Tema 6: SQL - DDL y consultas de actualización (parte III)

3.1 Introducción

Hasta el momento sólo hemos visto una sentencia, el SELECT, una sentencia muy potente para poder consultar el contenido de las tablas de una Base de Datos. Pero el lenguaje SQL es más completo, y permite también crear la estructura de las tablas y otros objetos. Y también nos permitirá manipular la información, introduciendo datos nuevos, eliminando o modificando las ya existentes.

Subdivide, esta parte del tema en dos grandes bloques:

- **DDL** (*Data Definition Language*) lenguaje de definición de datos. Es lo que nos permitirá definir las estructuras de datos: tablas, vistas, y como veremos en otros temas, más objetos.
- **Consultas de actualización** . No cambian ninguna estructura de ningún mesa, sino que modifican el contenido de las tablas. Y sólo hay 3 posibilidades en la modificación del contenido: insertar nuevas filas (INSERT), modificar las ya existentes en algún campo determinado (UPDATE) o borrar filas (DELETE)

Durante toda esta tercera parte de SQL haremos consultas para crear o modificar tablas, o para modificar los datos de las tablas.

Trabajaremos con dos Bases de Datos nuevas:

- **pruebas** (conectándonos como el usuario **pruebas**): servirá para hacer pruebas, como su propio nombre indica. Todos los ejemplos los haremos en esta BD
- **factura_III** , Es donde debe trabajar los ejercicios.

Debe **Crear las Bases de Datos: pruebas y factura_III**

No debe importado datos, sólo crear la BD y el usuario propietario

- Base de Datos: pruebas
  - Crearemos un usuario -proves- que será el propietario.
- Base de Datos: factura_III
  - Crearemos un usuario -factura3- que será el propietario.

De esta manera, seguramente tendréis 4 bases de datos: la de **geo** , la de **factura** , la de **pruebas** y la de **factura_III**

Ejercicios apartado 3.1

Debe **Crear las Bases de Datos: pruebas y factura_III**

No debe importado datos, sólo crear la BD y el usuario propietario

- Base de Datos: **pruebas**
  - Crearemos un usuario -proves- que será el propietario.
- Base de Datos: **factura_III**
  - Crearemos un usuario -factura3- que será el propietario.

De esta manera, seguramente tendréis 4 bases de datos: la de **geo** , la de **factura** , la de **pruebas** y la de **factura_III**

3.2 DDL

DDL (*Data Definition Language*) o **Lenguaje de Definición de Datos** es el conjunto de sentencias que nos permiten definir, retocar o borrar la estructura de la Base de Datos. Y como la estructura básica de una Base de Datos Relacional es la mesa, nos dedicaremos básicamente a estudiar las sentencias que nos permiten definir las tablas (o modificarlas o borrarlas), con todas las restricciones que hemos visto en el Modelo Relacional: clave principal, claves externas, campos no nulos, ... También veremos otros objetos que podremos definir, sobre todo **vistas** , que se corresponden al esquema externo que vimos en el Tema 1, es decir, la visión particular que puede tener un usuario.

Serán 3 sentencias las que veremos:

- **CREATE** , que permite crear un objeto nuevo.
- **DROP** , que permite borrar un objeto ya existente.
- **ALTER** , que permite modificar un objeto ya existente.

En el momento de crear una mesa definiremos todos sus campos, con las restricciones pertinentes a cada uno de ellos. Cada campo deberá ser de un tipo de datos. En cada SGBD hay unos tipos de datos particulares, aunque los más básicos son similares, y en ellos será donde incidiremos más.

3.2.1 Tipos de datos

En el momento de definir un campo deberemos especificar obligatoriamente de qué tipo será. Ya se vieron los tipos básicos de Access en el tutorial del tema 5. Ahora los veremos los tipos básicos de **PostgreSQL** , y veremos que habrá muchos tipos similares (como en todos los SGBD).

En el siguiente cuadro se resumen los tipos de datos más importantes de PostgreSQL. Es un conjunto muy extenso, que incluso puede ampliar el usuario con la instrucción **CREATE TYPE** , como veremos al final del tema. Son especialmente interesantes los tipos geométricos (con ***POINT*** , ***BOX*** , ...) y el **INET** (dirección IP).

	TIPO DE DATOS	DESCRIPCIÓN	BYTES
CARÁCTER	**CHAR**	Un carácter.	1 byte
	CHAR (n)	Cadena fija de n caracteres.	(4 + n) bytes
	VARCHAR (*n*)	Cadena de caracteres de longitud variable, con un máximo de *n* caracteres. Se debe especificar la longitud.	(4 + x) bytes
	TEXTO	Al igual que el anterior, pero no se debe especificar la longitud máxima.	(4 + x) bytes
NUMÉRIC	**DECIMAL (*n, d*)**	Número con una precisión de *n* cifras, con *de* decimales. Si no se pone *de* no hay decimales. La precisión máxima es de 1000 cifras.	variable
	NUMERIC (*n, d*)	Al igual que la anterior	
	FLOAT4, REAL	Coma flotante de simple precisión (6 cifras decimales)	4 bytes
	FLOAT8, FLOAT, DOUBLE PRECISION	Coma flotante doble precisión (15 cifras decimales)	8 bytes
	Int2, small	Enter (-32768,32767)	2 bytes
	INT4, INT, INTEGER	Enter (-2147483648, 2147483647)	4 bytes
	INT8, BIGINT	Enter con unas 18 cifras	8 bytes
	SERIAL	Autonumérico (internamente se crea una secuencia)	4 bytes
DATA	**DATE**	Tipo fecha. Valor mínimo 1-1-4713 AC. Valor máximo 31-12-5874897 DC.	4 bytes
	TIME	Tipo hora. Guarda hasta la micra de segundo	8 bytes
	TIMESTAMP	Tipo fecha-hora (combinando las características los dos anteriores)	8 bytes
	INTERVALO	Un intervalo de tiempo (con precisión de un microsegundo, pero que puede llegar a los 178 millones años)	12 bytes
	BOOL	Booleano, con valores True y False	1 byte
GEOMÉTRIC	**POINT**	Un punto del espacio bidimensional (x, y) (dos float8)	16 bytes
	LSEG	Segmento de línea definido por 2 puntos (x1, y1) (x2, y2)	32 bytes
	BOX	Rectángulo, definido por los extremos (x1, y1) (x2, y2)	32 bytes
	PATH	Conjunto de puntos que representan una figura abierta o cerrada: (x1, y1), ..., (xn, yn)	16 + 16n bytes
	Polygon	Conjunto de puntos que representan un figura cerrada (x1, y1), ..., (xn, yn) (similar al path cerrado)	40 + 16n bytes
	CIRCLE	Círculo representado por el centro y el radio	24 bytes
	INET	Dirección IP, de 4 números separados por puntos, con número de bits de la máscara separado para /	variable

También dispondremos de un tipo **enumerado** . El veremos en la última pregunta del tema.

En la documentación de PostgreSQL encontraremos todos los tipos posibles:
http://www.postgresql.org/docs/9.5/static/datatype.html

3.2.2 CREATE TABLE

Permite crear una nueva tabla. Obligatoriamente deberán especificar los campos y los tipos de datos de cada campo. Obviamente, una vez creada la tabla estará vacía, sin ninguna fila.

sintaxis

```
CREATE TABLE tabla
     (campo1 tipo [(tamaño)] [DEFAULT valor] [restricció11] [restricció12] [...]
    [, camp2 tipo [(tamaño)] [DEFAULT valor] [restricció21] [restricció22] [...]
    [...]]
    [, restriccióMultiple1 [...]])
```

Podemos observar que la definición de la estructura de la mesa va entre paréntesis, separando por comas la definición de cada campo.

- El nombre de la tabla no debe ser el de ningún otro objeto anterior (tabla o vista). Si queremos poner un nombre con más de una palabra o con una palabra reservada, la tendremos que poner entre comillas dobles; pero no se lo aconseje, es preferible la utilización del guión bajo, y así sólo es una palabra.
- En cada campo pondremos su nombre y el tipo. Si el tipo de datos es VARCHAR, podremos poner opcionalmente el tamaño máximo (si no la ponemos será de 255 en el caso de texto). Si el tipo de datos es NUMERIC, podremos poner opcionalmente el tamaño (número de cifras significativas) y número de cifras de la parte fraccionaria.
- Podemos poner ocionalment un valor por defecto con la cláusula **DEFAULT** . De este modo, al introducir una nueva fila en la tabla, si no le ponemos valor a este campo, tomará el valor predeterminado. En el valor se puede poner una constante del tipo del campo, o una expresión con funciones, siempre que vuelva un dato de los tipos del campo.
- Podemos poner opcionalmente restricciones a cada campo. Deberán ir antes de la coma que separa del cuadro. También pueden haber restricciones que afectan a más de un campo, que preferiblemente pondremos al final de la definición de la tabla. Veremos las restricciones en el siguiente punto.

ejemplos

Si desea practicar estos ejemplos, haga que sobre la Base de Datos **pruebas** . Por da error porque la mesa que vaya a crear ya está creada la tabla, elimínelo la primero, y vuelva a ejecutar la sentencia.

1. Crear una nueva tabla llamada **EMPLEAT1** con dos campos, uno llamado **dni** de tipo texto y longitud 10 y otro llamado **nombre** con longitud 50.

```
CREATE TABLE EMPLEAT1 (dni VARCHAR (10), nombre varchar (50));
```

2. Crear una tabla llamada **EMPLEAT2** con un campo texto de 10 caracteres llamado **dni** ; otro campo de tipo texto de longitud predeterminada (255) llamado **nombre;** otro campo llamado **fecha_nacimiento** de tipo fecha; otro llamado **sueldo** de tipo numérico, con 6 cifras significativas, de las cuales 2 debe ser de la parte fraccionaria y un último llamado **departamento** de tipo numérico pequeñito (Int2 o small).

```
CREATE TABLE EMPLEAT2
    (dni VARCHAR (10),
      nombre VARCHAR,
```

```
      fecha_nacimiento DATE,
      sueldo NUMERIC (6,2),
      departamento Int2)
```

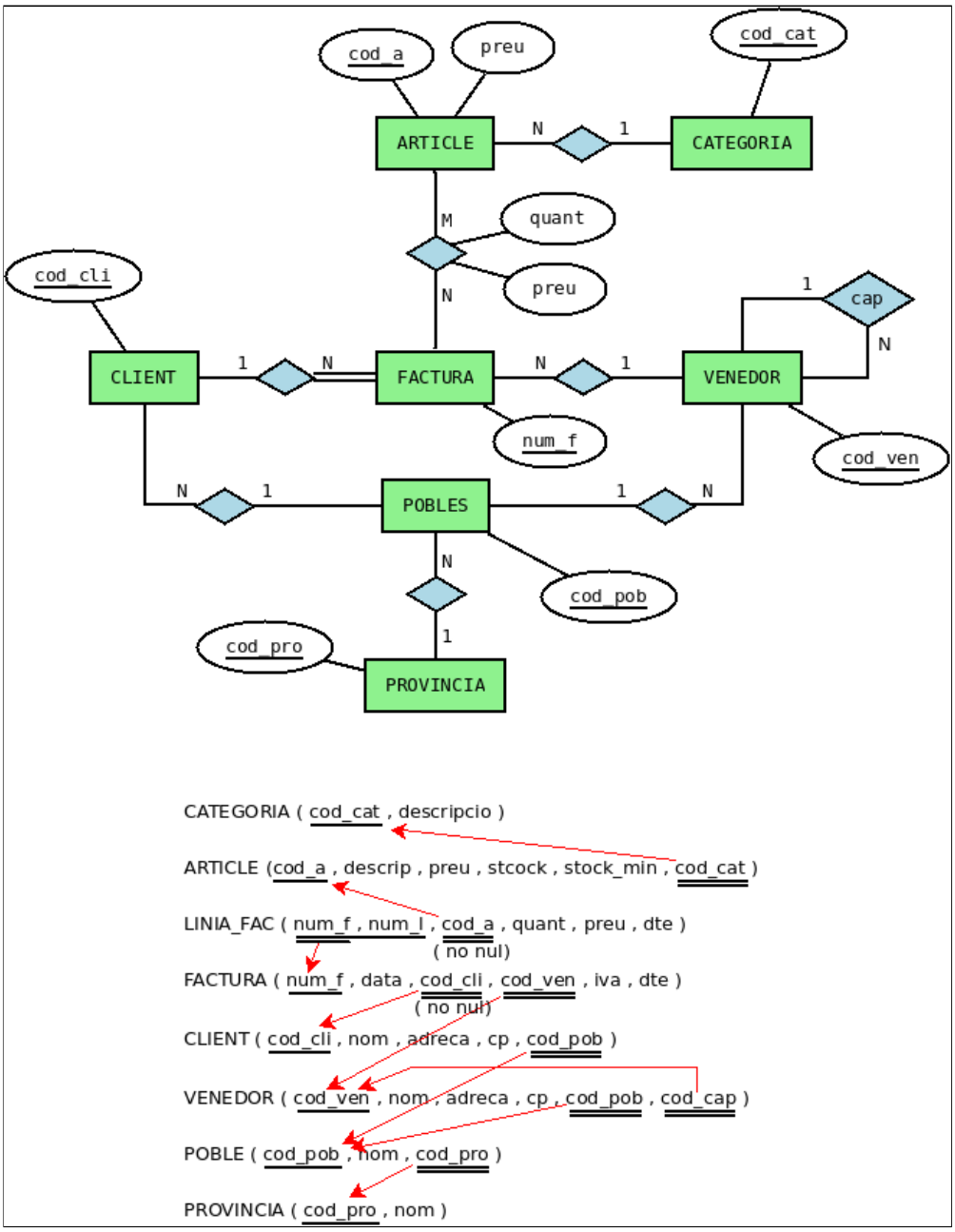
3. Crear una tabla llamada **EMPLEAT3** como el del ejemplo anterior, pero con dos campos más al final: un campo llamado **población** de tipo texto de 50 caracteres, y con el valor por defecto **Castellón** y un último llamado **data_incorporacio** de tipo fecha y valor para defecto la fecha de hoy

```
CREATE TABLE EMPLEAT3
    (dni VARCHAR (10),
      nombre VARCHAR,
      fecha_nacimiento DATE,
      sueldo NUMERIC (6,2),
      departamento Int2,
      población VARCHAR (50) DEFAULT 'Castellón',
      data_incorporacio DATE DEFAULT CURRENT_DATE)
```

