Bases de Datos objeto-relacionales

Introducción a las Bases de Datos objeto-relacionales

Las Bases de Datos objeto-relacionales son una extensión de las Bases de Datos relacionales tradicionales, a las cuales se les añaden ciertas características propias de la Programación Orientada a Objetos. Algunas de estas características se comentarán en este Resultado de Aprendizaje , y concretamente para el motor de Bases de Datos objeto-relacional *PostgreSQL*.

Antes de continuar con las extensiones propias de las Bases de Datos objeto-relacionales, hay que destacar las diferencias existentes entre PostgreSQL y el motor MySQL con el que se ha venido trabajando todo el curso. Hay que tener en cuenta que, independientemente de que el nuevo motor sea objeto-relacional, existirán algunas diferencias de funcionamiento y de síntaxis a la hora de trabajar con el mismo.

Conectar con PostgreSQL

A la hora de conectar con *PostgreSQL*, como ocurría con *MySQL*, tenemos dos opciones:

- Conectar mediante su propia consola de comandos (lado cliente)
- Utilizar alguna herramienta gráfica (*PostgreSQL* incluye la suya propia, *pgAdmin*)

En este apartado veremos como manejarnos con la primera de las opciones, con la consola de comandos de *PostgreSQL* de forma que al menos sepamos llevar a cabo las funciones básicas como conectarnos, acceder a una Base de Datos y realizar las operaciones más comunes sobre ella.

- Conectar con *PostgreSQL*: Ejecutaremos el comando **`psql`**

Herramienta de administración pgAdmin

Creación de tablas

Una vez conectados a una base de datos, la sentencia

create table persona (
nombre varchar(30),
direccion varchar(30)
);
El comando **drop table** permite eliminar tablas:
demo=# drop table persona;

Para ver la lista de las tablas se usará **\dt**

La tabla está lista para insertar en ella algunos registros.
demo=# insert into persona values (‘Alejandro Magno’ , ‘Babilonia’);
INSERT 24756 1

demo=# insert into persona values (‘Federico García Lorca ‘ , ‘Granada 65’);
INSERT 24757 1

El número con el que responde el comando insert se refiere al **OID** del registro insertado.

Las tablas creadas en PostgreSQL incluyen, por defecto, varias columnas ocultas que almacenan información acerca del identificador de transacción en que pueden estar implicadas, la localización física del registro dentro de la tabla (para localizarla muy rápidamente) y, los más importantes, **el OID y el TABLE OID**. Estas últimas columnas están definidas con un tipo de datos especial llamado identificador de objeto (OID) que se implementa como un entero positivo de 32

bits. Cuando se inserta un nuevo registro en una tabla se le asigna un número consecutivo como OID, y el TABLEOID de la tabla que le corresponde.

En la programación orientada a objetos, el concepto de OID es de vital importancia, ya que se refiere a la identidad propia del objeto, lo que lo diferencia de los demás objetos.

Para observar las columnas ocultas, debemos hacer referencia a ellas específicamente en el comando select:

demo=# select oid, tableoid, * from persona;

```
Oid |tableoid |nombre |direccion
-------+----------+----------------------+------------
17242 |17240 | Alejandro Magno | Babilonia
17243 |17240 | Federico García Lorca | Granada 65
(2 rows)
```

Estas columnas se implementan para servir de identificadores en la realización de enlaces desde otras tablas.

Ejemplo de la utilización de OID para enlazar dos tablas

Retomamos la tabla persona y construimos una nueva tabla para almacenar los teléfonos.

```
demo=# create table telefono (
demo(# tipo char(10),
demo(# numero varchar(16),
demo(# propietario oid
demo(# );
```

La tabla teléfono incluye la columna propietario de tipo OID, que almacenará la referencia a los registros de la tabla persona. Agreguemos dos teléfonos a ‘Alejandro Magno’, para ello utilizamos su OID que es 17242:

```
demo=# insert into telefono values( ‘móvil’ , ‘12345678’, 17242 );
demo=# insert into telefono values( ‘casa’ , ‘987654’, 17242 );
```

Las dos tablas están vinculadas por el OID de persona.

```
demo=# select * from telefono;
```

```
Tipo |numero | propietario
------------+-------------+-------------
Móvil | 12345678 | 17242
casa | 987654 | 17242
(2 rows)
```

