BD
Bases de Datos

UD 6

DML: Inserción, borrado y modificación de datos

UD6 – DML: inserción, borrado y modificación de datos

INDICE

1. Introducción

El **Lenguaje de Manipulación de Datos** (LMD, en inglés Data Manipulation Language, DML) es un lenguaje proporcionado por el sistema de gestión de base de datos que permite a los usuarios de la misma llevar a cabo las tareas de consulta o manipulación de los datos (inserción, borrado, actualización y consultas) basado en el modelo de datos adecuado.

Mediante las consultas de inserción, borrado y modificación de datos podemos gestionar la información de los registros de la base de datos. A continuación, veremos las consultas más habituales y ejemplos de uso.

4.1 INSERT

INSERT inserta nuevas filas en una tabla existente.

El formato INSERT ... VALUES insertas filas basándose en los valores especificados explícitamente.

El formato INSERT ... SELECT inserta filas seleccionadas de otra tabla o tablas.

La sintaxis para insertar un registro es la siguiente:

```
INSERT INTO nombreTabla (col_name1, col_name2, ...)
VALUES (valor_col1, valor_col2, ... );
```

Consejo: Es muy importante que nos aseguremos que "col_name1" va con "valor_col1", si no fuera así, algo inesperado podría suceder. En el mejor de los casos lo sabremos porque la sentencia INSERT devolverá un error cuando la ejecutemos, y en el peor lo sabremos cuando ya sea demasiado tarde…

La sintaxis para insertar varios registros es la siguiente:

```
INSERT INTO nombreTabla (col_name1, colname2, ...)
VALUES (valor1_col1, valor1_col2, ... ),
       (valor2_col1, valor2_col2, ... ),
        ...
       (valorn_col1, valorn_col2, ... );
```

La sintaxis para insertar registros desde una consulta SELECT es la siguiente:

```
INSERT INTO nombreTabla2 (col_name1, col_name2, ...)
   SELECT col_name1, col_name2, ...]
    FROM nombreTabla1
  [WHERE condicion];
```

Consejo: La consulta puede ser todo lo complicada que se requiera, lo único que debe cumplir es que lo que devuelva coincida exactamente con lo que se quiere insertar y en el mismo orden.

Si diera la casualidad de que uno de los campos que queremos insertar fuera el mismo para todos los registros que devuelva la SELECT, podemos indicarlo directamente con el valor como en el siguiente ejemplo:

```
INSERT INTO HISTORICO (codCliente, nombre, apellidos, fechaAlta)
   SELECT codCliente, nombre, apellidos, GetDate()
    FROM CLIENTE;
```

Si hay campos en la tabla que no se especifican en la instrucción INSERT, se le asignará el valor que se haya indicado en su creación por defecto (DEFAULT) y en caso contrario será NULL.

Insertar en tablas con campo PK IDENTITY activo

Imagina que tenemos una tabla PRODUCTO (idProducto, nombre, precio, unidades) y que la PK idProducto se ha definido como INT IDENTITY(1,1).

Si insertamos directamente el idProducto:

```
INSERT INTO PRODUCTO (idProducto, nombre, precio, unidades)
   VALUES  (1, 'Playstation 5', 599.99, 0);
```

SQL Server nos devolverá un error, indicando que esta tabla tiene autoincremento del campo idProducto.

Para hacerlo correctamente deberíamos omitir el campo idProducto, de este modo:

```
INSERT INTO PRODUCTO (nombre, precio, unidades)
   VALUES  ('Playstation 5', 599.99, 0);
```

Automáticamente, SQL Server buscará cuál es el último idProducto insertado y lo utilizará para nuestra sentencia.

NOTA: Si borramos el registro que acabamos de insertar y volvemos a insertar otro, en lugar de obtener el id anterior, obtendrá el siguiente libre, por lo que, si eliminamos registros, se generarán huecos en los id de la tabla.

Si en algún caso justificado (por ejemplo, que tengamos una incidencia y necesitemos indicar nosotros mismos el id a insertar), podremos utilizar la instrucción **SET IDENTITY_INSERT**.

Es un comando que se ejecuta antes del INSERT, pero tiene una limitación importante… sólo puede estar activo en una sola tabla a la vez y únicamente para nosotros, cuando cerremos la conexión a la BD dejará de estar activo.