Ejercicios apartado 3.2.2

A lo largo de esta tercera parte, en el conjunto de ejercicios de DDL, crearemos toda la estructura de la Base de Datos **FACTURA** y lo haremos sobre la BD que hemos creado: **Factura_III**

El esquema Entidad-Relación y el esquema relacional que implementaremos será el siguiente:



En la Base de Datos llamada **FACTURA_III** :

6.77 Crea la tabla **CATEGORÍA** , con los mismos campos y del mismo tipo que en la tabla CATEGORÍA de **FACTURA** , pero de momento sin clave principal ni ninguna otra restricción. Guarde la consulta de creación como **Ex_6_77.sql**

6.78 Crea la tabla **ARTÍCULO** , también sin restricciones. Guardar la consulta como **Ex_6_78.sql**

Nota

Durante todos estos ejercicios de DDL puede ser muy conveniente tener abiertas las dos conexiones: la de **FACTURA** (para ir consultando) y la de **FACTURA_III** (para ir creando y modificando)

3.2.3 Restricciones (Constraint)

Por medio de las restricciones podremos definir dentro de una tabla restricciones de usuario como son la definición de la clave principal, claves externas, campos no nulos y campos únicos.

Hay dos formas de definir restricciones: las que afectan a un único campo (y que se ponen en la misma definición del campo) y las que afectan o pueden afectar a más de un campo, que se han definido por separado de la definición de los campos. Empezamos por las primeras, por ser más sencillas de entender:

Restricciones de campo único

Son restricciones que se ponen en la misma definición del campo y sólo afectarán a este campo: van por tanto después del tipo de datos del campo y antes de la coma de separación de los campos.

La sintaxis es

```
[CONSTRAINT nombre] {PRIMARY KEY | UNIQUE | NOT NULL | REFERENCES Tabla2 [(campo1)] |
    CHECK ( condición )}
```

Si no ponemos nombre a la restricción (CONSTRAINT nombre) PostgreSQL le asignará automáticamente un nombre. Esto puede resultar cómodo en ocasiones, para no tener que inventarnos nombres para las restricciones, pero luego nos limitaría a que no podríamos retocar estas restricciones.

Los tipos de restricciones que podemos definir son:

- **PRIMARY KEY** : el campo será clave principal.

  Por ejemplo, de esta manera definiremos la mesa EMPLEAT3 (como la del apartado anterior) con el campo dni como clave principal. Recuerde que la tiene que eliminar primero (tal vez no lo esté viendo en pgAdmin, pero sí está creada; refrescarse constantemente las tablas para saber la situación actual)

  ```
  CREATE TABLE EMPLEAT3
      (dni VARCHAR (10) CONSTRAINT cp_emp3 PRIMARY KEY,
        nombre VARCHAR,
        fecha_nacimiento DATE,
        sueldo NUMERIC (6,2),
        departamento Int2,
        población VARCHAR (50) DEFAULT 'Castellón',
        data_incorporacio DATE DEFAULT CURRENT_DATE)
  ```

  Nota

  Puede comprobar que si no ponga nombre a la restrició, es decir colocando directamente dni **VARCHAR** `(10) PRIMARY KEY` , y vaya al diseño de la mesa (haciéndole clic desde pgAdmin; recuerde que debe refrescar para que aparezcan los nuevos objetos), PostgreSQL pone automáticamente un nombre a la restricción formado por el nombre de la tabla seguido de **_pkey** .

  Tenga en cuenta también que si la tabla ya existía dará un error. Sólo tiene que eliminar primero.

- **UNIQUE** : el campo será único, es decir, no se podrá coger dos veces el mismo valor en este campo (*Indexado sin duplicados* en Access). PostgreSQL generará automáticamente un índice para este campo. Veremos qué es un índice en la pregunta 3.2.6.

  Por ejemplo, de esta manera definiríamos la mesa EMPLEAT3 con la restricción de que el campo **nombre** no se puede repetir (si desea probar la sentencia haga que en la BD **pruebas** , y si ya existe la elimine primero):

```
CREATE TABLE EMPLEAT3
    (dni VARCHAR (10),
      nombre VARCHAR CONSTRAINT u_nom UNIQUE,
      fecha_nacimiento DATE,
      sueldo NUMERIC (6,2),
      departamento Int2,
      población VARCHAR (50) DEFAULT 'Castellón',
      data_incorporacio DATE DEFAULT CURRENT_DATE)
```

- **NOT NULL** : el campo no podrá coger un valor nulo (*Requerido* en Access). Debemos ser conscientes de que no vale la pena definir como no nula la clave principal, ya que por definición ya lo es.

  Por ejemplo, de esta manera definiremos que el campo **nombre** debe ser no nulo.

```
CREATE TABLE EMPLEAT3
    (dni VARCHAR (10),
      nombre VARCHAR CONSTRAINT nn_nom NOT NULL,
      fecha_nacimiento DATE,
      sueldo NUMERIC (6,2),
      departamento Int2,
      población VARCHAR (50) DEFAULT 'Castellón',
      data_incorporacio DATE DEFAULT CURRENT_DATE)
```

- **REFERENCES** : servirá para definir que este campo es una clave externa. Tendremos que especificar obligatoriamente la tabla a la que apunta, y opcionalmente podemos poner entre paréntesis el campo de la tabla al que apunta, aunque si no lo ponemos, por defecto apuntará a la clave principal (y nosotros siempre querremos apuntar a la clave principal).

  Por ejemplo, de esta manera podemos definir la clave externa que apunta a la tabla DEPARTAMENTO (y que indica que el empleado pertenece al departamento). Antes de crear esta versión de EMPLEAT3, debemos tener creada la tabla DEPARTAMENTO, sino dará error:

```
CREATE TABLE DEPARTAMENTO
    (num_d Int2 CONSTRAINT cp_dep PRIMARY KEY,
      nom_d VARCHAR (50),
      director VARCHAR (10),
      fecha DATE);
```

```
CREATE TABLE EMPLEAT3
    (dni VARCHAR (10),
      nombre VARCHAR,
      fecha_nacimiento DATE,
      sueldo NUMERIC (6,2),
      departamento Int2 CONSTRAINT ce_emp3_dep REFERENCES DEPARTAMENTO,
      población VARCHAR (50) DEFAULT 'Castellón',
      data_incorporacio DATE DEFAULT CURRENT_DATE);
```

  Como ya se vio en el tema 3 (Modelo Relacional) y en el tema 5 (Access), hay 3 formas de actuar cuando se borra o se modifica una fila de la tabla principal que tiene asociadas filas en la tabla relacionada por medio de la clave externa. Por ejemplo, ¿qué hacemos con los familiares de un empleado si borramos el empleado? Estos modos de actuar se especificarán en el momento de definir la clave externa. La manera de ponerlas en SQL y el significado son las siguientes:

    - **NO ACTION** : no se dejará borrar o modificar de la tabla principal si tiene alguna fila relacionada. Es la opción por defecto. Así en el ejemplo de EMPLEAT3, con una clave externa que apunta a DEPARTAMENTO, si intentamos borrar o modificar el numero de un departamento que tiene empleados, nos dará un mensaje de error, avisando que como tiene registros relacionados en otra mesa no se puede borrar o modificar.

- **CASCADE** : borrarán (o modificarán) en cascada los registros relacionados de la tabla donde está la clave externa. Se especificará con **ON DELETE CASCADE** o **ON UPDATE CASCADE** .

```
CREATE TABLE EMPLEAT3
    (dni VARCHAR (10),
      nombre VARCHAR,
      fecha_nacimiento DATE,
      sueldo NUMERIC (6,2),
      departamento Int2 CONSTRAINT ce_emp3_dep REFERENCES
    DEPARTAMENTO DONDE DELETE CASCADE ON UPDATE CASCADE,
      población VARCHAR (50) DEFAULT 'Castellón',
      data_incorporacio DATE DEFAULT CURRENT_DATE)
```

  De esta manera si borramos un departamento de la tabla DEPARTAMENTO, se borrarán también los empleados de la tabla EMPLEATS3 de este departamento. Y si en la tabla DEPARTAMENTO modificamos un número de departamento, por ejemplo de 5 a 50, este valor será el nuevo valor en el campo departamento de la tabla EMPLEAT3 para aquellos que antes tenían un 5.

- **SET NULL** : pondrá a nulo el campo que es clave externa de los registros que estén relacionados con el borrado o modificado de la tabla principal. Así, si hicimos la siguiente definición de la tabla EMPLEAT3

```
CREATE TABLE EMPLEAT3
    (dni VARCHAR (10),
      nombre VARCHAR,
      fecha_nacimiento DATE,
      sueldo NUMERIC (6,2),
      departamento Int2 CONSTRAINT ce_emp3_dep REFERENCES
    DEPARTAMENTO DONDE DELETE SET NULL,
      población VARCHAR (50) DEFAULT 'Castellón',
      data_incorporacio DATE DEFAULT CURRENT_DATE)
```

  en caso de que borramos el departamento 5, no daría ningún error por esta restricción de integridad, y pondría a nulo el departamento de aquellos empleados que antes eran del departamento 5.

- **CHECK** : hará una comprobación para validar los valores introducidos para este campo. La condición de validación debe ir entre paréntesis, y debe ser una expresión, normalmente de comparación del campo en cuestión con algún valor.

  Por ejemplo, vamos a exigir que el sueldo sea estrictamente positivo (por tipo de datos numérico, podría tomar el valor 0 o valores negativos)

```
CREATE TABLE EMPLEAT3
    (dni VARCHAR (10),
      nombre VARCHAR,
      fecha_nacimiento DATE,
      sueldo NUMERIC (6,2) CONSTRAINT sou_positiu CHECK (sueldo> 0),
      departamento Int2,
      población VARCHAR (50) DEFAULT 'Castellón',
      data_incorporacio DATE DEFAULT CURRENT_DATE)
```

Evidentemente se puede poner más de una restricción en la definición de una mesa. En este ejemplo recogemos todas las anteriores, es decir, definimos la tabla **EMPLEAT3** con todos sus campos, y definiendo la *clave principal* (**dni**), con el campo **nombre** *único* , con el **sueldo** *estrictamente positivo* , y con el campo **departamento** que será *clave externa* que apunta a la tabla DEPARTAMENTO. Para complicarlo un poco más también exigiremos que el campo **nombre** sea *no nulo* , y así ver que se puede poner más de una restricción en un campo.

```
CREATE TABLE EMPLEAT3
    (dni VARCHAR (10) CONSTRAINT cp_emp3 PRIMARY KEY,
```

```
    nombre VARCHAR CONSTRAINT u_nom UNIQUE CONSTRAINT nn_nom NOT NULL,
    fecha_nacimiento DATE,
    sueldo NUMERIC (6,2) CONSTRAINT sou_positiu CHECK (sueldo> 0),
    departamento Int2 CONSTRAINT ce_emp3_dep REFERENCES DEPARTAMENTO,
    poblacion VARCHAR (50) DEFAULT 'Castellón',
    data_incorporacio DATE DEFAULT CURRENT_DATE)
```

Obsérvese que como cuestión de estilo se han puesto nombres a las restricciones que de alguna manera sugieren el motivo de la restricción. Así, **cp_emp3** significa *clave princpal de EMPLEAT3* , **u_nom** significa que el campo *nombre* es *único* , **nn_nom** significa que *nombre* es *no nulo* , **nn_sou** significa que *usted* es *no nulo* , y **ce_emp3_dep** significa *clave externa de la tabla EMPLEAT3 la tabla DEPARTAMENTO*. Si tenemos un criterio claro para los nombres de las restricciones, si después las queremos desactivar temporalmente o sencillamente borrarlas, lo podremos hacer desde SQL.

