 40 = 60 euros. A continuación el usuario A, que se ha retrasado un poco, saca también su dinero y actualiza la cuenta, que ahora tendrá 100 - 50 = 50 euros. Como se puede ver se ha producido un error porque la cuenta debería tener tras las dos retiradas un total de 100 - 50 - 40 = 10 euros pero termina teniendo 50 euros en su lugar.

- **Lecturas de valores incorrectos**

  Imaginemos el mismo escenario anterior pero ahora la secuencia de pasos es distinta. El usuario A lee el saldo (100) y como hay suficiente, lo actualiza al nuevo saldo (100 - 50 = 50). El usuario B lee el saldo (50) y como tiene suficiente, lo actualiza al nuevo saldo (50 - 40 = 10). A continuación el usuario A decide cancelar la operación y deja el saldo de la cuenta como estaba (100). El usuario B continúa y obtiene su efectivo. Ahora la cuenta tiene un saldo de 100 pero realmente se han retirado 40 euros y debería tener un saldo de 100 - 40 = 60 euros.

- **El problema del resumen incorrecto**

  Imaginemos un escenario distinto. El usuario A está calculando el total de las facturas de un mes determinado, recorriendo las facturas una a una y sumando su importe a un total. Al mismo tiempo el usuario B añade una factura nueva a ese mes que se había traspapelado y el usuario C modifica el importe de otra factura de ese mismo mes. Según el orden en que se hagan estas operaciones, el resumen dará un valor distinto, pero probablemente en ninguno de los casos sea correcto con el valor que debería tener si las tres operaciones se hubieran hecho una después de la otra.

Estos problemas pueden provocar inconsistencias cuando se accede concurrentemente a la misma información. Por lo tanto el SGBD debe poder asegurar de alguna forma que estos problemas no ocurren.

El mecanismo por el que se materializa la solución se denomina *transacción*. Una transacción es un grupo de operaciones sobre la base de datos que cumple las siguientes reglas:

- **Atomicidad**

  O bien el efecto de todas las operaciones realizadas dentro de la transacción se almacenan de forma permanente o bien ninguna lo hace. Dicho en otras palabras, o bien todas las operaciones se realizan o no se realiza ninguna (porque se produzca un error o la transacción se aborte). Todas las operaciones de la transacción funcionan como si fueran una sola.

- **Consistencia**

  Cada transacción se encuentra la base de datos en un estado consistente antes de comenzar y la debe dejar en un estado consistente al terminar, ya se realicen sus operaciones o no.

- **Aislamiento**

  Las transacciones no pueden interferir entre ellas, es decir, los efectos de una transacción que aún no ha terminado no deben ser visibles para otras transacciones. Este es el principal objetivo del control de acceso concurrente.

- **Durabilidad**

  Cuando una transacción se ha confirmado con éxito, sus efectos deben permanecer en la base de datos aunque ocurran errores en la base de datos o el equipo se apague.

Estas cuatro reglas son las que se conocen como ACID, por las siglas en inglés de sus nombres (Atomicity, Consistency, Isolation, Durability).

Las transacciones utilizan distintas técnicas para obtener el cumplimiento de estas reglas. De ellas, las más utilizadas son:

- **Bloqueo (Locking)**

  Cada dato posee un bloqueo (lock). Si un usuario quiere acceder a un dato durante una transacción debe obtener el bloqueo antes. Si lo obtiene, ningún otro usuario podrá escribir a ese dato hasta que el bloqueo se libere y el nuevo usuario lo adquiera. El problema de esta técnica es que puede provocar problemas de esperas mientras se liberan los bloqueos y también problemas de interbloqueos (situaciones en que dos usuarios tienen un bloqueo cada uno y están esperando

por el bloqueo que tiene el otro). Aún con todos sus problemas es uno de los sistemas más utilizados.

- **Comprobación de grafos de serialización**

  El sistema reordena las transacciones en curso de forma que se ejecuten unas antes que otras. Si se bloquean entre ellas aborta una y deja seguir a la otra.

- **Multiversión**

  Se mantienen varias versiones de cada dato con un timestamp que indica el momento de la modificación. Una transacción sólo accede a los datos en su versión anterior al inicio de su ejecución. Ocupa más espacio pero previene esperas y es muy rápido.