Para activarlo en una tabla llamada PRODUCTO utilizaremos:

```
SET IDENTITY_INSERT PRODUCTO ON;
```

Para desactivarlo utilizaremos la instrucción contraria:

```
SET IDENTITY_INSERT PRODUCTO OFF;
```

Como se ha comentado, si ejecutamos:

```
SET IDENTITY_INSERT PRODUCTO ON;
GO
SET IDENTITY_INSERT CLIENTE ON;
-- ERROR
```

El Sistema Gestor devolverá un error indicando que sólo puede estar activo en una tabla. Para hacerlo correctamente:

```
SET IDENTITY_INSERT PRODUCTO ON;
GO
SET IDENTITY_INSERT PRODUCTO OFF;
GO
SET IDENTITY_INSERT CLIENTE ON;
-- Aquí ya podemos usarlo
```

Más información en: https://learn.microsoft.com/es-es/sql/t-sql/statements/set-identity-insert-transact-sql?view=sql-server-ver16

4.2 UPDATE

UPDATE actualiza columnas de filas existentes de una tabla con nuevos valores.

La cláusula SET indica las columnas a modificar y los valores que deben tomar.

La cláusula WHERE, si se indica, especifica qué filas deben ser actualizadas. Si no se especifica, **todas las filas de la tabla serán actualizadas**.

La sintaxis para actualizar registros es la siguiente:

```
UPDATE TABLA
    SET campo1 = valor1,
        Campo2 = valor2,
        ...        ...
        CampoN = valorN
  [WHERE condición/es];
```

Los valores actualizables pueden ser expresiones que se apliquen sobre campos o funciones predefinidas de SQL Server como por ejemplo GetDate().

Ejemplo 1 – Actualiza el campo fechaDevolución de un préstamo de un libro realizado en una biblioteca. Esta consulta se ejecutará cuando el socio acuda a la biblioteca a devolver el libro.

```
UPDATE PRESTAMO
    SET fechaDevolucion = GetDate()
 WHERE codPrestamo = 1234;
```

Es importante asegurarse que los valores que indiquemos en la cláusula SET coincidan con el tipo de dato del campo que se desea actualizar (cadena de caracteres, número entero, decimal, fecha, fecha/hora, etc.)

Ejemplo 2 – Actualizar el nombre a una persona

```
UPDATE PERSONA
    SET nombre = 'Juan Martínez'
 WHERE DNI = '11111111A';
```

Ejemplo 3 – Aumentar la edad de una persona por su cumpleaños

```
UPDATE PERSONA
    SET edad = edad + 1
 WHERE DNI = '11111111A';
```

Ejemplo 4 – Aumentar el precio de todos los artículos de la tienda un 15%

```
UPDATE PRODUCTO
  SET precio = precio * 1.15;
```

Ejemplo 5 – Actualizar varios campos de una persona

```
UPDATE PERSONA
      SET nombre = 'Álvaro',
            apellido1 = 'Martínez',
            apellido2 = 'Pérez'
 WHERE DNI = '22222222B';
```

También podemos utilizar la condición del WHERE como la respuesta a una consulta, como por ejemplo:

Ejemplo 6 – Actualiza el campo esCliente a "S" para todos aquellos clientes que hayan realizado algún pedido

```
UPDATE PERSONA
      SET esCliente = 'S'
 WHERE codCliente IN (SELECT codCliente        -- Si el cliente aparece en la tabla PEDIDO
                      FROM PEDIDO);            -- es que ha realizado al menos uno.
```

Aquí toman mucha relevancia las **subconsultas** para poder dinamizar mucho más el alcance de las sentencias UPDATE.

Contenido de ampliación

Hasta ahora, hemos estudiado que es posible utilizar una subconsulta para que actualice los resultados que devuelva. No obstante, existe una técnica más avanzada que permite actualizar los datos de una tabla en función de los datos que devuelva una SELECT.

Vamos a preparar el entorno para demostrar el funcionamiento de este tipo de UPDATE.

```
------------------------
--  DATABASE CREATION
------------------------
CREATE DATABASE TEST_UPDATE
GO

USE TEST_UPDATE
GO
```

```
------------------------
--   TABLE CREATION
------------------------

-- Client's table
CREATE TABLE CLIENT (
       idClient      INT IDENTITY(1,1),
       name          VARCHAR(100),
       surname       VARCHAR(100),
       postalCode    CHAR(5) NULL,
       city          VARCHAR(100) NULL,
       phoneNumber   VARCHAR(12) NULL
       CONSTRAINT PK_CLIENT PRIMARY KEY (idClient)
);

GO

-- Additional data from clients
CREATE TABLE CLIENT_DATA (
       idData               INT IDENTITY(1,1),
       idClient             INT,
       postalCodeData       CHAR(5),
       cityData             VARCHAR(100),
       phoneNumberData      VARCHAR(12)
       CONSTRAINT PK_CLIENTDATA PRIMARY KEY (idData)
);

GO

------------------------
--   DATA INSERT
------------------------

INSERT INTO CLIENT (name, surname)
VALUES ('N1', 'S1'),
       ('N2', 'S2'),
       ('N3', 'S3'),
       ('N4', 'S4'),
       ('N5', 'S5');

GO

INSERT INTO CLIENT_DATA (idClient, postalCodeData, cityData, phoneNumberData)
VALUES (1, '11111', 'City1', '96111111'),
       (2, '22222', 'City2', '96222222'),
       (3, '33333', 'City3', '96333333'),
       (4, '44444', 'City4', '96444444'),
       (5, '55555', 'City5', '96555555');
```