Restricciones de campo múltiple

También se denominan restricciones de mesa, en contraposición a las anteriores, que son restricciones de campo. Son restricciones que van dentro de la definición de una mesa pero fuera de la definición de un campo, y que pueden afectar a uno o más de un campo. Se deberá definir expresamente a cuál o cuáles campos afectan.

La sintaxis general, en esta ocasión es

```
[CONSTRAINT nombre] {PRIMARY KEY | UNIQUE | FOREIGN KEY | CHECK ( condición )} (c11
    [, c12] [...])
    [REFERENCES Tabla2 [(c21 [, C22] [...])]]
    [ON DELETE {CASCADE | SET NULL}] [ON UPDATE {CASCADE | SET NULL}]]
```

Al igual que antes, si no ponemos nombre a la restricción (CONSTRAINT nombre) PostgreSQL le asignará uno automáticamente, que será construido de manera muy lógica.

Observe que ahora siempre especificamos el o los campos afectados.

Los tipos de restricciones son los mismos que en el caso anterior, pero la sintaxis variará ligeramente:

- **PRIMARY KEY** : pondremos entre paréntesis el campo o campos (en este caso separados por comas) que serán clave principal.

  Por ejemplo, definimos otra vez el campo dni como clave principal de la tabla EMPLEAT3

  ```
  CREATE TABLE EMPLEAT3
      (dni VARCHAR (10),
        nombre VARCHAR,
        fecha_nacimiento DATE,
        sueldo NUMERIC (6,2),
        departamento Int2,
        población VARCHAR (50) DEFAULT 'Castellón',
        data_incorporacio DATE DEFAULT CURRENT_DATE,
        CONSTRAINT cp_emp3 PRIMARY KEY (dni))
  ```

  Y ahora otro para definir la clave principal de FAMILIAR. Como la clave está formada por 2 campos, estamos obligados a utilizar una restricción de campo múltiple.

  ```
  CREATE TABLE FAMILIAR
      (dni VARCHAR (10),
        nombre VARCHAR,
        data_n DATE,
        parentesco VARCHAR (50),
        CONSTRAINT cp_fam2 PRIMARY KEY (dni, nombre))
  ```

Como comentábamos, si la clave principal está formada por 2 campos estaremos obligados a utilizar una restricción de campo múltiple. Un **error bastante común** sería el siguiente:

```
CREATE TABLE FAMILIAR2
    (dni VARCHAR (10) PRIMARY KEY,
      nombre VARCHAR PRIMARY KEY,
      data_n DATE,
      parentesco VARCHAR (50))
```

Puede comprobar que dará **error** , porque estamos intentando definir 2 claves principales. La clave principal es única, eso sí formada por 2 campos en esta ocasión.

- **UNIQUE** : ahora pondremos entre paréntesis el o los campos que serán únicos (en su conjunto). PostgreSQL generará automáticamente un índice para esta combinación de campos. Veremos qué es un índice en la pregunta 3.2.6.

  Por ejemplo, modificamos la definición de EMPLEAT3 (llamándola EMPLEAT4), con un campo para los **apellidos** y un campo para el **nombre** . Definiremos la restricción que los campos apellidos y nombre (en conjunto) no se pueden repetir.

```
CREATE TABLE EMPLEAT4
    (dni VARCHAR (10),
      apellidos VARCHAR,
      nombre VARCHAR,
      fecha_nacimiento DATE,
      sueldo NUMERIC (6,2),
      departamento Int2,
      CONSTRAINT u_nom4 UNIQUE (apellidos, nombre))
```

- **NOT NULL** .

  No existe esta opción como restricción múltiple. Por lo tanto se debe definir siempre como restricción de campo único.

- **FOREIGN KEY** : servirá para definir que este o estos campos son una clave externa. Es la que más varía en su sintaxis, ya que tenemos que especificar tanto el o los campos de esta tabla que son clave externa, como la tabla a la que apunta (y en todo caso el o los campos donde se apunta , aunque si no lo ponemos apuntará a la clave principal de la otra mesa, lo que querremos siempre):

```
[CONSTRAINT nombre] FOREIGN KEY (c11 [, c12] [...]) REFERENCES Tabla2 [(c21 [, C22]
    [...])] [ON DELETE {CASCADE | SET NULL}] [ON UPDATE {CASCADE | SET NULL}]
```

  En el ejemplo de la clave externa que apunta a la tabla DEPARTAMENTO quedará así:

```
CREATE TABLE EMPLEAT3
    (dni VARCHAR (10),
      nombre VARCHAR,
      fecha_nacimiento DATE,
      sueldo NUMERIC (6,2),
      departamento Int2,
      población VARCHAR (50) DEFAULT 'Castellón',
      data_incorporacio DATE DEFAULT CURRENT_DATE,
      CONSTRAINT ce_emp3_dep FOREIGN KEY (departamento) REFERENCES DEPARTAMENTO)
```

- **CHECK** : ahora la condición de validación podrá afectar a más de un campo

  Por ejemplo podríamos exigir que la fecha de incorporación sea estrictamente posterior a la fecha de nacimiento

```
CREATE TABLE EMPLEAT3
    (dni VARCHAR (10),
      nombre VARCHAR,
      fecha_nacimiento DATE,
```

```
      sueldo NUMERIC (6,2),
      departamento Int2,
      población VARCHAR (50) DEFAULT 'Castellón',
      data_incorporacio DATE DEFAULT CURRENT_DATE,
      CONSTRAINT check_dates CHECK (data_incorporacio> fecha_nacimiento) )
```

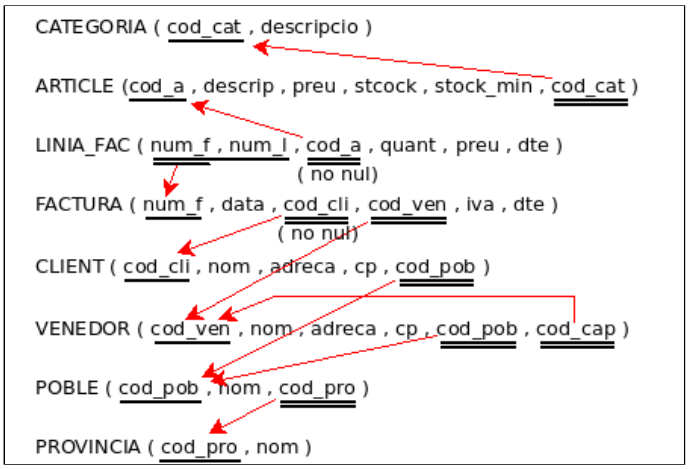
U otra algo más real, vamos a coger empleados de más de 18 años, y por lo tanto vamos a exigir que la fecha de incorporación sea más de 18 años posterior a la fecha de nacimiento. Para ello utilizamos la función **AGE (f1, f2)** que calcula el tiempo entre la fecha d2 y la fecha de 1 (que debe ser la posterior), y de ahí extraeremos los años con **EXTRACT (year FROM ...)**

```
CREATE TABLE EMPLEAT3
    (dni VARCHAR (10),
      nombre VARCHAR,
      fecha_nacimiento DATE,
      sueldo NUMERIC (6,2),
      departamento Int2,
      población VARCHAR (50) DEFAULT 'Castellón',
      data_incorporacio DATE DEFAULT CURRENT_DATE,
      CONSTRAINT check_dates
        CHECK (EXTRACT (year FROM AGE (data_incorporacio, fecha_nacimiento))> =
    18))
```

Evidentemente, se pueden mezclar las restricciones de campo único y las de campo múltiple. Aquí tenemos un ejemplo donde se recogen muchas (no todas) las restricciones anteriores. Hemos puesto de campo múltiple la de los 18 años de los empleados, porque no hay otro remedio, y también la de no repetición del campo **nombre** , aunque podía ser de campo único:

```
CREATE TABLE EMPLEAT3
    (dni VARCHAR (10) CONSTRAINT cp_emp3 PRIMARY KEY,
      nombre VARCHAR CONSTRAINT nn_nom NOT NULL,
      fecha_nacimiento DATE,
      sueldo NUMERIC (6,2) CONSTRAINT sou_positiu CHECK (sueldo> 0),
      departamento Int2 CONSTRAINT ce_emp3_dep REFERENCES DEPARTAMENTO,
      población VARCHAR ( 50) DEFAULT 'Castellón',
      data_incorporacio DATE DEFAULT CURRENT_DATE,
      CONSTRAINT u_nom3 UNIQUE (nombre),
      CONSTRAINT check_dates
        CHECK (EXTRACT (year FROM AGE (data_incorporacio, fecha_nacimiento))> = 18))
```

Ejercicios apartado 3.2.3



en **FACTURA_III**

6.79 Crear la tabla **PROVINCIA** , con la llave principal.

6.80 Crear la tabla **PUEBLO** , con la clave principal y la restricción que el campo **cod_pro** es clave externa que apunta a PROVINCIA.

6.81 Crear la tabla **VENDEDOR** , con la clave principal y la clave externa a PUEBLO (de momento no definimos la clave externa a VENDEDOR, que es reflexiva).

6.82 Crear la tabla **CLIENTE** , con la clave principal y la clave externa a PUEBLO

6.83 Crear la tabla **FACTURA** , con la llave principal y las claves externas a CLIENTE y VENDEDOR. También debe exigir que **cod_cli** sea no nulo.

6.84 Crear la tabla **LINIA_FAC** , con la clave principal (observa que está formada por 2 campos) pero de momento sin la clave externa que apunta a ARTÍCULO. Además **cod_a** debe ser no nulo.

3.2.4 ALTER TABLE

Permite modificar la estructura de una tabla ya existente, bien añadiendo, quitando o modificando campos (columnas), bien añadiendo o quitando restricciones. También servirá para cambiar el nombre de un campo e incluso cambiar el nombre de la tabla

sintaxis

Para alterar la estructura de algún campo o restricción utilizaremos esta sintaxis:

```
ALTER TABLE tabla
    {ADD | DROP | ALTER} {COLUMN campo | CONSTRAINT restricciómúltiple}
```

Para cambiar el nombre de un campo:

```
ALTER TABLE tabla
    RENAME [COLUMN] campo TO nou_nom_camp
```

Para cambiar el nombre de la tabla:

```
ALTER TABLE tabla
    RENAME TO nou_nom_taula
```

Añadir campo o restricción

Si queremos añadir una columna o una restricción, la tendremos que definir totalmente.

- En el caso de un campo, tendremos que especificar el nombre, el tipo y opcionalmente una restricción que afecte sólo al campo. Por ejemplo, esta sentencia añade el campo supervisor (de tipo texto de 10) en la tabla EMPLEAT3. Obsérvese que en la definición del campo pueden entrar restricciones de campo único.

  ```
  ALTER TABLE EMPLEAT3
      ADD COLUMN supervisor VARCHAR (10)
  ```

- En el caso de una restricción, ésta será del tipo de restricción múltiple, con la sintaxis que vimos en el apartado de restricciones. Por ejemplo, esta sentencia añade la clave externa reflexiva (de EMPLEAT3 a EMPLEAT3) que indica los supervisores. El dni debería ser la clave principal de EMPLEAT3

  ```
  ALTER TABLE EMPLEAT3
      ADD CONSTRAINT ce_emp3_emp3 FOREIGN KEY (supervisor) REFERENCES EMPLEAT3
      (dni)
  ```

Modificar un campo

Podemos hacer dos cosas: modificar el tipo del campo o modificar el valor por defecto (poner valor predeterminado o quitarlo)

Para cambiar el tipo tendremos que utilizar la sintaxis **... ALTER COLUMN *campo* TYPE *nou_tipus*** . Por ejemplo vamos a hacer que el campo población sea de 25 caracteres

```
ALTER TABLE EMPLEAT3
    ALTER COLUMN población TYPE VARCHAR (25)
```

Cambiar el tipo de datos es automático cuando los tipos son compatibles entre ellos. Si no lo son nos dará error, pero seguramente lo podremos esquivar con la cláusula **USING** , que nos permite poner a continuación el campo y aprovechamos para poner un **operador de conversión de tipo** (**::**) con esta sintaxis:

```
ALTER TABLE TABLA
    ALTER COLUMN campo TYPE tipus_nou USING campo :: tipus_nou
```

Para cambiar el valor por defecto utilizaremos la sintaxis: **... ALTER COLUMN *campo* {SET | DROP} DEFAULT [*expresión*]**

```
ALTER TABLE EMPLEAT3
    ALTER COLUMN población DROP DEFAULT
```