5.1.- Ejecutar una secuencia de sentencias en una transacción

Ejecutar sentencias en una transacción es muy sencillo.

Una transacción se comienza utilizando la sentencia:

```
BEGIN TRANSACTION;
```

La palabra `TRANSACTION` es opcional.

Esta sentencia indica que comienza una transacción. A partir de ese punto, y hasta que se termine la transacción, todas las sentencias forman parte de la transacción y se ejecutarán de forma atómica.

Una transacción se finaliza utilizando las sentencias:

```
COMMIT;
```

o

```
ROLLBACK;
```

`COMMIT` confirma la transacción y finaliza la misma incorporando los cambios realizados a la base de datos.

`ROLLBACK` aborta la transacción y finaliza la misma descartando los cambios realizados a la base de daros, que permanecerá como estaba al iniciar la transacción.

EJEMPLO

Supongamos que tenemos una tabla `cuenta` en las que se almacenan las cuentas corrientes de los clientes. Entre otros datos cada cuenta tiene un `id` único de cliente y el `saldo actual` de la cuenta. Si se deseara retirar dinero (100 euros) de la cuenta número `222444555444`, una posible secuencia de sentencias a ejecutar por parte de una aplicación sería:

```
BEGIN TRANSACTION;
SELECT saldo FROM cuenta WHERE idCliente =  222444555444;
```

```
.... El programa comprueba que el saldo es suficiente, esto es, que es igual o mayor a lo que se desea retirar (100 euros)....
... Supongamos que el saldo es suficiente (525 euros)....
UPDATE cuenta SET saldo = 425 WHERE idCliente =  222444555444;
COMMIT;
```

Una posible secuencia si el saldo no fuera suficiente...

```
BEGIN TRANSACTION;
SELECT saldo FROM cuenta WHERE idCliente =  222444555444;
.... El programa comprueba que el saldo es suficiente, esto es, que es igual o mayor a lo que se desea retirar (100 euros)....
... Supongamos que el saldo NO es suficiente (75 euros)....
ROLLBACK;
```

6.- Vistas actualizables

En la mayoría de los casos, las vistas son mecanismos de sólo lectura, es decir, pueden utilizarse para consultar datos pero no se pueden insertar, modificar ni eliminar filas.

En algunos casos, se puede permitir que una vista pueda ser modificable, actualizándose la tabla subyacente pero para que esto pueda ser posible debe existir una relación uno a uno entre las filas de la vista y las filas de la(s) tabla(s) subyacente(s). Si esto no se cumple, la vista es no actualizable. Esto ocurre especialmente cuando en la sentencia `SELECT` se utilizan:

- Funciones de agregación (`SUM, AVG`,..)
- `DISTINCT`
- `GROUP BY`
- `HAVING`
- `UNION`
- Subconsultas en la lista de columnas.
- Algunos tipos de composiciones (`JOINS`)

También pueden ocurrir problemas si una vista no contiene todos los campos de la tabla subyacente y alguno de estos campos no puede ser `NULL` ni tiene un valor por defecto. En este caso se producirá un error al intentar utilizar la sentencia `INSERT` sobre la vista.

En general, se podrán actualizar con mayor o menor seguridad vistas sobre una sola tabla (sin subconsultas ni composiciones) y que tengan todos las columnas de la tabla original que no puedan valer `NULL` o no tengan valor por defecto.

EJEMPLO

Si se tiene la definición de la vista `empleado_v`, y suponiendo que los campos no contenidos en la vista pueden valer `NULL`, se podría realizar la siguiente inserción:

```
INSERT INTO empleado_v (codigo, nombre, oficina)
VALUES (111, 'Lola Sanchez', 22);
```

que produciría el siguiente resultado sobre la tabla subyacente:

