Universidad Nacional del Litoral

Facultad de Ingeniería y Ciencias Hídricas

Departamento de Informática

Bases de Datos

SQL: Guía de Trabajo Nro. 2

Data Manipulation Language

2023

1. Pre-requisitos

Ejercicio 1. A continuación definiremos un pequeño esquema de tablas para realizar ejercicios de manipulación de datos. Sobre la base de datos `pubs` ejecute las siguientes sentencias SQL. No importa si observan cláusulas que no conocen. Las analizaremos a continuación:

Sobre SQL Server:

```
CREATE TABLE cliente
  (
   codCli     int           NOT NULL,
   ape        varchar(30)   NOT NULL,
   nom        varchar(30)   NOT NULL,
   dir        varchar(40)   NOT NULL,
   codPost    char(9)       NULL DEFAULT 3000
  )
```

cliente	
codCli	int
ape	varchar(30)
nom	varchar(30)
dir	varchar(40)
codPost	varchar(9)

```
CREATE TABLE productos
  (
   codProd    int           NOT NULL,
   descr      varchar(30)   NOT NULL,
   precUnit   float         NOT NULL,
   stock      smallint      NOT NULL
  )
```

productos	
codProd	int
descr	varchar(30)
precUnit	float
stock	smallint

```
CREATE TABLE proveed
  (
   codProv    int           IDENTITY(1,1),
   razonSoc   varchar(30)   NOT NULL,
   dir        varchar(30)   NOT NULL
  )
```

proveed	
codProv	int
razonSoc	varchar(30)
dir	varchar(30)

```
CREATE TABLE pedidos
  (
   numPed     int           NOT NULL,
   fechPed    datetime      NOT NULL,
   codCli     int           NOT NULL
  )
```

pedidos	
numPed	int
fechPed	datetime
codCli	int

```
CREATE TABLE detalle
  (
   codDetalle       int          NOT NULL,
   numPed           int          NOT NULL,
   codProd          int          NOT NULL,
   cant             int          NOT NULL,
   precioTot        float        NULL
  )
```

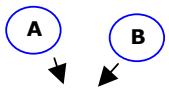
detalle	
CodDetalle	int
numPed	int
codProd	int
cant	int
precioTot	float

Sobre PostgreSQL:

```
CREATE TABLE cliente
  (
   codCli     int           NOT NULL,
   ape        varchar(30)   NOT NULL,
   nom        varchar(30)   NOT NULL,
   dir        varchar(40)   NOT NULL,
   codPost    char(9)       NULL DEFAULT 3000
  )


CREATE TABLE productos
  (
   codProd    int           NOT NULL,
   descr      varchar(30)   NOT NULL,
   precUnit   float         NOT NULL,
   stock      smallint      NOT NULL
  )


CREATE TABLE proveed
  (
   codProv    SERIAL,
   razonSoc   varchar(30)   NOT NULL,
   dir        varchar(30)   NOT NULL
  )


CREATE TABLE pedidos
  (
   numPed     int           NOT NULL,
   fechPed    date          NOT NULL,
   codCli     int           NOT NULL
  )


CREATE TABLE detalle
  (
   codDetalle       int           NOT NULL,
   numPed           int           NOT NULL,
   codProd          int           NOT NULL,
   cant             int           NOT NULL,
   precioTot        float         NULL
  )
```

Columnas con valores autoincrementales

En SQL Server, las columnas con valores autoincrementales se definen especificando la cláusula **`IDENTITY`**:




```
CREATE TABLE proveed
  (
   codProv    int           IDENTITY(1,1),
   razonSoc   varchar(30)   NOT NULL,
   dir        varchar(30)   NOT NULL,
  )
```

La cláusula posee dos parámetros opcionales: `SEED` (**A**) y `STEP` (**B**).
`SEED` es el valor inicial que recibirá la primer fila insertada. Su valor por omisión es `1`.
`STEP` es el valor de incremento entre filas consecutivas. Su valor por omisión también es `1`.

En PostgreSQL, las columnas con valores autoincrementales se definen especificando un tipo de dato especial llamado `SERIAL`:

```
CREATE TABLE proveed
    (
     codProv  SERIAL,
     razonSoc varchar(30)   NOT NULL,
     dir      varchar(30)   NOT NULL
    )

NOTICE:     CREATE   TABLE   will   create   implicit   sequence
"proveed_codprov_seq" for serial column "proveed2.codprov"
```

Tal como indica el mensaje devuelto, PostgreSQL crea una **secuencia implícita** para la columna definida con el tipo de dato `SERIAL`.