Borrar campo o restricción

Si queremos quitar un campo o una restricción es suficiente con especificar el nombre del campo o de la restricción (por eso puede ser muy interesante dar nombre a las restricciones). En el primer ejemplo quitamos la clave externa del supervisor. En la segunda quitamos el campo supervisor.

```
ALTER TABLE EMPLEAT3
    DROP CONSTRAINT ce_emp3_emp3;
```

```
ALTER TABLE EMPLEAT3
    DROP COLUMN supervisor
```

Renombrar un campo

Por ejemplo renombrar el campo **data_incorporacio** a **data_inc** :

```
ALTER TABLE EMPLEAT3
    RENAME COLUMN data_incorporacio TO data_inc
```

Renombrar la tabla

Ahora le pondremos el nombre EMP3 en la tabla EMPLEAT3

```
ALTER TABLE EMPLEAT3
    RENAME TO EMP3
```

ejemplos

1. Modificar la tabla **EMP3** para añadir el campo cp (código postal) de tipo texto de 5 caracteres.

```
ALTER TABLE EMP3 ADD COLUMN cp VARCHAR (5);
```

2. Modificar la tabla **EMP3** para modificar el campo anterior y que sea de tipo numérico.

```
ALTER TABLE EMP3 ALTER COLUMN cp TYPE NUMERIC (5) USING CP :: NUMERIC;
```

3. Modificar la tabla **EMP3** para añadir la restricción (aunque sea un poco extraña) que no se puede repetir la combinación código postal y población.

```
ALTER TABLE EMP3 ADD CONSTRAINT u_cp_pobl UNIQUE (cp, población);
```

4. Modificar la tabla **EMP3** para borrar la restricción anterior

```
ALTER TABLE EMP3 DROP CONSTRAINT u_cp_pobl;
```

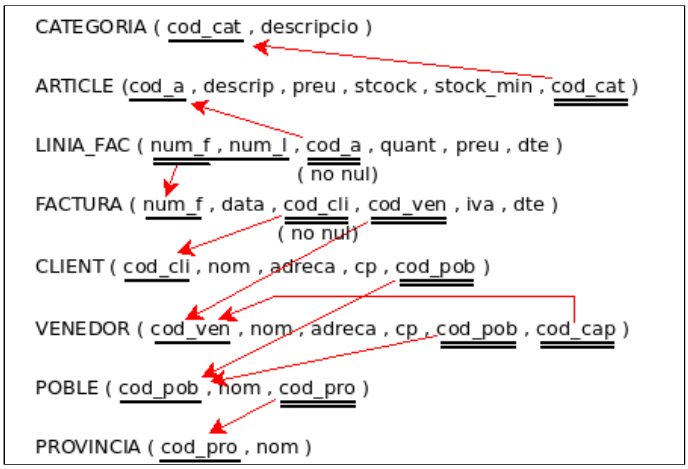
5. Modificar la tabla **EMP3** para modificar el nombre del campo **cp** a **codi_postal**

```
ALTER TABLE EMP3 RENAME COLUMN cp TO codi_postal;
```

6. Renombrar la tabla **EMP3** a **EMPLEAT3**

```
ALTER TABLE EMP3 RENAME TO EMPLEAT3;
```

Ejercicios apartado 3.2.4



en **FACTURA_III**

6.85 Añadir un campo a la mesa **VENDEDOR** llamado **alias** de tipo texto, que debe ser no nulo y único.

6.86 Borrar el campo anterior, **alias** , de la tabla **VENDEDOR** .

6.87 Añadir la clave principal de **CATEGORÍA** .

6.88 En la tabla **ARTÍCULO** añadir la clave principal y la clave externa a CATEGORÍA.

6.89 En la tabla **LINIA_FAC** añadir la clave externa que apunta a FACTURA, **exigiendo que se borre en cascada** (si se borra una factura, se borrarán automáticamente sus líneas de factura). Y también la clave externa que apunta a ARTÍCULO (esta normal, es decir NO ACTION)

Nota

Para no hacerlo demasiado largo se han dejado de definir alguna restricción, concretamente la reflexiva de VENDEDOR a VENDEDOR (que marca la cabeza)

3.2.5 DROP TABLE

Nos servirá para borrar absolutamente una mesa, tanto los datos como la estructura. Se debe tener cuidado con ella, porque es una operación que no se puede deshacer, y por lo tanto potencialmente muy peligrosa.

sintaxis

```
DROP TABLE tabla
```

ejemplos

```
DROP TABLE FAMILIAR
```

3.2.6 Índice

Los índices son estructuras de datos que permiten mantener ordenada una mesa respecto a uno o más de un campo, cada uno de ellos de forma ascendente o descendente.

Tener un índice para un determinado campo o campos permite reducir drásticamente el tiempo utilizado en ordenar por ellos (porque ya se mantiene este orden) y también cuando se busca un determinado valor de este campo, ya que como está ordenado se pueden hacer búsquedas binarias o dicotómicas. Cuando no está ordenado no hay más remedio que hacer una búsqueda secuencial, que es considerablemente más lenta.

De todos modos no se debe abusar de los índices, ya que es una estructura adicional de datos que ocupará espacio, y que como se han de mantener los índices constantemente actualizados, cada vez que se realiza una operación de actualización (inserción , modificación o borrado) se ha de reestructurar el índice, para poner o quitar el elemento en su lugar.

El siguiente vídeo intenta explicar de forma muy sencilla como se podría implementar un índice, para poder observar las características anteriores. Debemos ser conscientes, sin embargo, que todo este proceso es transparente para el usuario, que sólo notaría, al crear un índice, que los SELECT van más rápidos, y los INSERT, DELETE y UPDATE van más lentos.

Creación de índice

Aparte de la creación explícita de índice que haremos en esta pregunta, PostgreSQL crea implícitamente un índice cada vez que:

- Creamos una llave princpal
- Creamos una restricción de unicidad (UNIQUE)

En ambos casos será un índice que no se podrá repetir. Por tanto, en los casos anteriores no es necesario crear un índice, porque PostgreSQL ya lo ha hecho automáticamente.

En PostgreSQL la creación de índice es muy completa. Vamos a ver una versión resumida:

```
CREATE [UNIQUE] INDEX nom_índex
    ON mesa (c1 [ASC | DESC] [, c2 [ASC | DESC], ...] [nula {FIRST | LAST}])
```

Si ponemos la opción UNIQUE impedirá que se repitan los valores del campo (o campos) que forman el índice, de forma similar a la restricción UNIQUE del CONSTRAINT.

La opción de ordenación por defecto es el ascendente.

Podemos hacer que los nulos estén al principio de todo o al final de todo, según nos convenga:

- **FIRST** : hará que en la ordenación los valores nulos vayan antes de cualquier otro valor. Esta es la opción por defecto si la orden es descendente además de crear el índice hace que sea clave principal. Evidentemente no debe haber una clave principal creada con anterioridad.
- **LAST** : los valores nulos estarán al final de todo, después de cualquier otro valor. Es la opción por defecto cuando la orden es ascendente.

ejemplos

1. Crear un índice en la tabla **EMPLEAT4** para el campo **departamento** en orden ascendente.

```
CREATE INDEX i_dep ON EMPLEAT4 (departamento);
```

2. Crear un índice para el campo **fecha_nacimiento** (descendente) y **sueldo** (ascendente) en la tabla **EMPLEAT4** . En ambos casos, **fecha_nacimiento** y **sueldo** , se ordenarán los valores nulos al final

```
CREATE INDEX i_dat_sou ON EMPLEAT4 (fecha_nacimiento DESC nula LAST, sueldo nula
LAST);
```

Borrar un índice

La sentencia de borrar un índice es muy sencilla. Sólo tenemos que especificar el nombre del índice y la mesa donde está definido.

```
DROP INDEX nom_índex ON tabla
```

Ejercicios apartado 3.2.6

en **FACTURA_III**

6.90 Añadir un índice llamado **i_nom_cli** en la tabla **CLIENTE** por el campo **nombre** .

6.91 Añadir un índice llamado **i_adr_ven** en la tabla **VENDEDOR** para que esté ordenado por **cp** (ascendente) y **direccion** (descendente).

3.2.7 Vistas

Las vistas, también llamadas esquemas externos, consisten en visiones particulares de la BD Se corresponde al nivel externo de la arquitectura a tres niveles. Las tablas, que son las que realmente contienen los datos y dan la visión global de la BD, corresponden al nivel conceptual.

La sintaxis básica es la siguiente:

```
CREATE VIEW nom_vista
    AS subconsulta
    [WITH READ ONLY];
```

donde se le da un nombre, es el resultado de una subconsulta (un SELECT), y tenemos la posibilidad de impedir la modificación de los datos.

Por ejemplo, una vista con las comarcas, el número de poblaciones de cada comarca, el total de habitats y la altura media:

```
CREATE OR REPLACE VIEW ESTADISTICA AS
    SELECT nom_c, count (nombre) AS num_p, sum (población) AS pobl, avg
    (altura) AS alt_mitjana
        FROM POBLACIONES
        GROUP BY nom_c
        ORDER BY nom_c;
```

Nota

Si desea probar esta vista, puede hacerlo en la Base de Datos **geo** , ya que en ella tenemos datos para la consulta que proporciona los datos a la vista. Recuerde que nos estamos conectando todos como el mismo usuario, por lo tanto si alguien ha creado un objeto (en este caso una vista) no se podrá utilizar este nombre para crear otro objeto. Por eso es conveniente que al hacer una prueba de crear un objeto, luego elimine este objeto.

Observe la sintaxis CREATE OR REPLACE, que va muy bien para crear, y si ya existe para sustituir. Como estamos accediendo todos como el mismo usuario, será bastante normal que el objeto ya exista por haberlo creado un compañero. De esta manera el sustituiremos. No pasa nada por borrar un objeto ya existente, ya que todo esto son pruebas.

La manera de utilizarla es como una mesa normal, pero ya sabemos, que en realidad los datos están en las tablas.

```
SELECT *
    FROM ESTADISTICA;
```

E incluso se puede jugar con tablas y vistas. En esta sentencia sacaremos la provincia, nombre de la comarca, número de pueblos y altura media de las comarcas con 5 o menos pueblos:

```
SELECT provincia, COMARQUES.nom_c, num_p, alt_mitjana
    FROM COMARCAS, ESTADISTICA
    WHERE COMARQUES.nom_c = ESTADISTICA.nom_c and num_p <= 5;
```

Para borrar una vista

```
DROP VIEW nom_vista
```

Por ejemplo, para borrar la vista anterior: borrar una vista

```
DROP VIEW ESTADISTICA;
```

Ejercicios del apartado 3.2.7

en **FACTURA_III**

6.92 Crear la vista **RESUM_FACTURA** , que nos doy el total del dinero de la factura, el total después del descuento de artículos, y el total después del descuento de la factura, tal y como teníamos en la consulta **06:56** . A partir de ese momento podremos utilizar la vista para sacar estos resultados

3.2.8. Creación de otros objetos: secuencias, dominios y tipos.

Una de las características de PostgreSQL es su gran versatilidad.

En concreto se pueden crear muchos objetos. Además de los habituales, vistas, secuencias, ... se pueden xcrear más tipos de objetos. En este curso daremos una ulladeta los dominios y la definición de nuevos tipos de datos.

Debemos tener en consideración que todos nos conectaremos como el mismo usuario para hacer pruebas. Por lo tanto los objetos que creamos pueden fastidiar a otros compañeros si utilizamos todos los mismos nombres.

En otros SGBD (como por ejemplo Oracle) existe la posibilidad de crear los objetos poniendo **CREATE OR REPLACE ...** , que si no existe lo crea, y si existe lo sustituye. Lamentablemente en PostgreSQL depende de la versión: en las más modernas sí se puede, pero en versiones anteriores (como la 8.4 que es la que tenemos ahora en el servidor) no podremos, excepto en las vistas y las funciones.

3.2.8.1 Secuencias

También se pueden crear secuencias (**SEQUENCE)** , que son objetos que toman valores numéricos que van incrementándose (como el autonumérico).

```
CREATE SEQUENCE nom_seqüència
    [START WITH valor_inicial ]
    [INCREMENTED BY valor_increment ] ...;
```

La sentencia es más larga, para considerar más casos. Para nosotros está bien así.

Por defecto, el valor inicial es 1, y el incremento también 1.

Es un objeto independiente de las tablas. Se utiliza de la siguiente manera:

CURRVAL (' *nom_seqüència* ') devuelve el valor actual (debe estar inicializado)

NEXTVAL (' *nom_seqüència* ') incrementa la secuencia y devuelve el nuevo valor (excepto la primera vez que lo inicia con el valor inicial)

La manera habitual de utilizarlo será para obtener un valor que se añadirá a un campo de una tabla (normalmente la clave principal).