La operación que nos permite unir las dos tablas es join, que en este caso une teléfono y persona, utilizando para ello la igualdad de las columnas telefono. propietario y persona.oid:

```
demo=# select * from telefono join persona on (telefono.propietario = persona.oid);
```

```
Tipo
|numero
| propietario |nombre
| direccion
------------+-------------+-------------+-----------------+-----------
móvil |
12345678 |
17242 | Alejandro Magno | Babilonia
casa |
987654 |
17242 | Alejandro Magno | Babilonia
(2rows)
```

Tipos de datos

Tipos de datos básicos

Independientemente de los nuevos tipos de datos que postgreSQL incorpora por ser un motor objeto-relacional, presenta algunas diferencias en cuanto a declarar los tipos tradicionales que ya conociamos en MySQL

- `VARCHAR` se declara también como **varchar** o **`text`** `sin ser` necesario indicar la longitud máxima
- `INTEGER` se declara de la misma forma, como **`integer`** o **`int`**
- `FLOAT` se declara como **`real`** y se utiliza para indicar números de coma flotante de precisión simple
- `DOUBLE` se declara como **`double precision`** para indicar números de coma flotante de doble precisión
- También muy común utilizar el tipo de datos **`numeric`** para almacenar cantidades de moneda o valores donde la precisión es importante.
- En el caso de campos autonuméricos se utiliza la palabra reservada **`serial`** sin indicar el tipo de dato, aunque internamente se almacena utilizando un campo *integer* de 4 bytes

Colecciones - Arrays

Los ***array***, como tipo de datos colección, puede ser usado en BBDD objeto-relacionales, creando así estructuras de datos más complejas tal y como se hace con lenguajes de programación de aplicaciones. Por ejemplo:

```
CREATE TABLE personas (
  id                serial PRIMARY KEY,
  nombre            text,
  apellidos         text,
  fecha_nacimiento TIMESTAMP,
  telefonos         text[]
);
```

Que a la hora de trabajar en SQL sería así (Presta atención a cómo debe asignarse el valor del array):

```
-- Inserta una persona con varios números de teléfono
INSERT INTO personas
  (nombre, apellidos, fecha_nacimiento, telefonos)
  VALUES ('Peter', 'Parker', '1999-01-08',
           '{"976654321", "654786556", "976123456"}');

-- Muestra todos los teléfonos de una persona
SELECT telefonos
FROM personas
WHERE nombre = 'Peter' AND apellidos = 'Parker';

---------------------------------
 {976654321,654786556,976123456}


-- Muestra el primer teléfono de una persona
SELECT telefonos[1]
FROM personas
WHERE nombre = 'Peter' AND apellidos = 'Parker';

-- Muestra desde el primero al segundo teléfono de una persona
SELECT telefonos[1:2]
FROM personas
WHERE nombre = 'Peter' AND apellidos = 'Parker';

-- Muestra todos los teléfonos de una persona
SELECT telefonos[1:array_length(telefonos, 1)]
FROM personas
WHERE nombre = 'Peter' AND apellidos = 'Parker';


-- Muestra el nombre y apellidos de las personas que
-- tienen un número determinado
SELECT nombre, apellidos
FROM personas
WHERE '97654321' = ANY(telefonos);
```

Además PostgreSQL proporciona una serie de funciones para operar sobre los arrays

Suponemos un vector compuesto por 3 ciudad españolas

ciudades = {'madrid', 'zaragoza', 'barcelona'}

array_length(unarray, dimension): Devuelve la longitud del array de la dimensión especificada

SELECT array_length(ciudades, 1)
> 3

array_cat(unarray, otroarray): Concatena ambos array y devuelve el resultado

SELECT array_cat(ciudades, ciudades)
>{'madrid', 'zaragoza', 'barcelona', 'madrid', 'zaragoza', 'barcelona'}

array_replace(unarray, valor_viejo, valor_nuevo): Reemplaza un valor por otro en todo el array

SELECT array_replace(ciudades, 'zaragoza', 'teruel')
>{'madrid', 'teruel', 'barcelona'}

array_to_string(unarray, texto): Convierte un vector en una cadena de texto

SELECT array_to_string(ciudades, ',')
> madrid,zaragoza,barcelona

string_to_array(texto, texto): Convierte un texto es un vector

SELECT string_to_array('madrid,zaragoza,barcelona', ',')
> {madrid,zaragoza,barcelona}