CLIENT

idClient	name	surname	postalCode	city	phoneNumber
1	N1	S1	NULL	NULL	NULL
2	N2	S2	NULL	NULL	NULL
3	N3	S3	NULL	NULL	NULL
4	N4	S4	NULL	NULL	NULL
5	N5	S5	NULL	NULL	NULL

CLIENT_DATA

idData	idClient	postalCodeData	cityData	phoneNumberData
1	1	11111	City1	96111111
2	2	22222	City2	96222222
3	3	33333	City3	96333333
4	4	44444	City4	96444444
5	5	55555	City5	96555555

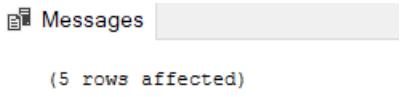
La idea es que el valor de los campos postalCodeData, cityData y phoneNumberData de la tabla CLIENT_DATA se actualicen en los campos respectivos de la tabla CLIENT.

Para ello, podemos hacer uso de la siguiente sentencia:

Ejemplo 7 – Actualizar tabla con el resultado de una SELECT

```
UPDATE cli
    SET cli.postalCode = dat.postalCodeData,
        cli.city = dat.cityData,
        cli.phoneNumber = dat.phoneNumberData
 FROM CLIENT cli INNER JOIN CLIENT_DATA dat ON cli.idClient = dat.idClient;
```

Si ejecutamos la consulta, obtendremos el siguiente resultado:



Revisamos el contenido de las tablas para asegurarnos de que se han actualizado correctamente:

CLIENT

idClient	name	surname	postalCode	city	phoneNumber
1	N1	S1	11111	City1	96111111
2	N2	S2	22222	City2	96222222
3	N3	S3	33333	City3	96333333
4	N4	S4	44444	City4	96444444
5	N5	S5	55555	City5	96555555

CLIENT_DATA

idData	idClient	postalCodeData	cityData	phoneNumberData
1	1	11111	City1	96111111
2	2	22222	City2	96222222
3	3	33333	City3	96333333
4	4	44444	City4	96444444
5	5	55555	City5	96555555

¡Todo correcto! Este tipo de sentencias no son muy frecuentes en el trabajo diario, pero si tenemos datos que pertenezcan a la misma tabla separados en varias, esta sentencia puede ser de mucha utilidad para integrarlos.

4.3 DELETE

DELETE elimina las columnas de una tabla que cumplan la condición dada por la cláusula, y devuelve el número de registros borrados. Sin cláusula WHERE, se eliminarán todos los registros.

La sintaxis para eliminar registros de una tabla es la siguiente:

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

Una sentencia DELETE solo puede eliminar los registros de una sola tabla, si necesitamos eliminar de dos tablas tendremos que diseñar y ejecutar dos sentencias DELETE.

Consejo: Si estamos eliminando tablas relacionadas por claves ajenas (FK), tendremos que eliminar previamente la tabla que tiene la FK y luego la que tiene la PK (si tratamos de hacerlo al revés no nos dejará eliminar la tabla con la PK hasta que no eliminemos todas las referencias que tenga ese campo).

Si necesitamos eliminar todos los registros de una tabla, en lugar de utilizar DELETE FROM sin WHERE es preferible utilizar la siguiente sentencia:

```
TRUNCATE TABLE nombreTabla;
```

El motivo es que TRUNCATE TABLE es más eficiente al no validar la integridad de la referencial cada vez que se borra un registro, por lo que la carga para el SGBD es menor y por lo tanto, esta sentencia es óptima para vaciar tablas (pero esta sentencia NO elimina la tabla, por lo que podremos insertar registros de nuevo). Por supuesto, debemos respetar el orden de vaciado, primero las tablas con las FKs y luego las tablas con las PK.