Supongamos que tenemos una tabla llamada **FACTURA** con la estructura que viene a continuación, y queremos que la clave principal sea un autonumérico. Lo podemos probar sobre la Base de Datos y usuario **pruebas** .

```
CREATE TABLE FACTURA
    (num_f NUMERIC (7) CONSTRAINT cp_fact PRIMARY KEY,
    fecha DATE,
    cliente VARCHAR (10));
```

Primero nos creamos la secuencia:

```
CREATE SEQUENCE s_num_fac START WITH 2016001;
```

Después lo utilizamos:

```
INSERT INTO FACTURA VALUES (NEXTVAL ( 's_num_fac'), '15 -01-2016 ',' cli001 ');
```

Si ahora miramos el contenido de la tabla **FACTURA** obtendremos:

	num_f [PK] nume	data date	client character
1	2016001	2016-01-15	cli001
*			

O incluso en el momento de declarar la mesa, le ponemos un valor predeterminado en el campo, que será el siguiente de la secuencia. No deberemos introducir ahora nada en la columna **num_f** , que toma el valor de la secuencia

```
CREATE TABLE FACTURA2
    (num_f NUMERIC (7) CONSTRAINT cp_fact2 PRIMARY KEY DEFAULT NEXTVAL (
    's_num_fac'),
    fecha DATE,
    cliente VARCHAR (10));
```

```
INSERT INTO FACTURA2 (fecha, cliente) VALUES ('15 -01-2016 ',' cli001 ');
```

Obsérvese que en la tabla **FACTURA2** , el valor empieza por **2.016.002** , ya que el primer valor la habíamos utilizado en **FACTURA** . Por lo tanto el contenido de la tabla **FACTURA2** será:

	num_f [PK] nume	data date	client characte
1	2016002	2016-01-15	cli001
*			

También lo podríamos haber hecho declarando la clave de tipo **SERIAL** , que lo que hace es implementar una secuencia, y dar valores sucesivos para el campo donde está definida. Pero de esta manera la secuencia comienza siempre por 1

```
CREATE TABLE FACTURA3
    (num_f SERIAL CONSTRAINT cp_fact3 PRIMARY KEY,
    fecha DATE,
    cliente VARCHAR (10));
```

Y luego hacer la inserción de este modo:

```
INSERT INTO FACTURA3 (fecha, cliente) VALUES ('15 -01-2016 ',' cli001 ')
```

Como comentábamos, el valor introducido por la secuencia será **1** :

	num_f [PK] serial	data date	client character
1	1	2016-01-15	cli001
*			

Volveremos a ver este ejemplo en las consultas de actualización, concretamente el **INSERT** .

Para borrar una secuencia utilizaremos la sentencia **DROP SEQUENCE** :

```
DROP SEQUENCE s_num_fac
```

De momento esta sentencia nos daría error, ya que la mesa **FACTURA2** utiliza esta secuencia. Borraremos primero las 3 tablas, y luego la secuencia para no interferir con los compañeros / as.

```
DROP TABLE FACTURA, FACTURA2, FACTURA3;
```

```
DROP SEQUENCE s_num_fac;
```

3.2.8.2 Dominios

Los dominios son los conjuntos de valores que puede tomar un determinado campo. Habitualmente se pone sencillamente un tipo de datos. Pero el modelo relacional teórico es más restrictivo, y los dominios aún podrían ser subconjuntos de estos tipos de datos.

Normalmente estos subconjuntos se realizan por medio de los ***check*** , que permiten una regla de validación (una condición) para dar los datos como buenas (por ejemplo, un sueldo siempre es positivo, por lo tanto, además de hacer que sea numérico podríamos obligar a que fuera positivo).

PostgreSQL permite definir dominios, que serán de un determinado tipo base, de un tamaño determinado (opcional), con un valor por defecto (opcional) e incluso con una cláusula ***check*** . A partir de ese momento, uno o más de un campo los podremos definir de este dominio (con la ventaja de que cambiando el dominio cambiamos el tipo de todos los campos que lo tienen).

Vamos a ver algunos ejemplos que se pueden realizar sobre el usuario **pruebas** .

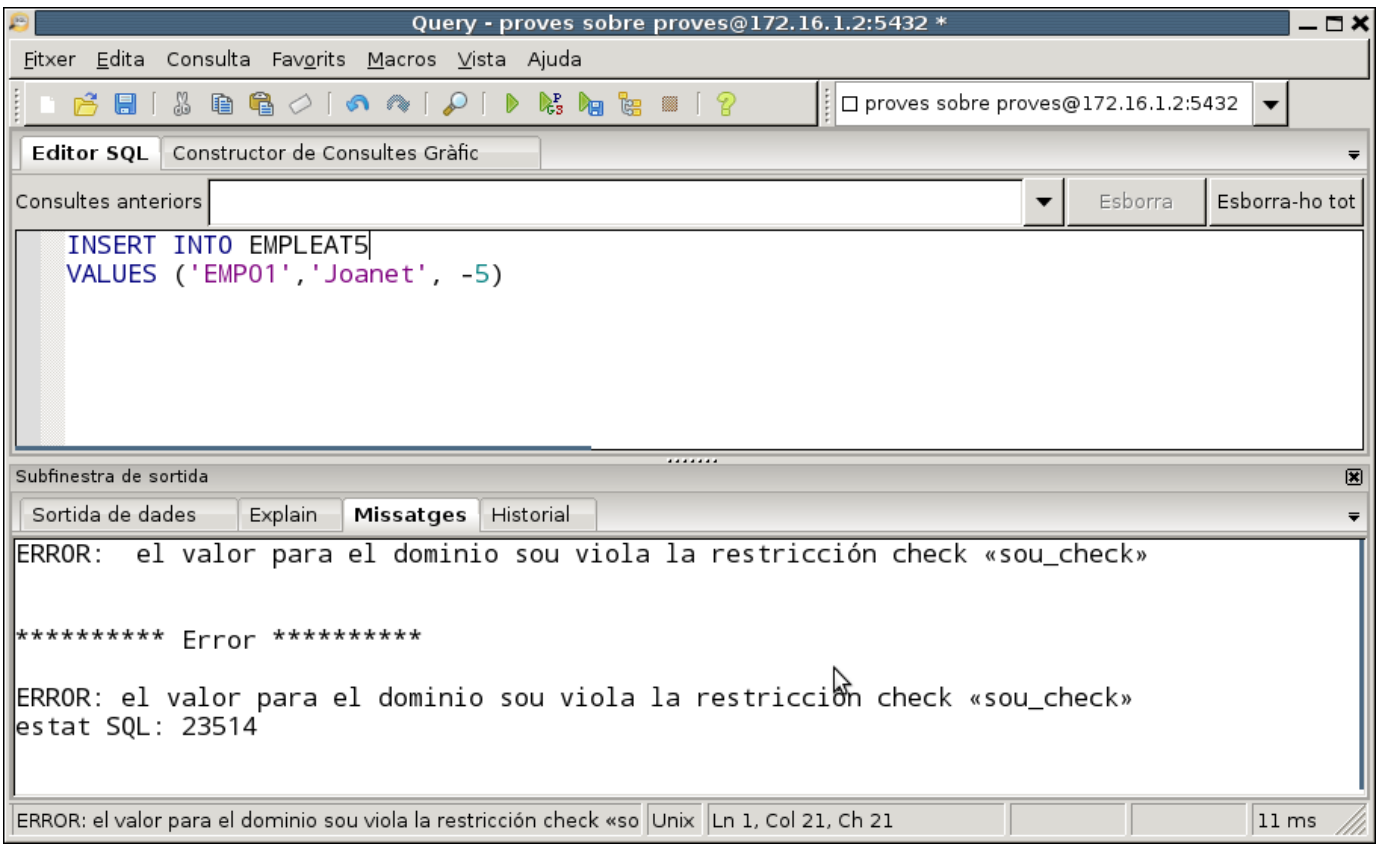
```
CREATE DOMAIN sueldo AS numeric (7,2)
    CHECK (VALUE> 0);
```

Por da error porque ya existía el dominio, sencillamente elimine el dominio anterior y lo intente

Ahora podríamos definir la tabla:

```
CREATE TABLE EMPLEAT5
    (cod_e varchar (5) primary key,
      nom_e varchar (50),
      salario sueldo);
```

Si intentamos introducir el dato del sueldo mal (por ejemplo poniéndolo negativo) veremos que da error.



Otro ejemplo, que haremos sobre la Base de Datos **pruebas** , aunque es un ejemplo basado en las tablas que teníamos en **geo** .

En la mesa de **POBLACIONES** tenemos las coordenadas (**latitud** y **longitud**). Podríamos intentar hacer unos dominios para marcar el **hemisferio** (NS para latitud, y EW para longitud), los **grados de latitud** (-90º a 90º), **grados de longitud** (-180º a 180º) en longitud, para los **minutos** (0 'a 59'), **según** (0 "a 59"). Centrémonos en la latitud:

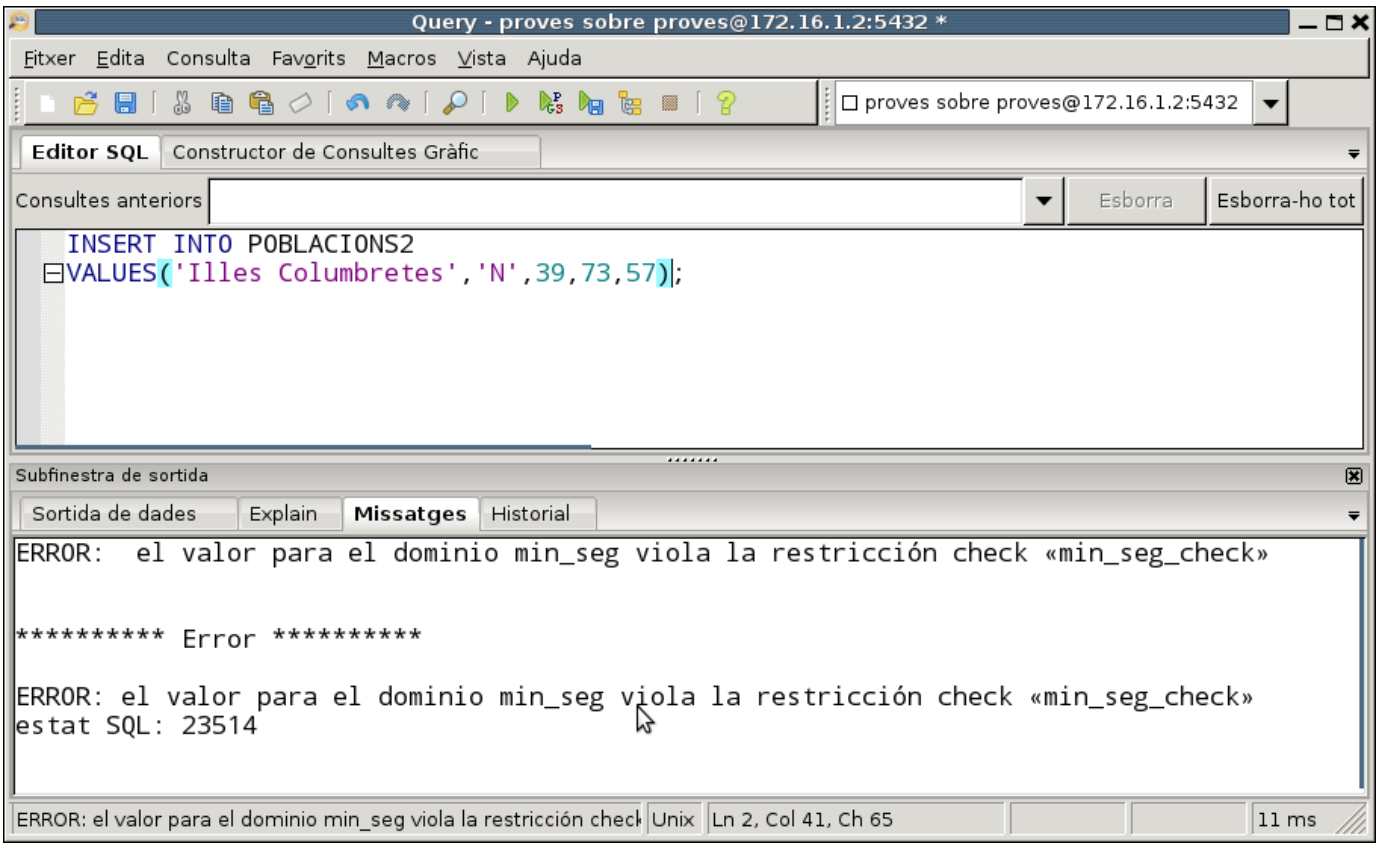
```
CREATE DOMAIN hemi_lat AS char (1)
    CHECK (VALUE IN ( 'N', 'S'));

CREATE DOMAIN graus_lat AS numérico (2)
    CHECK (VALUE BETWEEN 0 AND 90);

CREATE DOMAIN min_seg AS numérico (2)
    CHECK (VALUE BETWEEN 0 AND 59);
```