Datos estructurados

Con este tipo de datos, **Composite en PostgreSQL**, existe la posibilidad de crear estructuras de datos más complejas, compuestas de varios campos de información. Se utilizan para definir la estructura de una fila o un registro. Por ejemplo:

```
CREATE TYPE direccion_postal AS (
  calle     text,
  numero    INTEGER,
  provincia text,
  cp        text
);

CREATE TABLE personas (
  id               serial PRIMARY KEY,
  nombre           text,
  apellidos        text,
  fecha_nacimiento TIMESTAMP,
  telefonos        text[],
  direccion        direccion_postal
);
```

Así, en el caso de una inserción, sería asi:

```
INSERT INTO personas (nombre, apellidos, fecha_nacimiento, telefonos, direccion)
  VALUES ('Peter', 'Parker', '2000-01-01', '{"976654321", "654786556"}', ROW('mi calle', 23, 'Zaragoza',
  50018));
```

Que a la hora de consultar sería como sigue:

```
SELECT (direccion).cp
FROM personas
WHERE nombre = 'Peter' AND apellidos = 'Parker';
```

En el caso de necesitar un array de estructuras sería de la siguiente manera:

```
CREATE TYPE direccion_postal2 AS
  ( calle text,
  cp text
);
```

```
CREATE TABLE alumnos
  ( id serial PRIMARY
  KEY,
  nexp text NOT NULL UNIQUE,
  nombre text NOT NULL,
  apellidos text NOT NULL,
  telefono text,
  email text UNIQUE,
  direcciones direccion_postal2[],
  id_companero INT REFERENCES alumnos  (id)
);
```

Que a la hora de insertar datos quedaría:

```
INSERT INTO alumnos (nexp, nombre, apellidos, telefono, email, id_companero, direcciones)
    VALUES ('11111', 'nombre', 'apellidos', '37834764',
    'asda@asda.es', 1,
    array[ROW('calle', '4663'),
          ROW('calle2', '34334')]::direccion_postal2[]);
```

Enumeraciones

Las enumeraciones en PostgreSQL se definen como un nuevo tipo de datos compuesto de una lista fija de valores constantes.

```
CREATE TYPE colores AS ENUM ('rojo', 'amarillo', 'verde', 'negro', 'blanco');
```

Además PostgreSQL proporciona una serie de funciones para trabajar sobre este tipo de datos:

- **enum_first:** Devuelve el primer valor de una enumeración

```
SELECT enum_first(NULL::colores)
> rojo
```

- **enum_last:** Devuelve el último valor de una enumeración

```
SELECT enum_last(NULL::colores)
 > blanco
```

- **enum_range:** Devuelve el rango de valores entre dos datos

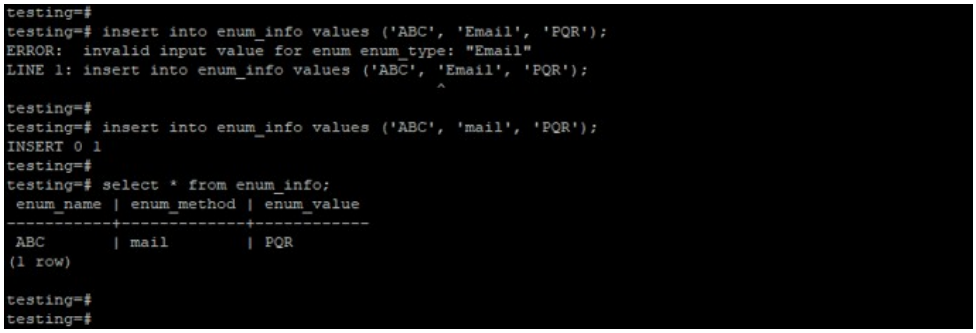
```
SELECT enum_range(NULL::colores)
> {rojo, amarillo, verde, negro, blanco}

SELECT enum_range('amarillo'::colores, 'negro'::colores)
> {amarillo, verde, negro}
```

Así, es posible utilizar enumeraciones previamente declaradas como tipos de datos para las columnas de una tabla

```
CREATE TABLE muebles (
  id     serial PRIMARY KEY,
  nombre text,
  peso   REAL,
  color colores
);
```

Ejemplo

CREATE TYPE enum_type AS ENUM ('mail', 'text_sms', 'Phone_no');
CREATE TABLE enum_info (enum_name text, enum_method enum_type, enum_value text);
INSERT INTO enum_info VALUES ('ABC', 'Email', 'PQR'); ==> Error
iINSERT INTO enum_info VALUES ('ABC', 'mail', 'PQR'); ==> Correcto
SELECT * FROM enum_info;



