

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

1. Creación de tablas

Recordemos que creamos una tabla con la sentencia ANSI SQL `CREATE TABLE`:

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 observa cláusulas que no conoce. 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
)
```

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

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

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

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 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
)
```

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

1.1. Constraints Not-Null

- Tal vez la constraint más simple que se puede asociar a un atributo o columna¹ es NOT NULL (**A**). El efecto de la misma es impedir que existan tuplas o filas² en las cuales esa columna posea un valor NULL:

```
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
)
```

A

1.2. Valores por omisión

- Podemos establecer valores por omisión para una columna a través de la constraint DEFAULT (**B**). La constraint DEFAULT especifica un valor que una nueva fila deberá asumir para ese atributo cuando no se indique para el mismo un valor explícito en una sentencia DML de inserción:

```
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
)
```

B

¹ A lo largo de esta Guía de Estudio utilizaremos el término **columna** o **atributo** de manera intercambiable. El término **atributo** adhiere a la terminología del álgebra relacional, mientras que los RDBMs normalmente usan el término **columna**.

² A lo largo de esta Guía de Estudio utilizaremos el término **fila** o **tupla** de manera intercambiable. El término **tupla** adhiere a la terminología del álgebra relacional, mientras que los RDBMs normalmente usan el término **fila**.

1.3. Constraints CHECK

1.3.1. Constraints CHECK basadas en atributos


- Podemos asociar a un atributo o columna una constraint algo más compleja a través de la cláusula `CHECK` y una condición entre paréntesis. Esta condición debe ser cumplida **para todo valor** de este atributo.
- Estas constraints restringen normalmente a un simple límite entre valores, tal como una enumeración de valores permitidos o una desigualdad aritmética.
- Esta constraint es chequeada cada vez que cualquier tupla obtiene un nuevo valor para ese atributo. Recordemos que este nuevo valor puede ser introducido tanto por una sentencia `UPDATE` como por una sentencia `INSERT`.
- La constraint no es chequeada si la modificación de la base de datos no altera el atributo al cual está asociada la constraint.

Ejemplo 1

```
CREATE TABLE productos10
(
  codProd    int          NOT NULL,
  descr      varchar(30)  NOT NULL,
  precUnit   float        NOT NULL CHECK (precUnit >= 100),
  stock      smallint     NOT NULL
)
```

Ejemplo 2

```
CREATE TABLE cliente10
(
  codCli      int          NOT NULL,
  ape         varchar(30)  NOT NULL,
  nom         varchar(30)  NOT NULL,
  dir         varchar(40)  NOT NULL,
  codPost     char(9)       NULL CHECK (codPost IN(
A
  )
)
```



En el Ejemplo 2, la expresión **(A)** describe una relación de un único componente (valor de atributo) con dos tuplas. La constraint establece que el valor de cualquier componente `codPost` debe estar en este conjunto.



Si bien el estándar ANSI lo admite como posible, la constraint `CHECK` no soporta subqueries ni en SQL Server ni en PostgreSQL. Una constraint como la siguiente, que “simula” de manera “unilateral” un chequeo de integridad referencial, provoca un error:



```
CREATE TABLE pedidos3
(
  numPed    int      NOT NULL,
  fechPed   date     NOT NULL,
  codCli    int CHECK (codCli IN (SELECT codCli FROM cliente))
)
```

1.3.2. Constraints CHECK basadas en tuplas

- El estándar ANSI permite que declaremos constraints `CHECK` basadas en tuplas. Estas constraints `CHECK`, a diferencia de las basadas en atributos, **involucran más de un atributo de la tupla en cuestión**.
- La condición definida será chequeada por el DBMS **cada vez** que una fila sea insertada o actualizada en la tabla. Si la condición es falsa para esa tupla, la constraint se considera violada y la sentencia de inserción o actualización es rechazada.

Ejemplo 3

Supongamos la siguiente tabla, que almacena destinatarios de una lista de email. La columna `titulo` sirve para indicar como nos dirigimos al destinatario. Por ejemplo, `'Sr.'`, `'Sra.'`, `'Ing.'`, etc.:

```
CREATE TABLE destinatario
(
  titulo varchar(10),
  apellido varchar(50),
  nombres varchar(70),
  sexo char(1),
  CHECK ((sexo = 'F' OR titulo NOT LIKE 'Sra.') AND
        (sexo = 'M' OR titulo NOT LIKE 'Sr.'))
)
```



Hemos agregado una constraint basada en tuplas (**A**) que valida una correspondencia entre los atributos `titulo` y `sexo`.



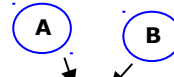
PostgreSQL no soporta constraints `CHECK` basadas en tuplas.

1.4. 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

Secuencia (sequence)

Una secuencia es una característica soportada por algunos DBMSs para generar valores enteros secuenciales únicos y asignárselos a columnas numéricas.