Ahora podríamos definir una alternativa a la mesa de POBLACIONES:

```
CREATE TABLE POBLACIONS2
    (nombre varchar (50) CONSTRAINT cp_pob2 PRIMARY KEY,
      lat_h hemi_lat,
      lat_g graus_lat,
      lat_m min_seg,
      lat_s min_seg);
```

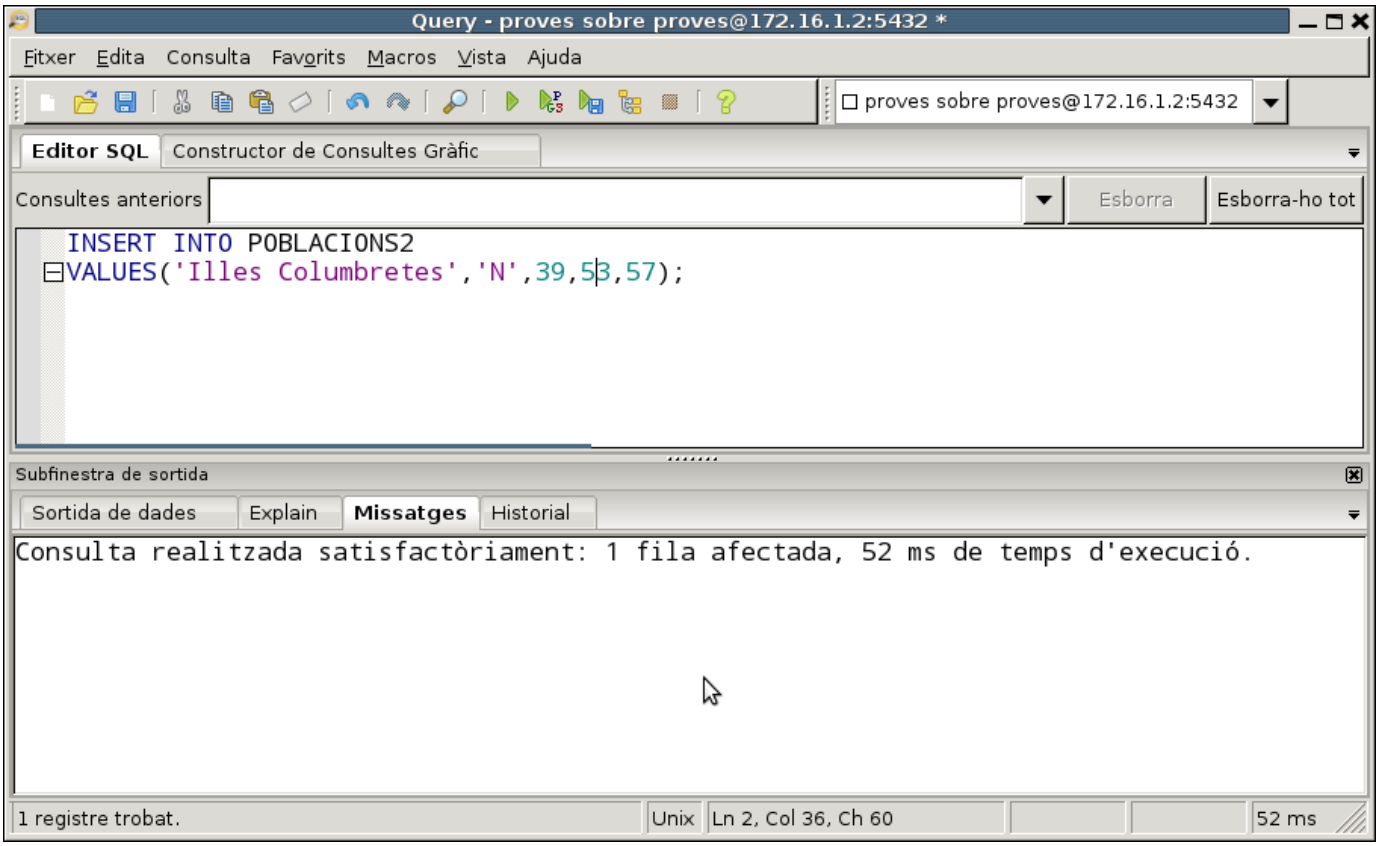
y comprobaríamos que las latitudes se han de introducir correctamente. Si ponemos por ejemplo los minutos mal, da error

```
INSERT INTO POBLACIONS2
    VALUES ( 'Islas Columbretes', 'N', 39, 73 , 57);
```



Pero no hay problema si los datos son correctos.

```
INSERT INTO POBLACIONS2
    VALUES ( 'Islas Columbretes', 'N', 39,53,57);
```



Para borrar un dominio, utilizaremos DROP DOMAIN.

```
DROP DOMAIN sueldo;
```

Si algún campo está definido con el dominio que borramos, dará error. Si borra con la opción CASCADE, borrarían los campos de las tablas con este dominio. Para dejar que los compañeros / as puedan trabajar también, borramos todo lo que hemos creado:

```
DROP TABLE EMPLEAT5;
```

```
DROP DOMAIN sueldo;
```

Aún no borramos los dominios **hemi_lat** , **graus_lat** y **min_seg** , porque los utilizaremos en la siguiente pregunta.

3.2.8.3 Tipos de datos

Ya habíamos comentado que PosgreSQL es muy versátil, y permite al usuario crear tipos de datos personales.

Tres son las maneras de crear un tipo de datos: compuesto, enumerado y externo.

- El **compuesto** es como un registro, donde se especifican los *campos* y los *tipos* .
- **Enumenat** : pondremos los posibles *valores* entre paréntesis.
- El **externo** es mucho más completo y complejo que permite crear un nuevo tipo base, con una estructura interna y con funciones (normalmente creadas en C) de entrada (para introducir el dato) y salida (para representarla), de análisis, ... este tipo escapa de los objetivos de este curso

compuesto

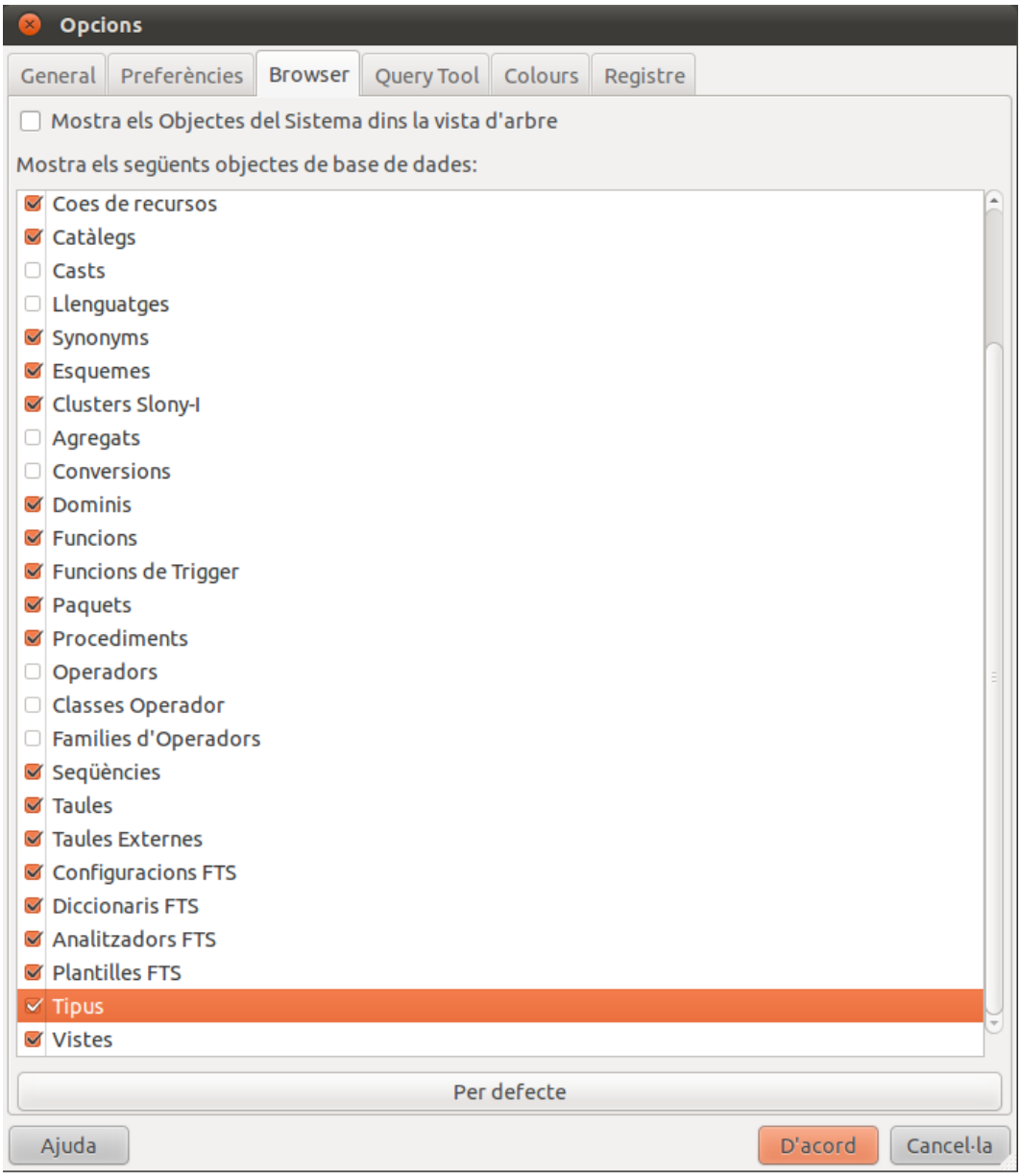
Pondremos entre paréntesis el nombre de los campos y el tipo que formarán parte de este tipo compuesto. Por ejemplo, sobre la Base de Datos **pruebas** :

```
CREATE TYPE num_complex AS (a float4, b float4);
```

que nos serviría para representar números complejos. Por da error porque ya existíamos el tipo (porque la ha creado un compañero / a), sencillamente elimine antes y lo intente

Nota

A partir de la versión 1.8 de **pgAdmin** , quizás por defecto no se visualizan todos las clases de objetos que tenemos definidos en nuestra Base de Datos. Por ejemplo, es fácil no poder ver los tipos de datos definidos por nosotros. Pero podemos ir al menú ***Archivo -> Opciones*** , y en la pestaña ***Browser*** elegir las clases de objetos que queremos visulizar:



Otro ejemplo (no hay que hacerlo, es ilustrativo solamente):

```
CREATE TYPE lat AS (
    h char (1),
    g numeric (2,0),
    m numeric (2,0),
    s numeric (2,0));
```

para representar la latitud, aunque podríamos aprovechar los dominios creados en el punto anterior, y nos saldrá mejor (este sí que lo puede hacer):

```
CREATE TYPE lat AS (
    h hemi_lat,
    g graus_lat,
    m min_seg,
    s min_seg);
```

y el tipo **lat** deberá respetar las restricciones de cada dominio de los cuatro campos.

Ahora podríamos definir una tabla

```
CREATE TABLE POBLACIONS3 (
    nombre varchar (50) CONSTRAINT cp_pob3 PRIMARY KEY,
```

```
    latitud lat,
    comarca varchar (50));
```

De esta manera queda muy compacto.

Por ejemplo, esta introducción de datos dará un error, ya que el hemisferio no es correcto:

```
INSERT INTO POBLACIONS3
    VALUES ( 'Castellón', '( K , 39,59,10)', 'Plana Alta');
```

En cambio esta funcionará perfectamente:

```
INSERT INTO POBLACIONS3
    VALUES ( 'Castellón', '(N, 39,59,10)', 'Plana Alta');
```

Para acceder a los campos de los tipos compuestos, bien ya definidos, bien definidos por nosotros, tendremos que poner el nombre de la columna seguida de un punto y seguido del nombre del campo. Pero en las sentencias SQL se nos presenta un problema: que esta sintaxis sirve para poner también el nombre de la tabla seguido del nombre de la columna. Así por ejemplo nos dará error la siguiente sentencia, en la que queremos sacar el nombre de la población, la latitud y si está en el hemisferio norte o sur:

```
SELECT nombre, latitud, latitud.h FROM POBLACIONS3;
```

El error lo da porque en **latitud.h** espera que latitud sea una mesa yh una columna. La manera de esquivar este error es poner entre paréntesis la columna:

```
SELECT nombre, latitud, (latitud) .h FROM POBLACIONS3;
```

En el siguiente tema, el de programación en **PL / pgSQL** veremos que allí no habrá que poner los paréntesis, porque no habrá la confusión de taula.columna

enumerado

El tipo enumerado es un conjunto de datos *estáticas* definidas en el momento de la creación del tipo, con un orden también predefinido. Está disponible desde la versión **8.3** de **PostgreSQL**.