empleado								
codigo	nombre	edad	oficina	titulo	fecha_contrato	codigo_jefe	cuota_ventas	ventas_obtenidas
101	Antonio Viguer	45	12	Representante	20/10/06	104	300000	305000
102	Álvaro Jaumes	48	21	Representante	10/12/06	108	350000	474000
103	Juan Rovira	29	12	Representante	01/03/07	104	275000	286000
104	José González	33	12	Dir Ventas	19/05/07	106	200000	143000
105	Vicente Pantall	37	13	Representante	12/01/08	104	350000	368000
106	Luis Antonio Rodríguez	52	11	Dir General	14/06/08	NULL	275000	299000
107	Jorge Gutiérrez	49	22	Representante	14/11/08	108	300000	186000
108	Ana Bustamante	62	21	Dir Ventas	12/10/09	106	350000	361000
109	María Sunta	31	11	Representante	12/10/16	106	300000	392000
110	Juan Víctor	41	22	Representante	13/01/10	104	500000	760000
111	Lolsa Sánchez	NULL	22	NULL	NULL	NULL	NULL	NULL

Un problema que se puede presentar al utilizar vistas es que pueden ocasionar problemas de seguridad. Tomemos el siguiente caso:

Supongamos la tabla `empleado` utilizada anteriormente. Por razones que no vienen al caso se decide que el usuario `gestor` debe tener acceso a los datos de los empleados de tipo `Representante` a fin de gestionar su trabajo, pero no debe tener acceso a otro tipo de empleados.

Para llevar esto a cabo se crea una vista, llamada `empleado_gestor` con la siguiente sentencia:

```
CREATE OR REPLACE VIEW empleado_gestor
AS SELECT * FROM empleado WHERE TITULO = 'Representante';
```

Esto crearía una vista, llamada `empleado_gestor` que sólo contiene los representantes. Como se puede comprobar, esta vista es actualizable ya que abarca todos los campos de una sola tabla y no se utilizan agregados, ni ninguna de las otras funcionalidades que impiden la actualización.

Sin embargo, a la hora de insertar registros en la vista se puede producir un problema de seguridad porque el usuario `gestor` podrá crear un director de ventas simplemente utilizando la sentencia:

```
INSERT INTO empleado_gestor
(codigo, nombre, edad, oficina, titulo, fecha_contrato, codigo_jefe, cuota_ventas, ventas_obtenidas)
VALUES
(112. 'Falso Director' 99, 22, 'Director General', '2017-04-04', NULL, 0, 0);
```

Además, otro efecto lateral es que esta nueva fila no aparecerá en la vista a pesar de haber sido insertada a través de ella. La razón es que la nueva fila no cumple la condición de la sentencia `SELECT` de la vista y, por lo tanto, no aparece en ella, aunque sí se ha insertado en la tabla subyacente (`empleado`).

Para evitar esto se utiliza la clausula `WITH` al final de la sentencia `CREATE OR REPLACE VIEW`, de la forma:

```
CREATE OR REPLACE VIEW nombre_vista AS sentencia_select WITH opcion_with;
```

donde **`opcion_with`** puede ser:

- **`CHECK OPTION`**

  Esta opción obliga a que las modificaciones que se realicen a través de la vista deben cumplir la restricción de la misma, es decir, que la nueva fila o la fila actualizada debe formar parte de la vista después de la inserción ó modificación. En caso de que no se cumpla esta condición, la inserción o modificación producirá un error.

- **`READ ONLY`**

  Esta opción obliga a que la vista sea de sólo lectura y se prohíba la inserción o modificación de filas. Esta opción no está soportada por MySQL.

Apéndice A: Transacciones en MariaDB

MariaDB utiliza las transacciones siguiendo la sintaxis especificada en los apartados anteriores. Sin embargo es conveniente indicar algunas características particulares de MariaDB:

- Por defecto, MariaDB realiza una transacción por cada sentencia que ejecuta. A esto es llamado el modo `autocommit`.
- Al ejecutar una sentencia `BEGIN`, MariaDB suspende temporalmente este modo hasta que la transacción finaliza.
- MariaDB soporta una sintaxis alternativa para iniciar una transacción utilizando la sentencia `START TRANSACTION`. El funcionamiento es idéntico.
- Para que se soporten transacciones, el motor de almacenamiento de la base de datos debe ser `InnoDB`. El resto de motores de almacenamiento soportados por MariaDB no soportan transacciones.
- MariaDB comprueba las reglas de integridad referencial durante una transacción, por lo que estas no se pueden violar temporalmente, como es el comportamiento estándar.