Esta secuencia posee un nombre conformado de esta manera:
`<nombre-tabla>_<nombre-columna-serial>_seq`

2. Inserción de filas

- Recordemos que insertamos filas en una tabla a través de la sentencia ANSI SQL `INSERT`. La sintaxis simplificada de `INSERT` es la siguiente:

```
INSERT [INTO] <tabla>
   [ (<columna1>, <columna2> [, <columnan> ...] ) ]
   VALUES ( <dato1> [, <dato2>...] )
```

En PostgreSQL la cláusula `INTO` es obligatoria.

- Si vamos a proporcionar datos para todas las columnas podemos omitir la lista de las mismas:

```
INSERT [INTO] <tabla>
   VALUES ( <dato1> [, <dato2>...] )
```

- Los datos de tipo `char` o `varchar` se especifican entre comillas simples. Los valores de tipo `float` se especifican con un punto decimal. (Por ejemplo: `243.2`). El formato de las fechas varía según la configuración del DBMS. Un formato usual es `aaaa/mm/dd`. El mismo se especifica entre comillas simples (Por ejemplo: `'2007/11/30'`).

Ejercicio 2. Inserte en la tabla `cliente` una fila con los siguientes datos: `1, 'LOPEZ', 'JOSE MARIA', 'Gral. Paz 3124'`. Permita que el código postal asuma el valor por omisión previsto. Verifique los datos insertados.

Ejercicio 3. Inserte en la tabla cliente una fila con los siguientes datos: `2, 'GERVASOLI', 'MAURO', 'San Luis 472'`. ¿Podemos evitar que la fila asuma el valor por omisión para el código postal?. Verifique los datos insertados.

Guía para resolver el Ejercicio 3

El `INSERT` es sencillo. La única complicación es que la tabla posee definido un valor por omisión para la columna `codPost`:

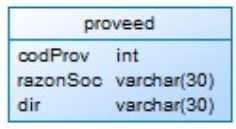
```
CREATE TABLE cliente
  (
   codCli     int           NOT NULL,
   ape        varchar(30)   NOT NULL,
   nom        varchar(30)   NOT NULL,
   dir        varchar(40)   NOT NULL,
   codPost    char(9)       NULL DEFAULT 3000
  )
```

...y por otro lado el ejercicio no nos indica valor para el código postal.
Si no especificamos Código Postal como parte de la sentencia `INSERT`, quedará almacenado en definitiva el valor especificado en la cláusula `DEFAULT`...

SQL permite especificar un valor `NULL` explícito como parte de la sentencia `INSERT`. Nuestra solución sería:

```
INSERT
   INTO cliente
   (codcli,ape, nom, dir, codPost)
   VALUES (2, 'GERVASOLI ', 'MAURO', ' San Luis 472', NULL)
```

Ejercicio 4. Inserte en la tabla `proveed` dos proveedores: `'FLUKE INGENIERIA'`, `'RUTA 9 Km. 80'` y `'PVD PATCHES'`, `'Pinar de Rocha 1154'`. Verifique los datos insertados.



Guía para resolver el Ejercicio

Recordemos que la tabla proveed fue creada con una columna `codProv` de tipo autoincremental:

```
CREATE TABLE proveed
  (
   codProv    int           IDENTITY(1,1),
   razonSoc   varchar(30)   NOT NULL,
   dir        varchar(30)   NOT NULL
  )
```

...por lo tanto, en la cláusulas `INSERT` **debemos omitir especificar** la primer columna.

Información del sistema

Usuario actual

En SQL Server, la función `USER` retorna el usuario actual de la base de datos como una cadena de caracteres.

En PostgreSQL la función `current_user` retorna el nombre del usuario actual. Las funciones `user` y `session_user` retornan la misma información.

Fecha y hora actuales

Tal como vimos en la Guía de Trabajo Nro. 1, en la Sección 4.1. *Fechas*, en SQL Server podemos obtener la fecha actual con la función `CURRENT_TIMESTAMP` y en PostgreSQL obtenemos esta información con las funciones `CURRENT_TIMESTAMP` y `now()`.

Ejercicio 5.

Defina una tabla de ventas (`Ventas`) que contenga:

-Un código de venta de tipo entero (`codVent`) autoincremental.
-La fecha de carga de la venta (`fechaVent`) no nulo con la fecha actual como valor por omisión.
-El nombre del usuario de la base de datos que cargó la venta (`usuarioDB`) no nulo con el usuario actual de la base de datos como valor por omisión.
-El monto vendido (`monto`) de tipo `FLOAT` que admita nulos.