Claves foráneas

Para definir las claves foráneas en una tabla es suficiente con indicar la tabla y campo clave al que se hace referencia en el momento de definir la tabla. Al igual que en MySQL habrá que tener en cuenta que la tabla a la que se hace referencia debe haber sido creada ya, por lo que tendremos cuidado de colocarla siempre por delante en el script de creación de las mismas.

```
CREATE TABLE ciudades (
  id         serial PRIMARY KEY,
  nombre     text,
  extension REAL
);

CREATE TABLE habitantes (
  id         serial PRIMARY KEY,
  nombre     text,
  apellidos  text,
  ciudad     INT REFERENCES ciudades(id)
);
```

Transacciones

En PostgreSQL una transacción se define entre las instrucciones `BEGIN` y `COMMIT` de la siguiente manera:

```
BEGIN;
  UPDATE ciudades SET extension = 100 WHERE id = 3;
  UPDATE habitantes SET ciudad = 1 WHERE id =  2;
  . . .
COMMIT;
```

Hay que tener en cuenta que algunas aplicaciones clientes engloban todas las instrucciones como transacciones de forma implícita, por lo que conviene leer detenidamente la documentación de dichas aplicaciones.

Herencia

PostgreSQL propone un sistema de herencia para las tablas, lo que permite a una tabla hija heredar las columnas de la tabla padre y, además, tener sus propias columnas.

Cuando se inserta una tupla en la tabla hija, los datos también son visibles desde la tabla padre. Solo se almacenan físicamente en esta tabla las columnas propias de la tabla hija. Los elementos de la tabla padre e hija tendrán el mismo OID ya que se trata de una única instancia en la tabla hija

En este ejemplo, podemos notar que tanto las entidades Clientes y Empleados son especializaciones particulares de Personas, por ende, tanto clientes como empleados heredarían sus atributos. De forma contraria, la entidad de “Personas” corresponderían a una generalización de Clientes y Empleados la cual no tiene atributos específicos de las “subentidades”.

```
-- PASO 1 : Se crea la super-entidad como una tabla normal
CREATE TABLE personas
(numero SERIAL NOT NULL PRIMARY KEY,
nombres   VARCHAR(50),
apellidos  VARCHAR(50),
direccion  VARCHAR(200),
telefono   VARCHAR(20),
fecha_nacim  DATE
);
```

```
-- PASO 2 : Se crean las sub-entidades usando herencia
CREATE TABLE clientes
(nro_cuenta BIGINT,
estado VARCHAR(10),
tipocliente  CHAR(1)
) INHERITS (personas); -- Aquí esta la herencia

CREATE TABLE empleados
(cargo  VARCHAR(25),
departamento  VARCHAR(25),
fecha_ing  DATE
) INHERITS (personas); -- Aquí esta la herencia
```

Algunos pros y contras al usar herencia con PostgreSQL :

Las operaciones DML (INSERT, UPDATE y DELETE) aplican desde las sub-entidades hacia la super-entidad, esto es muy útil ya que permite alivianar las operaciones sobre múltiples tablas a la vez

```
-- Insertamos  en personas
INSERT INTO personas
(numero, nombres, apellidos, direccion,
telefono, fecha_nacim)
VALUES
(2009001, 'Alfonso', 'Lopez', 'Calle Gran Via',
'643960452', NULL);
```

```
- Los datos se insertan sólo en personas
 numero  | nombres | apellidos |   direccion    | telefono  | fecha_nacim
---------+---------+-----------+----------------+-----------+-------------
 2009001 | Alfonso | Lopez     | Calle Gran Via | 643960452 |
(1 row)
```

```
-- Insertamos  en  clientes
INSERT INTO clientes
(numero, nombres, apellidos, direccion, telefono,
fecha_nacim, nro_cuenta, estado, tipocliente)
VALUES
(3006001, 'Antonio', 'Perez', 'Avda. Kansas City',
'670933933', NULL, 900100, 'A', 'V' );
```

```
-- Los datos se insertan en clientes...
SELECT * FROM clientes;
```

```
 numero  | nombres | apellidos |     direccion     | telefono  | fecha_nacim | nro_cuenta | estado | tipocliente
---------+---------+-----------+-------------------+-----------+-------------+------------+--------+-------------
 3006001 | Antonio | Perez     | Avda. Kansas City | 670933933 |             |     900100 | A      | V
(1 row)
```

```
-- ....y también se agregan a personas
SELECT * FROM personas;
```

```
 numero  | nombres | apellidos |     direccion     | telefono  | fecha_nacim
---------+---------+-----------+-------------------+-----------+-------------
 2009001 | Alfonso | Lopez     | Calle Gran Via    | 643960452 |
 3006001 | Antonio | Perez     | Avda. Kansas City | 670933933 |
```