Ejemplo 1 – Elimina la persona que tenga el DNI 11111111A. DNI es la PK de la tabla, por lo que sólo se eliminará UN registro.

```
DELETE FROM PERSONA
 WHERE DNI = '11111111A';
```

Ejemplo 2 – Elimina las personas que tengan el DNI 11111111A y el DNI 22222222B. DNI es la PK de la tabla

Si escribimos la siguiente sentencia, veremos que NO se actualiza ningún registro… ¿por qué?

```
DELETE FROM PERSONA
 WHERE DNI = '11111111A'
    AND DNI = '22222222B';
```

La sentencia correcta sería utilizando OR en lugar de AND, puesto que NO hay ningún registro que cumpla simultáneamente que tenga el DNI 11111111A y el DNI 22222222B:

```
DELETE FROM PERSONA
 WHERE DNI = '11111111A'
      OR DNI = '22222222B';
```

Si tenemos varias condiciones como en el caso anterior, es preferible el uso del operador IN:

```
DELETE FROM PERSONA
 WHERE DNI IN ('11111111A', '22222222B');
```

Ejemplo 3 – Elimina las personas que no hayan realizado ningún pedido.

```
DELETE FROM PERSONA
 WHERE DNI IN (SELECT DNI
                  FROM PEDIDO);
```

La subconsulta devuelve en la columna DNI las personas que han realizado algún pedido.

Consideremos que devuelve los DNI '33333333C' y '44444444D'.

Sería equivalente a la consulta anterior utilizar, lo único es que en lugar de indicarlo lo obtenemos dinámicamente de una SELECT:

```
DELETE FROM PERSONA
 WHERE DNI IN ('33333333C', '44444444D');
```

Ejemplo 4 – Elimina las personas que no hayan realizado ningún pedido y cuyo nombre empiece por la letra C

```
DELETE FROM PERSONA
 WHERE DNI IN (SELECT DNI
                  FROM PEDIDO)
    AND nombre LIKE 'C%';
```

Podemos definir tantas condiciones como sean necesarias utilizando los operadores AND, OR y NOT y paréntesis cuando se requieran.

Ejemplo 5 – Elimina las personas que están repetidas. Para este ejemplo utilizaremos una tabla CLIENT que tiene como PK un campo llamado idClient con IDENTITY(1,1).

Puede ocurrir que se inserten por error varias personas con los mismos valores excepto el campo idClient, el cual es automático.

Los clientes con id 6 y 7 están repetidos (son los mismos que los clientes 5 y 3, respectivamente)

idClient	name	surname	postalCode	city	phoneNumber
1	N1	S1	11111	City1	96111111
2	N2	S2	22222	City2	96222222
3	N3	S3	33333	City3	96333333
4	N4	S4	44444	City4	96444444
5	N5	S5	55555	City5	96555555
6	N5	S5	NULL	NULL	NULL
7	N3	S3	NULL	NULL	NULL

```
SELECT name, surname, COUNT(*) numRep
  FROM CLIENT
GROUP BY name, surname
HAVING COUNT(*) > 1;
```

name	surname	numRep
N3	S3	2
N5	S5	2

Necesitamos eliminar uno de los dos registros, pero NO los dos. Para ello, podemos utilizar la función de agregado MAX o MIN para obtener el ID que queramos mantener (dependerá de cada caso, habría que estudiar qué ha provocado los duplicados, pero por lo general serán siempre los que hayan llegado los últimos, es decir, los de MAX).

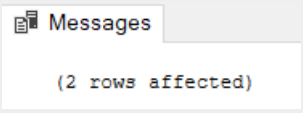
La consulta que obtiene esos IDs repetidos que han llegado después es la siguiente:

```
 SELECT *
  FROM CLIENT
WHERE idClient NOT IN (SELECT MIN(idClient)
                       FROM CLIENT
                       GROUP BY name, surname);
```

idClient	name	surname	postalCode	city	phoneNumber
6	N5	S5	NULL	NULL	NULL
7	N3	S3	NULL	NULL	NULL

Para eliminar estos registros ejecutamos la siguiente sentencia DELETE:

```
DELETE FROM CLIENT
WHERE idClient NOT IN (SELECT MIN(idClient)
                       FROM CLIENT
                       GROUP BY name, surname);
```



Comprobamos cómo ha quedado la tabla y vemos que todo es correcto de nuevo.

idClient	name	surname	postalCode	city	phoneNumber
1	N1	S1	11111	City1	96111111
2	N2	S2	22222	City2	96222222
3	N3	S3	33333	City3	96333333
4	N4	S4	44444	City4	96444444
5	N5	S5	55555	City5	96555555