Guía para resolver el Ejercicio

El ejercicio nos solicita que la columna `codVent` sea de tipo autoincremental, así que tendríamos:

```
codVent int IDENTITY(1,1),
```

La columna `fechaVent` debe ser no nula con la fecha actual como valor por omisión.
La fecha actual se puede especificar como `CURRENT_TIMESTAMP`, así que tendríamos:

```
fechaVent datetime NOT NULL DEFAULT CURRENT_TIMESTAMP
```

La columna `UsuarioDB` debe ser no nula y tener el nombre del usuario actual como valor por omisión.
El usuario actual se puede especificar como `USER`, así que podríamos tener:

```
usuarioDB varchar(40) NOT NULL DEFAULT USER
```

Variantes de INSERT

- Podemos crear una nueva tabla e insertar a la vez filas de una existente usando la sentencia `SELECT` con la cláusula `INTO`:

```
SELECT <lista de columnas>
   INTO <tabla-nueva>
   FROM <tabla-existente>
   WHERE <condicion>
```

- Una variante de la sentencia `INSERT` permite que insertemos en una tabla los datos de salida de una sentencia `SELECT`. La tabla sobre la que vamos a insertar debe existir previamente:


```
INSERT <tabla-destino>
   SELECT *
      FROM <tabla-origen>
      WHERE <condicion>
```


```
INSERT INTO <tabla-destino>
   SELECT *
      FROM <tabla-origen>
      WHERE <condicion>
```

Ejercicio 6. Cree una tabla llamada `clistafe` a partir de los datos de la tabla `cliente` para el código postal `3000`. Verifique los datos de la nueva tabla.

Guía para resolver el Ejercicio

Nos solicitan a crear una tabla nueva, así que podemos usar la forma `SELECT... INTO`:

```
SELECT <lista de columnas>
   INTO <tabla-nueva>
   FROM <tabla-existente>
   WHERE <condicion>
```

Donde <tabla-nueva> sería `clistafe` y `<condicion>` que la columna `codPost` tenga el valor `'3000'`.

Ejercicio 7. Inserte en la tabla `clistafe` la totalidad de las filas de la tabla `cliente`. Verifique los datos insertados.

Guía para resolver el Ejercicio

Aquí nos solicitan insertar datos en una tabla existente, así que resulta ideal la forma `INSERT... SELECT`:

```
INSERT clistafe
   SELECT *
      FROM cliente
```

3. Modificación de datos

- Recordemos que modificamos los datos de una o varias filas a través de la sentencia ANSI SQL `UPDATE`. La sintaxis simplificada de `UPDATE` es la siguiente:

```
UPDATE <tabla>
   SET <col> = <nuevo valor-o-expres> [,<col> = <nuevo-valor-o-expres>...]
   WHERE <condición>
```

Si omitimos la cláusula `WHERE`, todas las filas de la tabla resultan modificadas.

Ejercicio 8. En la tabla `cliente`, modifique el dato de domicilio. Para todas las columnas que incluyan el texto `'1'` reemplace el mismo por `'TCM 168'`.

Guía para resolver el Ejercicio

Tenemos que reemplazar el valor existente en la columna `dir` por el valor `'TCM 168'`.
La sentencia sería algo como:

```
UPDATE cliente
   SET dir = 'TCM 168'
```

cliente	
codCli	int
ape	varchar(30)
nom	varchar(30)
dir	varchar(40)
codPost	varchar(9)

...pero solo para cuando la columna `dir` posea el caracter `'1'`.

Ver "*El predicado LIKE. Caracteres comodín*" en la *Guía de Trabajo Nro. 1*.

4. Eliminación de filas

- Recordemos que eliminamos una o varias filas usando sentencia ANSI SQL `DELETE`. Su sintaxis simplificada es la siguiente:

```
DELETE [FROM] <tabla>
   WHERE <condicion>
```

Si omitimos la cláusula `WHERE`, todas las filas de la tabla resultan eliminadas.

En PostgreSQL la cláusula `FROM` es obligatoria.

Ejercicio 9. Elimine todos las filas de la tabla `cliStaFe` cuyo código postal sea nulo.

Guía para resolver el Ejercicio

Ver el Ejercicio 12 de la *Guía de Trabajo Nro. 1*.

5. Eliminar tablas

Recordemos que eliminamos una tabla a través del comando ANSI SQL `DROP TABLE`. Por ejemplo, para eliminar la tabla `cliente`:

```
DROP TABLE cliente
```

6. Obtener una copia de una tabla

PostgreSQL proporciona una sintaxis especial `CREATE TABLE` para crear una tabla con una estructura idéntica a otra existente. En el siguiente ejemplo creamos una copia (vacía por supuesto) de la tabla `titles`:

```
CREATE TABLE titles10
   (LIKE titles);
```

De todas maneras existen alternativas para obtener el mismo resultado sin usar `CREATE TABLE`. A continuación dos ejemplos:

```
SELECT *
   INTO titles10
   FROM titles
   WHERE 1=0

SELECT *
   INTO titles10
   FROM titles
   LIMIT 0
```