Las super-entidades genéricas permiten organizar de mejor forma los datos para búsquedas distribuidas en distintas tablas.
No es posible modificar (ALTER TABLE) las columnas heredadas en las sub-entidades.
Ya que la herencia no es completa, no se heredan de forma automática las claves primarias, foráneas, secuencias ni índices que puedan tener las super-entidades, sin embargo, esto se puede suplir generando manualmente estos objetos en caso de ser necesario.

ONLY

Cuando seleccionamos los registros de una tabla padre, también se visualizan los que han sido insertado a través de la tabla hija (es como si existiera un vínculo entre ellas a través del OID). Para visualizar los datos que pertenecen solamente a la tabla padre, se usaría ONLY de la siguiente formaban

```
SELECT * FROM ONLY personas;
```

```
 numero  | nombres |  apellidos  |   direccion    | telefono  | fecha_nacim
---------+---------+-------------+----------------+-----------+-------------
 2009001 | GERARDO | VILLABLANCA | Calle Gran Via | 643960452 |
```

Cuando borramos en la tabla padre, la información será borrada también en la hija si existe y viceversa. Si usamos ONLY en la tabla padre sólo se verán afectados aquellos registros que no tengan ocurrencias en la hija, es decir, si borrásemos toda la tabla padre indicando ONLY, no borraría los registros obtenidos a través de la tabla hija

```
DELETE FROM ONLY personas;
```

```
 numero | nombres | apellidos | direccion | telefono | fecha_nacim
--------+---------+-----------+-----------+----------+-------------
(0 rows)
```

```
SELECT * FROM personas;
```

```
 numero  | nombres | apellidos |     direccion     | telefono  | fecha_nacim
---------+---------+-----------+-------------------+-----------+-------------
 3006001 | Antonio | Perez     | Avda. Kansas City | 670933933 |
(1 row)
```

```
SELECT * FROM clientes;
```

```
 numero  | nombres | apellidos |     direccion     | telefono  | fecha_nacim | nro_cuenta | estado | tipocliente
---------+---------+-----------+-------------------+-----------+-------------+------------+--------+-------------
 3006001 | Antonio | Perez     | Avda. Kansas City | 670933933 |             |     900100 | A      | V
(1 row)
```

Funciones

Funciones matemáticas

- `abs(numero)`: Devuelve el valor absoluto de un número

```
SELECT abs(-4)
> 4
```

Funciones de cadena

- `concat(cadena1, cadena2)`: Concatena dos cadena y devuelve el resultado

```
SELECT concat('esto será ', 'una cadena')
> esto será una cadena
```

- `length(cadena)`: Devuelve la longitud de una cadena

```
SELECT LENGTH('una cadena')
> 10
```

- `md5(texto)`: Devuelve el hash del texto que se pasa como parámetro

```
SELECT md5('texto')
> 62059a74e9330e9dc2f537f712b8797c
```

- `substr(cadena, indice, cantidad)`: Devuelve la subcadena que resulta de extraer desde el índice especificado el número de caracteres indicados por `cantidad`

```
SELECT substr('una cadena',
```

- `reverse(cadena)`: Devuelve la cadena inversa

Funciones de Fecha

- `age(timestamp)`: Devuelve el tiempo pasado entre hoy y la fecha que se pasa como parámetro

```
SELECT age(TIMESTAMP '2000-01-01')
> 17 years 3 mons 11 days
```

- `current_date`: Devuelve la fecha de hoy

```
SELECT CURRENT_DATE
> 2012-12-03
```

- `current_time`: Devuelve la hora actual

```
SELECT CURRENT_TIME
> 18:05:25.13039485
```

- `current_timestamp`: Devuelve la fecha y hora de hoy

```
SELECT CURRENT_TIMESTAMP
> 2012-12-03 11:35:32.58700
```

- `extract(datepart)`: Extrae una parte de fecha de una fecha determinada

```
SELECT EXTRACT(MONTH FROM CURRENT_DATE)
> 12
SELECT EXTRACT(DAY FROM CURRENT_DATE)
> 03
```