Internamente, una secuencia es generalmente una tabla con una columna numérica en la cual se almacena un valor que es consultado e incrementado por el sistema.



Secuencias

SQL Server permite crear SEQUENCE objects desde la versión 2012.
Ver (<https://msdn.microsoft.com/en-us/library/ff878058.aspx>)

2. Inserción de filas

- 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> [, <columna3> ...] ) ]
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` el siguiente lote de 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` el siguiente lote de 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.

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.

2.1. Información del sistema

Usuario actual



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



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.

Ejercicio 6. Inserte dos ventas de \$100 y \$200 respectivamente. No proporcione ninguna información adicional. Verifique los datos insertados.

2.2. 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 7. 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.

Ejercicio 8. Inserte en la tabla `clistafe` la totalidad de las filas de la tabla `cliente`. Verifique los datos insertados.

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

3.1. UPDATE a valores por omisión

SQL permite que, en una sentencia `UPDATE`, establezcamos el valor de una columna al valor por omisión especificado en su definición (constraint `DEFAULT`). La sintaxis a utilizar es la siguiente:

```
UPDATE <tabla>
  SET <columna> = DEFAULT
```

Ejercicio 10. Establezca el valor de la columna `codPost` de la tabla `cliente` a su valor por omisión para todas las filas de la misma.

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 11. Elimine todas las filas de la tabla `cliStaFe` cuyo código postal sea nulo.

4.1. Truncar tablas

Podemos eliminar todas las filas de una tabla conservando su estructura usando la sentencia `TRUNCATE TABLE`:

```
TRUNCATE TABLE <tabla>
```

Tanto en SQL Server como en PostgreSQL, `TRUNCATE TABLE` reinicializa las secuencias de las columnas autoincrementales.

Podríamos obtener un resultado similar disparando una sentencia `DELETE` sin cláusula `WHERE`, pero esta modalidad no reinicializa las secuencias de las columnas autoincrementales.

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
```


7. Tablas temporales

Tabla temporal

Una tabla temporal es una tabla que, una vez creada, no permanece en el schema de la base de datos como los demás objetos, sino que deja de existir luego de transcurrido un lapso de tiempo determinado.

Hay dos características que definen una tabla temporal, su **lapso de vida** (lifetime) y su **visibilidad**.



En T-SQL podemos definir **tablas temporales locales**. Las mismas se crean como cualquier otra tabla, pero su nombre debe ser precedido por el símbolo #. Por ejemplo:

```
CREATE TABLE #clienteTemp
(
  codCli      int          NOT NULL,
  ape         varchar(30)  NOT NULL,
  nom         varchar(30)  NOT NULL,
  dir         varchar(40)  NOT NULL,
  codPost     char(9)      NULL
)
```

Estas tablas son eliminadas cuando finaliza la sesión (conexión) que las creó. No son visibles desde otras sesiones a la base de datos.

Una variante son las **tablas temporales globales**. Las mismas **son visibles desde otras sesiones** diferentes a la que la creó. Se eliminan también automáticamente cuando finaliza la sesión que las creó, pero -si todavía están en uso por alguna otra sesión en ese momento- el DBMS espera hasta que se libere totalmente su uso antes de eliminarlas.

También se crean como cualquier otra tabla, pero su nombre debe ser precedido por dos símbolos #. Por ejemplo:

```
CREATE TABLE ##clienteTemp
(
  codCli      int          NOT NULL,
  ape         varchar(30)  NOT NULL,
  nom         varchar(30)  NOT NULL,
  dir         varchar(40)  NOT NULL,
  codPost     char(9)      NULL
)
```



En PostgreSQL sólo podemos definir **tablas temporales locales**. Las mismas se crean con una sintaxis `CREATE TABLE` extendida. No requieren de un nombre especial para distinguirlas. Por ejemplo:

```
CREATE TEMP TABLE clienteTemp
(
  codCli      int          NOT NULL,
  ape         varchar(30)  NOT NULL,
  nom         varchar(30)  NOT NULL,
  dir         varchar(40)  NOT NULL,
  codPost     char(9)      NULL
)
```

No son visibles desde otras sesiones a la base de datos.
Son eliminadas cuando finaliza la sesión (conexión) que las creó.

Nota: También podemos especificar, en el momento de crearlas, que sean eliminadas antes de que finalice la sesión. Específicamente, que sean eliminadas cuando finalice la transacción de la que forman parte. (Veremos transacciones en la Guía de Trabajo Nro. 4)

Ejercicio 12. Cree una tabla temporal local llamada `Tempi` con un par de columnas: `codcli` `int NOT NULL` y `ape` `varchar(30) NOT NULL`.

Ejercicio 13. Utilice `SELECT...INTO` para crear una copia temporal de los autores del estado de California (CA) de la tabla `authors`, pero solo con apellido, nombre domicilio y ciudad. Verifique los datos en la tabla temporal creada.