Aquí tenemos un ejemplo:

```
CREATE TYPE d_set AS ENUM ( 'lunes', 'martes', 'miércoles', 'Jueves', 'viernes',
    'sábado', 'el domingo');
```

Posteriormente lo podremos utilizar, por ejemplo, en la definición de una tabla:

```
CREATE TABLE HORARIO (
    cod_pr NUMERIC (5) PRIMARY KEY,
    d_tut d_set,
    h_tut NUMERIC (4,2));
```

Ahora lo podemos utilizar. Tendremos que tener en cuenta que los valores son únicamente los definidos. Y se debe respetar mayúsculas y minúsculas.

```
INSERT INTO HORARIO VALUES (1, 'martes', 10);
```

En cambio, esta daría error:

```
INSERT INTO HORARIO VALUES (2, 'Miércoles', 12);
```

ya que el nombre de la semana comienza en mayúscula.

Introducimos algunos datos más

```
INSERT INTO HORARIO VALUES (2, 'miércoles', 12), (3, 'lunes', 9), (4, 'viernes', 11);
```

Podemos comprobar cómo el orden Predef es el del momento de la creación del tipo:

```
SELECT * FROM HORARIO ORDER BY d_tut;
```

Y como se puede comprobar en esta sentencia:

```
SELECT * FROM HORARIO WHERE d_tut> 'martes';
```

donde sólo saldrán las filas correspondientes al miércoles y el viernes (filas 2 y 4).

Por último, vamos a borrar los objetos que hemos creado, para no interferir con los compañeros:

```
DROP TABLE POBLACIONS2, POBLACIONS3, HORARIO;
```

```
DROP TYPE num_complex, lat, d_set;
```

```
DROP DOMAIN hemi_lat, graus_lat, min_seg;
```

3.3 Consultas de actualización

La consulta SELECT nos permitía consultar la información ya introducida. Las sentencias de DDL nos permiten crear (o modificar o borrar) las estructuras: tablas, índice, u otros objetos.

Nos falta, en SQL, aquellas sentencias que nos deben permitir alterar los datos. Únicamente son 3 sentencias, y ninguna de ellas modifica las estructuras; es decir, la estructura de las tablas quedará intancta, y sólo se modificará su contenido.

- **INSERT** , para introducir filas nuevas en la tabla.
- **DELETE** , para borrar filas enteras de la tabla.
- **UPDATE** , para modificar el contenido de una o más filas ya existentes.

3.3.1 INSERT

Servirá para introducir nuevas filas en una determinada mesa. Hay dos variantes de esta sentencia. La primera servirá para introducir nuevas filas proporcionándole los datos, es decir, indicando expresamente los nuevos valores de los campos. La otra podría servir para introducir nuevas filas de forma más masiva, tomando los datos de las tablas ya existentes para medio de una sentencia SELECT.

Sintaxis para inserción con valores

```
INSERT INTO tabla [(campo1 [, camp2 [...]])]
    VALUES (valor1 [, valor2 [...]]) [, (...)]
```

Como se puede comprobar de esta manera se introducirá una fila nueva, y se proporcionan los datos directamente, como constantes. Opcionalmente podemos introducir los valores para una segunda fila o cuantas se quieran, siempre poniendo los datos de cada fila entre paréntesis, y si hay más de una fila, separados por comas.

Es opcional poner la lista de campos de la tabla. Si no se ponen, se deberá poner un valor para cada campo de la tabla, y en el mismo orden como están definidos en la tabla.

Si ponemos expresamente los campos de la tabla, los podremos poner con el orden que queramos, y no será necesario ponerlos todos. Los campos no especificados quedarán con el valor nulo (a no ser que tengan un valor predeterminado, un valor por defecto). Por lo tanto, tendremos que poner obligatoriamente todos los campos definidos como no nulos (incluida la clave principal).

Para poner el valor nulo a un campo pondremos explícitamente **NULL** .

Si tenemos definido un campo autonumérico (SERIAL) y queremos continuar con el siguiente de la secuencia, no le tenemos que poner ningún valor. Luego en la sentencia SQL no deberá constar el campo: tendremos que poner la lista de campos, y en este no debe estar el campo autonumérico. Veremos después un ejemplo para verlo más claro.

Por el contrario, podemos romper la secuencia poniéndole un valor en el campo autonumérico, pero esto no tendrá modificado la secuencia. Si queremos modificarla deberíamos hacerlo con la sentencia **ALTER SEQUENCE** .

ejemplos

1. Introducir un departamento nuevo con los siguientes datos: Número: **6** ; Nombre: **Personal** ; Director: **18922222** y Fecha: **05/01/99** . La tabla **DEPARTAMENTO** la creamos en la pregunta **3.2.3** (Restricciones)

   ```
   INSERT INTO DEPARTAMENTO
       VALUES (6, 'Personal', '18.922.222', '5.1.99');
   ```

2. Introducir dos departamentos nuevos, este vez sin poner la fecha: Número: **7** ; Nombre: **Ventas** ; Director: **18.876.543** . Y Número: **8** ; Nombre: **Internacional** ; Director: **18999999** . Podemos poner además el orden que queramos:

   ```
   INSERT INTO DEPARTAMENTO (num_d, director, nom_d)
       VALUES (7, '18876543', 'Ventas'), (8, '18999999', 'Internacional');
   ```

   También lo podíamos haber hecho así. Observe que ahora el orden debe ser exactamente el de la definición de la tabla.

   ```
   INSERT INTO DEPARTAMENTO
       VALUES (7, 'Ventas', '18876543', NULL), (8, 'Internacional', '18999999', NULL);
   ```

3. Introducir 3 empleados en la tabla **EMPLEAT1** , con los valores siguientes (la tabla **EMPLEAT1** la creamos en la primera sentencia de ejemplo de la pregunta 3.2.2, CREATE TABLE):

    - Dni: **11111111** ; Nombre: **Albert**
    - Dni: **22222222b** ; Nombre: **Berta**
    - Dni: **33333333c** ; Nombre: **Claudia**

```
INSERT INTO EMPLEAT1
    VALUES ( '11111111', 'Albert'), ( '22222222b', 'Berta'), ( '33333333c',
    'Claudia')
```

4. Introducir 3 empleados en la tabla EMPLEAT2, con los valores siguientes (la tabla **EMPLEAT2** la creamos en la segunda sentencia de ejemplo de la pregunta 3.2.2, CREATE TABLE):

    - Dni: **44444444d** ; Nombre: **David** ; Departamento: **6** ; Sueldo: **1000**
    - Dni: **55555555e** ; Nombre: **Elena** ; Departamento: **6** ; Sueldo: **1500**
    - Dni: **66666666f** ; Nombre: **Ferran** ; Departamento: **7** ; Sueldo: **1750**

```
INSERT INTO EMPLEAT2 (dni, nombre, departamento, sueldo)
    VALUES ( ' 44444444d ', ' David ', 6,1000), ( ' 55555555e ', ' Elena ', 6,1500),
    ( ' 66666666f ', ' Ferran ', 7,1750)
```

5. Supongamos que tenemos una tabla de **FACTURAS** en la BD **pruebas** , con una clave principal que es un **autonumérico** (**SERIAL**). Como queremos practicar únicamente el autonumérico, la crearemos sencillita, con la siguiente estructura:

```
CREATE TABLE FACTURAS
    (num_f SERIAL PRIMARY KEY,
      descripcion VARCHAR,
      importe NUMERIC);
```

    La manera de ir introduciendo normalmente será sin especificar num_f, para que vaya cogiendo valores consecutivos.

```
INSERT INTO FACTURAS (descripcion, importe)
    VALUES ( 'Factura 1', 54.23);
```

    Pero si queremos romper la secuencia, sí pondremos num_f.

```
INSERT INTO FACTURAS
    VALUES (2016001, 'Factura 1 de 2016', 23.98);
```

    Eso sí, si introducimos otra fila sin especificar el número de factura, lo cogerá de la secuencia, que no se había modificado, y por tanto el valor que nos dará será **2** .

```
INSERT INTO FACTURAS (descripcion, importe)
    VALUES ( 'Factura 2', 36.27);
```

    Para cambiar definitivamente el valor que debe ir autoincrementant a, deberíamos modificar la secuencia (**ALTER SEQUENCE**).

    Después de las 3 inserciones anteriores, el contenido de la tabla FACTURAS será este:

	num_f [PK] serial	descripcio character varying	import numeric
1	1	Factura 1	54.23
2	2	Factura 2	36.27
3	2016001	Factura 1 del 2016	23.98
*			

Sintaxis para inserción con subconsulta

```
INSERT INTO tabla [(campo1 [, camp2 [...]])]
    SELECT ...
```

Es decir, los datos que van a insertarse las sacamos a partir de una sentencia SELECT. La sentencia SELECT puede ser tan complicada como haga falta, y podemos poner las cláusulas necesarias: WHERE, GROUP BY, ... El requisito es que tendrá que volver tantos campos como tenga la mesa o como estén especificados en la sentencia INSERT, y que los datos que vuelven sean del mismo tipo (o tipos compatibles, aunque PostgreSQL puede hacer una conversión de tipo automática). Al igual que en el anterior formato, cuando no se especifique un campo se le asignará el valor nulo, o el valor predeterminado si tiene definido.

Se podría especificar incluso una tabla de una base de datos externa. Consulte la sentencia SELECT, el apartado "Especificación de una BD externa".

ejemplos

1. Insertar todos los registros de la tabla **EMPLEAT1** en **EMPLEAT3** , suponiendo que existen estas tablas y aunque tienen una estructura diferente, las dos tienen los campos **dni** y **nombre** . Si no existe la tabla **EMPLEAT3** , puedes volver a crearla, utilizando la última sentencia de la pregunta 3.2.3 (Restricciones)

```
INSERT INTO EMPLEAT3 (dni, nombre)
    SELECT dni, nombre FROM EMPLEAT1;
```

2. Insertar en la tabla **EMPLEAT3** todos los empleados de la tabla **EMPLEAT2** los departamentos 6 y 7.

```
INSERT INTO EMPLEAT3
    SELECT * FROM EMPLEAT2
        WHERE departamento IN (6,7);
```

3. Insertar en la tabla EMPLEAT3 los empleados que tienen más de un familiar. Esta sentencia no la podremos ejecutar, ya que no tenemos la tabla **FAMILIAR** . Está únicamente de manera ilustrativa.

```
INSERT INTO EMPLEAT3
    SELECT * FROM EMPLEADO
        WHERE dni IN (SELECT dni
                      FROM FAMILIAR
                      GROUP BY dni
                      HAVING COUNT (*)> 1);
```

Ejercicios apartado 3.3.1

en **FACTURA_III**

6.93 Insertar en la tabla **CATEGORÍA** las siguientes filas:

cod_cat	descripcion
BjcOlimpia	Componentes Bjc Sería Olimpia
Legrand	Componentes marca Legrand
IntMagn	interruptor magnetotérmico
Niessen	Componentes Niesen Serie Lisa

6.94 Insertar los siguientes artículos.

cod_art	descrip	precio	stock	stock_min	cod_cat
B10028B	Cruzamiento Bjc Serie Olimpia	04:38	2	1	BjcOlimpia
B10200B	Cruzamiento Bjc Olimpia Con Visor	0.88	29		BjcOlimpia
L16550	Cartucho Fusible Legrand T2 250 A	5.89	1	1	Legrand
L16555	Cartucho Fusible Legrand T2 315 A	5.89	3	3	Legrand
IM2P10L	Interruptor magnetotérmico 2p, 4	14.84	2	1	IntMagn
N8008BA	Base Tt Lateral Niessen Trazo Bla	04:38	6	6	Niessen

6.95 Insertar en la tabla **CLIENTE** tres filas con los siguientes datos

cod_cli	nombre	direccion	cp	cod_pob
303	MIRAVET SALA, MARIA MERCEDES	URBANIZACION EL BALCO, 84-11		
306	SAMPEDRO SIMO, MARIA MERCEDES	Finelli, 161	12217	
387	TUR MARTIN, MANUEL FRANCISCO	CALLE PEDRO VIRUELA, 108-8	12008	

6.96 Insertar la siguiente factura:

num_f	fecha	cod_cli	cod_ven	iva	dto
6535	2015-01-01	306		21	10

num_f	num_l	cod_art	cuanto	precio	dto
6535	1	L16555	2	5.89	25

6.97 Insertar la siguiente factura (esta tiene más de una línea de factura).

num_f	fecha	cod_cli	cod_ven	iva	dto
6559	2015-02-16	387		10	10

num_f	num_l	cod_art	cuanto	precio	dto
6559	1	IM2P10L	3	14.84	
6559	2	N8008BA	6	04:38	20

3.3.2 DELETE

Esta sentencia servirá para **borrar filas enteras** de una determinada mesa.

Si se quieren borrar sólo algunos campos de una o unas determinadas filas, no se debe utilizar esta sentencia, sino que se deberá utilizar la sentencia **UPDATE** para poner los campos a NULL.

sintaxis

```
DELETE FROM tabla
    [WHERE condición]
```

Se borrarán las filas que cumplan la condición. Si no se pone el WHERE borrarán tetas las filas de la tabla (pero no la misma mesa, la estructura, es decir, seguirá existiendo la mesa, pero vacía).