Funciones de información del sistema

- `current_database()`

```
SELECT current_database()
> prueba
```

- `current_user`

```
SELECT CURRENT_USER
> postgres
```

- `version()`

```
SELECT version()
PostgreSQL 9.5.3 ON x86_64-pc-linux-gnu, compiled BY gcc (Debian 5.3.1-19) 5.3.1 20160509, 64-bit
```

Programación en PostgreSQL

Procedimientos almacenados

Funciones almacenadas

Las funciones de *PostgreSQL*, como ocurre con las de *MySQL*, quedan almacenadas en la Base de Datos donde se crean y pueden ser luego utilizadas en otras estructuras de código o bien directamente desde las consultas `SQL`. Además, como ocurre en todos los lenguajes de programación, las funciones de *PostgreSQL* deben devolver siempre un valor, aunque, como ocurre en lenguajes como *Java*, es posible indicar `void` como palabra reservada en el tipo de devolución y entonces la función ya no tiene que devolver un valor.

```
CREATE [OR REPLACE] FUNCTION <nombre_funcion>(<param1> <tipo>, <param2> <tipo>)
 RETURNS <tipo> AS $$
DECLARE
 -- Declaración de variables
```

```
BEGIN
 -- Instrucciones
END;
$$ LANGUAGE <lenguaje>;
```

Donde `<lenguaje>` se puede sustituir por el lenguaje que queramos usar para escribir el código. Hay varios en PostgreSQL y nosotros nos centraremos en dos de ellos: `SQL` y `plpgsql`. Si empleamos el primero podremos prescindir de las marcas de inicio y final de bloque (`BEGIN` y `END`) y no podremos declarar variables por lo que el bloque `DECLARE` no puede aparecer.

A continuación se muestran algunos ejemplos de funciones almacenadas de *PostgreSQL* con los diferentes lenguajes que se han comentado.

La primera función es un simple ejemplo que lanza una sentencia `SQL` y no devuelve ningún valor. Será suficiente con una simple sentencia `SQL` por lo que podemos utilizar únicamente dicho lenguaje y especificarlo asi al final de la función.

```
CREATE FUNCTION limpiar_articulos() RETURNS void AS $$
  DELETE FROM articulos WHERE precio < 0;
$$ LANGUAGE SQL;
```

La función equivalente utilizando `plpgsql` como lenguaje sería la siguiente.

```
CREATE FUNCTION limpiar_articulos() RETURNS void AS $$
BEGIN
  DELETE FROM articulos WHERE precio < 0;
END;
$$ LANGUAGE plpgsql;
```

En el caso de estas funciones donde indiquemos que vamos a utilizar el lenguaje `SQL` sólo podremos usarlas para lanzar diferentes sentencias del lenguaje realizando sustituciones con los parámetros que se le pasen y devolviendo resultados mediante la instrucción `SELECT`.

Si queremos poder utilizar un lenguaje de programación completo para realizar funciones almacenadas tendremos que indicar, como ya se ha adelantado anteriormente, que usamos el lenguaje `plpgsql` que permitirá emplear sentencias de flujo de código, declaración de variables, asignaciones, . . .

La siguiente función incrementa en una cantidad el precio de un articulo y devuelve el precio final del mismo

```
CREATE FUNCTION subir_precio(id_articulo INTEGER, subida REAL)
  RETURNS REAL AS $$
DECLARE
  precio_final INT;
BEGIN
  UPDATE articulos
    SET precio = precio + subida
    WHERE id = id_articulo;
  precio_final := (SELECT precio FROM articulos WHERE id = id_articulo);
  RETURN precio_final;
END;
$$ LANGUAGE plpgsql;
```

La siguiente función realiza la misma operación pero no se ha dado nombre a los parámetros sino que se ha utilizado su posición en la declaración de la función para identificarlos

```
CREATE FUNCTION subir_precio(INTEGER, REAL)
  RETURNS REAL AS $$
DECLARE
  precio_final INT;
BEGIN
  UPDATE articulos
    SET precio = precio + $2
    WHERE id = $1;
  precio_final := (SELECT precio FROM articulos WHERE id = $1);
  RETURN precio_final;
END;
$$ LANGUAGE plpgsql;
```

La siguiente función vuelve a realizar la misma operación pero se utiliza una claúsula de la propia sentencia `UPDATE` para devolver el precio final del articulo

```
CREATE FUNCTION subir_precio(INTEGER, REAL)
  RETURNS REAL AS $$
BEGIN
```

```
  UPDATE articulos
    SET precio = precio + $2
    WHERE id = $1
    RETURNING precio;
END;
$$ LANGUAGE plpgsql;
```