Por la peligrosidad de la sentencia, es conveniente hacer primero una sentencia SELECT, y cuando estemos seguros de que se seleccionan sólo las filas que queremos borrar, cambiarla por DELETE. También es muy conveniente hacer copias de seguridad de las tablas o de toda la Base de Datos.

ejemplos

1. Borrar todas las filas de **EMPLEAT2**

```
DELETE FROM EMPLEAT2;
```

2. Borrar de **EMPLEAT3** los empleados del departamento 7

```
DELETE FROM EMPLEAT3
    WHERE departamento = 7;
```

3. Borrar de **EMPLEAT3** los empleados mayores de 47 años (en realidad no se borrará ninguna porque no tenemos introducida la fecha de nacimiento)

```
DELETE FROM EMPLEAT3
    WHERE EXTRACT (year FROM AGE (CURRENT_DATE, fecha_nacimiento))> = 47;
```

4. Borrar de EMPLEADO aquellos empleados que Trebal en proyectos menos de 20 horas (no se muy conveniente confirmar la eliminación de los registros ya que nos quedaríamos sin datos importantes)

```
DELETE FROM EMPLEADO
    WHERE dni IN (SELECT dni
                  FROM TRABAJA
                  GROUP BY dni
                  HAVING SUM (s) <20);
```

Ejercicios apartado 3.3.2

En **FACTURA_999x** (donde 999 son las 3 últimas cifras de su DNI, sale la letra del NIF)

6.98 Borrar la factura **6559** . Comprobar que se han borrado sus líneas de factura

6.99 Borrar los artículos de los que **no** tenemos **stock mínimo** .

3.3.3 UPDATE

Esta sentencia servirá para modificar alguno o algunos campos de determinadas filas de una tabla.

En concreto también servirá para "borrar" el contenido de algún campo de alguna fila. Lo que haremos será poner a NULL este campo.

sintaxis

```
UPDATE tabla
    SET campo1 = valor1 [, camp2 = valor2 [...]]
    [WHERE condición];
```

Pondremos por tanto, después de especificar la tabla, el campo (o campos) que queremos cambiar seguido del nuevo valor. En caso de querer cambiar más de un campo, irán separados por comas.

Sólo cambiarán los valores de las filas que cumplan la condición. Si no se pone condición se modificarán todas las filas.

Haremos la misma consideración de peligrosidad que en el DELETE: es conveniente hacer primero una sentencia SELECT, y cuando estemos seguros de que se seleccionan sólo las filas que queremos modificar, cambiarla por UPDATE con la modificación de los valores. También es muy conveniente hacer copias de seguridad de las tablas o de toda la Base de Datos.

ejemplos

1. Borrar todas las fechas de incorporación de la mesa EMPLEAT3

```
UPDATE EMPLEAT3
    SET data_incorporacio = NULL;
```

2. Aumentar el sueldo un 5% a los empleados del departamento 6 de la tabla EMPLEAT3.

```
UPDATE EMPLEAT3
    SET sueldo = sueldo * 01:05
    WHERE departamento = 6;
```

3. Cambiar la población a Ares y el departamento al 8, a todos los empleados de Castellón de la tabla EMPLEAT3

```
UPDATE EMPLEAT3
    SET departamento = 8,
        poblacion = Ares '
    WHERE población = 'Castellón';
```

Ejercicios apartado 3.3.3

en **FACTURA_III**

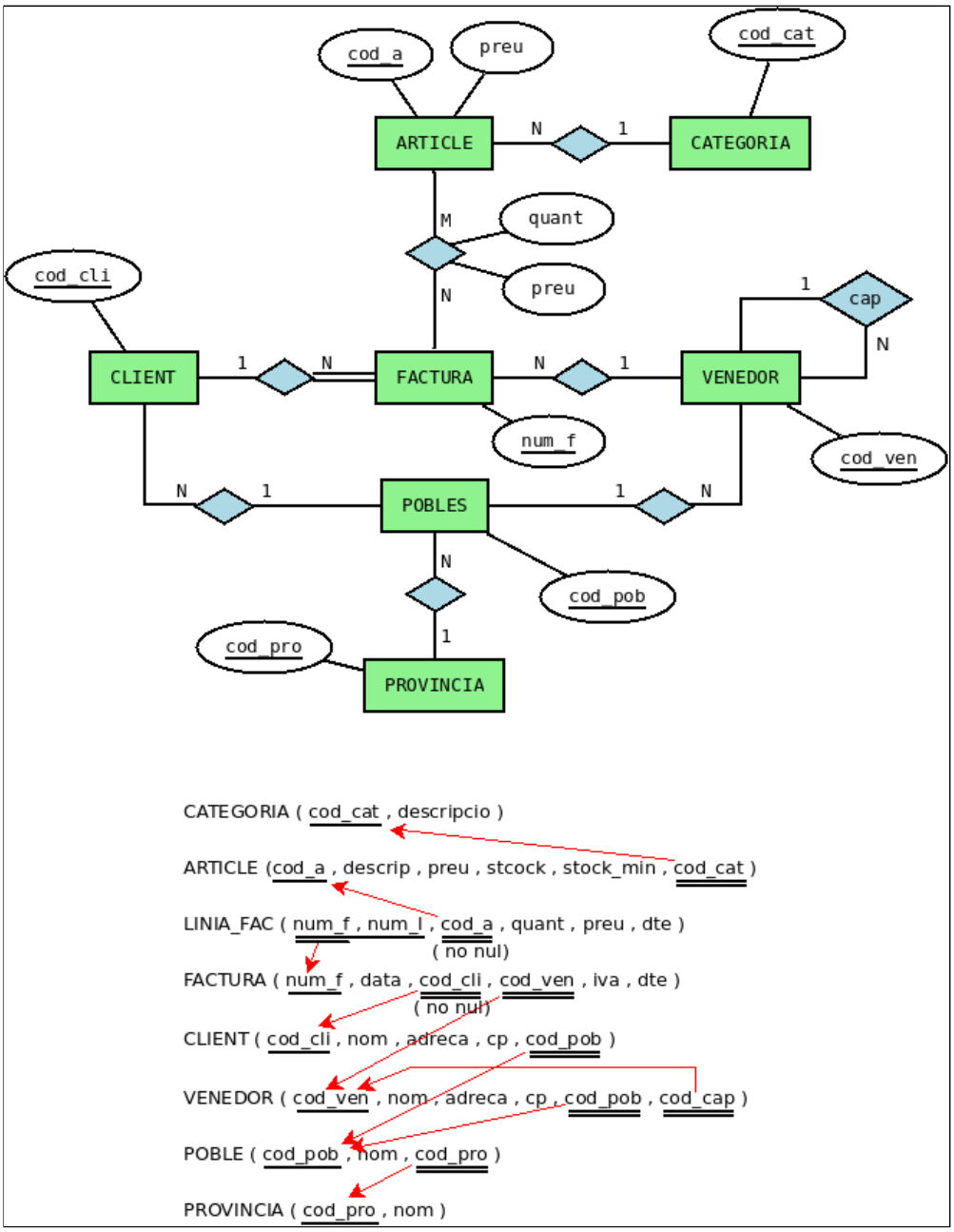
6.100 Quitar todos los códigos postales de los clientes.

6.101 Subir el precio de los artículos de la categoría **BjcOlimpia** un **5%** (el resultado será que el único artículo de esta categoría habrá pasado de un precio de 4:38 a **4.60 €**)

Ejercicios de todo el tema

A lo largo de esta tercera parte, en el conjunto de ejercicios de DDL, crearemos toda la estructura de la Base de Datos **FACTURA** y lo haremos sobre la BD que hemos creado: **Factura_III**

El esquema Entidad-Relación y el esquema relacional que implementaremos será el siguiente:



En la Base de Datos llamada **FACTURA_III:**

Nota

Durante todos estos ejercicios de DDL puede ser muy conveniente tener abiertas las dos conexiones: la de **FACTURA** (para ir consultando) y la de **FACTURA_III** (para ir creando y modificando)

6.77 Crea la tabla **CATEGORÍA** , con los mismos campos y del mismo tipo que en la tabla CATEGORÍA de **FACTURA** , pero de momento sin clave principal ni ninguna otra restricción. Guarde la consulta de creación como **Ex_6_77.sql**

6.78 Crea la tabla **ARTÍCULO** , también sin restricciones. Guardar la consulta como **Ex_6_78.sql**

6.79 Crear la tabla **PROVINCIA** , con la clave principal.

6.80 Crear la tabla **PUEBLO** , con la clave principal y la restricción que el campo **cod_pro** es clave externa que apunta a PROVINCIA.

6.81 Crear la tabla **VENDEDOR** , con la clave principal y la clave externa a PUEBLO (de momento no definimos la clave externa a VENDEDOR, que es reflexiva).

6.82 Crear la tabla **CLIENTE** , con la clave principal y la clave externa a PUEBLO

6.83 Crear la tabla **FACTURA** , con la llave principal y las claves externas a CLIENTE y VENDEDOR. También debe exigir que **cod_cli** sea no nulo.

6.84 Crear la tabla **LINIA_FAC** , con la clave principal (observa que está formada por 2 campos) pero de momento sin la clave externa que apunta a ARTÍCULO. Además **cod_a** debe ser no nulo.

6.85 Añadir un campo a la mesa **VENDEDOR** llamado **alias** de tipo texto, que debe ser no nulo y único.

6.86 Borrar el campo anterior, **alias** , de la tabla **VENDEDOR** .

6.87 Añadir la clave principal de **CATEGORÍA** .

6.88 En la tabla **ARTÍCULO** añadir la clave principal y la clave externa a CATEGORÍA.

6.89 En la tabla **LINIA_FAC** añadir la clave externa que apunta a FACTURA, **exigiendo que se borre en cascada** (si se borra una factura, se borrarán automáticamente sus líneas de factura). Y también la clave externa que apunta a ARTÍCULO (esta normal, es decir NO ACTION)

6.90 Añadir un índice llamado **i_nom_cli** en la tabla **CLIENTE** por el campo **nombre** .

6.91 Añadir un índice llamado **i_adr_ven** en la tabla **VENDEDOR** para que esté ordenado por **cp** (ascendente) y **direccion** (descendente).

6.92 Crear la vista **RESUM_FACTURA** , que nos doy el total del dinero de la factura, el total después del descuento de artículos, y el total después del descuento de la factura, tal y como teníamos en la consulta **06:56** . A partir de ese momento podremos utilizar la vista para sacar estos resultados

6.93 Insertar en la tabla **CATEGORÍA** las siguientes filas:

cod_cat	descripcion
BjcOlimpia	Componentes Bjc Sería Olimpia
Legrand	Componentes marca Legrand
IntMagn	interruptor magnetotérmico
Niessen	Componentes Niesen Serie Lisa

6.94 Insertar los siguientes artículos.

cod_art	descrip	precio	stock	stock_min	cod_cat
B10028B	Cruzamiento Bjc Serie Olimpia	04:38	2	1	BjcOlimpia
B10200B	Cruzamiento Bjc Olimpia Con Visor	0.88	29		BjcOlimpia
L16550	Cartucho Fusible Legrand T2 250 A	5.89	1	1	Legrand
L16555	Cartucho Fusible Legrand T2 315 A	5.89	3	3	Legrand
IM2P10L	Interruptor magnetotérmico 2p, 4	14.84	2	1	IntMagn
N8008BA	Base Tt Lateral Niessen Trazo Bla	04:38	6	6	Niessen

6.95 Insertar en la tabla **CLIENTE** tres filas con los siguientes datos

cod_cli	nombre	direccion	cp	cod_pob
303	MIRAVET SALA, MARIA MERCEDES	URBANIZACION EL BALCO, 84-11		
306	SAMPEDRO SIMO, MARIA MERCEDES	Finelli, 161	12217	

387	TUR MARTIN, MANUEL FRANCISCO	CALLE PEDRO VIRUELA, 108-8	12008	

6.96 Insertar la siguiente factura:

num_f	fecha	cod_cli	cod_ven	iva	dto
6535	2015-01-01	306		21	10

num_f	num_l	cod_art	cuanto	precio	dto
6535	1	L16555	2	5.89	25

6.97 Insertar la siguiente factura (esta tiene más de una línea de factura).

num_f	fecha	cod_cli	cod_ven	iva	dto
6559	2015-02-16	387		10	10

num_f	num_l	cod_art	cuanto	precio	dto
6559	1	IM2P10L	3	14.84	
6559	2	N8008BA	6	04:38	20

6.98 Borrar la factura **6559** . Comprobar que se han borrado sus líneas de factura

6.99 Borrar los artículos de los que **no** tenemos **stock mínimo** .

6.100 Quitar todos los códigos postales de los clientes.

6.101 Subir el precio de los artículos de la categoría **BjcOlimpia** un **5%** (el resultado será que el único artículo de esta categoría habrá pasado de un precio de 4:38 a **4.60 €**)