El siguiente bloque (función y su llamada en una consulta) muestra cómo es posible pasar como parámetro una fila completa para, desde la función, acceder a los campos que sean necesarios para realizar la operación que se desee

```
CREATE FUNCTION precio_iva(articulos)
  RETURNS REAL AS $$
BEGIN
  SELECT $1.precio * 1.16;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT nombre, precio_iva(articulos.*)
FROM articulos
```

Estructuras de control de flujo

- Sentencia IF

```
IF condicion-1 THEN
  . . .
ELSIF condicion-2 THEN
  . . .
ELSE
  . . .
END IF:
```

- Sentencia LOOP

```
<etiqueta>
LOOP
 -- Instrucciones
 EXIT [<etiqueta>] WHEN <condicion>;
END LOOP;
```

- Sentencia FOR

```
[ <etiqueta> ]
FOR <variable_contador> IN [REVERSE] <valor_inicial>.. <valor_inicial> [BY <expresion>]  LOOP
    -- Instrucciones
END LOOP [etiqueta];
```

Sobrecarga de funciones

PostgreSQL permite lo que se conoce como sobrecarga de funciones, que consiste en que es posible declarar más de una función con el mismo nombre siempre y cuando cambie el número de parámetros de la misma.

La siguiente función incrementa el precio de un artículo en una cantidad determinada pero comprueba además que el precio final no sobrepase un precio impuesto como límite. En ese caso no realiza ninguna acción y devuelve un valor `NULL`. En este caso esta segunda función sobrecarga a la primera y en función de los parámetros que se pasen *PostgreSQL* ejecutará una u otra.

```
CREATE FUNCTION subir_precio(id_articulo INTEGER, subida REAL, precio_maximo REAL) RETURNS REAL AS $$
DECLARE
  precio_final REAL;
BEGIN
  precio_final := (SELECT precio FROM articulos WHERE id = id_articulo) +  subida;
  IF precio_final > precio_maximo THEN
    RETURN NULL;
  END IF;
  UPDATE articulos
    SET precio = precio + subida
    WHERE id = id_articulo;
  RETURN precio_final;
END;
$$ LANGUAGE plpgsql;
```

Eliminar una función

Para eliminar una función se utiliza la instrucción `DROP FUNCTION` de la siguiente manera.

```
DROP FUNCTION [IF EXISTS] <nombre_funcion>(tipo_param1, tipo_param2, . . .);
```

Por ejemplo, si quisieramos eliminar las últimas dos funciones creadas justo arriba

```
DROP FUNCTION subir_precio(INTEGER, REAL);
DROP FUNCTION subir_precio(INTEGER, REAL, REAL);
```

Triggers

De forma similar a como ocurren en *MySQL*, los triggers en PostgreSQL se ejecutan siempre asociados a un evento que ha ocurrido sobre una tabla.

```
CREATE TRIGGER trigger_name {BEFORE | AFTER | INSTEAD OF} {event}
   ON TABLE_NAME
   [FOR [EACH] {ROW | STATEMENT}]
   EXECUTE PROCEDURE trigger_function
```

```
CREATE OR REPLACE FUNCTION nombre_mayusculas()
  RETURNS TRIGGER AS
$$
BEGIN
  NEW.nombre := UPPER(NEW.nombre);
END;
$$ LANGUAGE plpgsql
```

```
CREATE TRIGGER nombre_mayusculas
  BEFORE INSERT
  ON articulos
  FOR EACH ROW
  EXECUTE PROCEDURE nombre_mayusculas();
```

Ejercicios

1. Se desea gestionar la información correspondiente a un centro de enseñanza:
   - Por cada alumno se almacenará la siguiente información: DNI, apellidos, nombre, domicilio, teléfono y ciclo que estudia. También se precisa conocer en cada momento las asignaturas en las que está matriculado, así como la nota en cada una de ellas
   - Por cada asignatura guardaremos un código, título y número de horas
   - Cada asignatura puede estar impartida por uno o más profesores. Del profesor se deben conocer los mismos datos que para los alumnos, salvo el del ciclo que estudia. El número máximo de asignaturas que puede impartir un profesor es de 6, aunque puede que no imparta ninguna
   - Algunos profesores tienen un supervisor (sólo uno) que es otro profesor
   - Cada asignatura tendrá un aula asignada, que se identifica mediante el número de edificio y el número de aula. Los números de aulas se pueden repetir entre edificios
2. La Base de Datos COMPAÑÍA se ocupa de los empleados, departamentos y proyectos de una empresa, de acuerdo con los siguientes requisitos:
   - La compañía está organizada en departamentos. Cada departamento tiene un nombre único, un número único y un empleado que lo dirige. Se debe almacenar la fecha en que dicho empleado comenzó a dirigir ese departamento. Hay que tener en cuenta que un departamento puede tener diferentes localizaciones.
   - Cada departamento controla un cierto número de proyectos, cada uno de los cuales tiene un nombre y un número únicos y se realizan en un solo lugar
   - Se almacena el nombre, número de la Seguridad Social, dirección, salario, sexo y fecha de nacimiento de cada empleado. Todo empleado está asignado a un departamento, pero puede trabajar en varios proyectos que no tienen por que ser del mismo departamento. Nos interesa saber el número de horas que un empleado dedica a cada uno de los proyectos asignados
   - También se quiere guardar la relación de las cargas familiares de cada empleado para administrar el seguro médico. Almacenaremos el nombre, sexo y fecha de nacimiento de cada una de las cargas familiares y su parentesco con el empleado
3. Se quiere diseñar una Base de Datos para almacenar todos los datos de un campeonato de fútbol sala que se organiza este año en la ciudad. Aquellos que quieran participar deberán formar un equipo (nombre, patrocinador, color_camiseta, color_2_camiseta, categoría, . . .) e inscribirse en el campeonato. A medida que transcurran los partidos se irán almacenando los resultados de éstos, así como qué equipos lo jugaron, en qué campo se jugó, quién lo arbitró y alguna incidencia que pudiera haber ocurrido (en caso

de que no ocurran incidencias no se anotará nada. Además, los participantes deberán rellenar una ficha de suscripción con algunos datos personales (nombre, apellidos, edad, dirección, teléfono, . . .)

4. Se quiere diseñar una Base de Datos para controlar el acceso a las pistas deportivas de Zaragoza. Se tendrán en cuenta los siguientes supuestos:
   - Todo aquel que quiera hacer uso de las instalaciones tendrá que registrarse y proporcionar su nombre, apellidos, email, teléfono, dni y fecha de nacimiento
   - Hay varios polideportivos en la ciudad, identificados por nombre, dirección, extensión (en m2)
   - En cada polideportivo hay varias pistas de diferentes deportes. De cada pista guardaremos un código que la identifica, el tipo de pista (tenis, fútbol, pádel, . . .), si está operativa o en mantenimiento, el precio y la última vez que se reservó
   - Cada vez que un usuario registrado quiera utilizar una pista tendrá que realizar una reserva previa a través de la web que el ayuntamiento ha creado. De cada reserva queremos registrar la fecha en la que se reserva la pista, la fecha en la que se usará y el precio. Hay que tener en cuenta que todos los jugadores que vayan a hacer uso de la pista deberán estar registrados en el sistema y serán vinculados con la reserva

Prácticas

- **Práctica 5.1** El modelo objeto-relacional en PostgreSQL
- **Práctica 5.2** Creación de una Base de Datos O-R sobre PostgreSQL
- **Práctica 5.3** Consultas y Programación de una Base de Datos O-R sobre PostgreSQL

---

1)
https://www.postgresql.org/docs/9.4/static/arrays.html [https://www.postgresql.org/docs/9.4/static/arrays.html]
2)
https://www.postgresql.org/docs/9.4/static/functions-array.html [https://www.postgresql.org/docs/9.4/static/functions-array.html]
3)
https://www.postgresql.org/docs/9.4/static/rowtypes.html [https://www.postgresql.org/docs/9.4/static/rowtypes.html]
4)
https://www.postgresql.org/docs/9.4/static/datatype-enum.html [https://www.postgresql.org/docs/9.4/static/datatype-enum.html]
5)
https://www.postgresql.org/docs/9.4/static/functions-enum.html [https://www.postgresql.org/docs/9.4/static/functions-enum.html]
6)
https://www.postgresql.org/docs/9.4/static/tutorial-inheritance.html [https://www.postgresql.org/docs/9.4/static/tutorial-inheritance.html]
7)
https://www.postgresql.org/docs/9.4/static/ddl-inherit.html [https://www.postgresql.org/docs/9.4/static/ddl-inherit.html]
