

PostgreSQL. Nivel II

El Sistema Gestor de Bases de Datos Relacionales Orientadas a Objetos conocido como PostgreSQL (y brevemente llamado Postgres95) está derivado del paquete Postgres escrito en Berkeley. Con cerca de una década de desarrollo tras él, PostgreSQL es el gestor de bases de datos de código abierto más avanzado hoy en día, ofreciendo control de concurrencia multi-versión, soportando casi toda la sintaxis SQL (incluyendo subconsultas, transacciones, y tipos y funciones definidas por el usuario), contando también con un amplio conjunto de enlaces con lenguajes de programación (incluyendo C, C++, Java, perl, tcl y python).

CARACTERISTICAS Y VENTAJAS

Postgres ofrece una potencia adicional sustancial a Los sistemas de mantenimiento de Bases de Datos relacionales tradicionales (DBMS,s) al incorporar los siguientes cuatro conceptos adicionales básicos en una vía en la que los usuarios pueden extender fácilmente el sistema:

- Clases
- Herencia
- Tipos
- Funciones

Otras características aportan potencia y flexibilidad adicional:

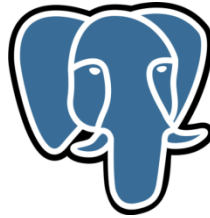
- Restricciones (Constraints)
- Disparadores (triggers)
- Reglas (rules)
- Integridad transaccional

Estas características colocan a Postgres en la categoría de las Bases de Datos identificadas como *objeto-relacionales*.

Además de haberse realizado corrección de errores, con PostgreSQL, el énfasis ha pasado a aumentar características y capacidades, aunque el trabajo continúa en todas las áreas. Algunas mejoras son:

- Los tipos internos han sido mejorados, incluyendo nuevos tipos de fecha/hora de rango amplio y soporte para tipos geométricos adicionales.
- Se han añadido funcionalidades en línea con el estándar SQL92, incluyendo claves primarias, identificadores entrecomillados, forzado de tipos cadena literales, conversión de tipos y entrada de enteros binarios y hexadecimales.
- La velocidad del código del motor de datos ha sido incrementada aproximadamente en un 20-40%, y su tiempo de arranque ha bajado el 80% desde que la versión 6.0 fue lanzada.

- Se han implementado importantes características del motor de datos, incluyendo subconsultas, valores por defecto, restricciones a valores en los campos (constraints) y disparadores (triggers).



PostgreSQL 9.0 incorpora nuevas características y funciones avanzadas en materia de seguridad, soporte de aplicaciones, seguimiento y control, rendimiento y almacenamiento de datos especiales.

Sistema de Gestión de Base de Datos (SGBD)

Los [Sistemas de Gestión de Base de Datos](#) (en inglés DataBase Management System) son un tipo de software muy específico, dedicado a servir de interfaz entre la base de datos, el usuario y las aplicaciones que la utilizan. Se compone de un lenguaje de definición de datos, de un lenguaje de manipulación de datos y de un lenguaje de consulta.

OTROS SISTEMAS DE GESTIÓN DE BASES DE DATOS

SGBD libres

- Firebird
- SQLite (<http://www.sqlite.org>) Licencia Dominio Público
- DB2 Express-C (<http://www.ibm.com/software/data/db2/express/>)
- Apache Derby (<http://db.apache.org/derby/>)
- MariaDB (<http://mariadb.org/>)
- MySQL (<http://dev.mysql.com/>)

SGBD no libres

- MySQL: Licencia Dual, depende del uso.
- dBase
- FileMaker
- Fox Pro
- IBM DB2: Universal Database (DB2 UDB)
- Interbase
- Microsoft Access
- Microsoft SQL Server
- NexusDB
- Open Access
- Oracle
- WindowBase

SGBD no libres y gratuitos

- Microsoft SQL Server Compact Edition Basica
- Sybase ASE Express Edition para Linux (edición gratuita para Linux)
- Oracle Express Edition 10 (solo corre en un servidor, capacidad limitada)

SQL ESTÁNDAR

SQL se ha convertido en el lenguaje de consulta relacional (se basa en el *modelo de datos relacional*) más popular. El nombre “SQL” es una abreviatura de *Structured Query Language* (Lenguaje de consulta estructurado).

SQL nos permite realizar consultas a nuestras bases de datos para mostrar, insertar, actualizar y borrar datos.

PHP

PHP es lenguaje de scripting que permite generar paginas HTML.

A diferencia de las paginas estáticas de HTML que son útiles para presentar documentos estáticos, es decir que no son modificables, PHP permite generar un página HTML en forma dinámica, por ejemplo como resultado de una consulta a una base de datos, o generar gráficos, o cualquier otra cosa que necesite ser generada en base a ciertos datos que pueden cambiar en el tiempo.

Practica

1. Primero que todo, debemos conectarnos al servidor. El usuario y la clave fueron definidos en la instalación.
2. Creamos la base de datos
3. Creamos las tablas necesarias con los campos correspondientes

- Sentencia SQL para crear una **Base de Datos**:

CREATE DATABASE nombre_base de datos

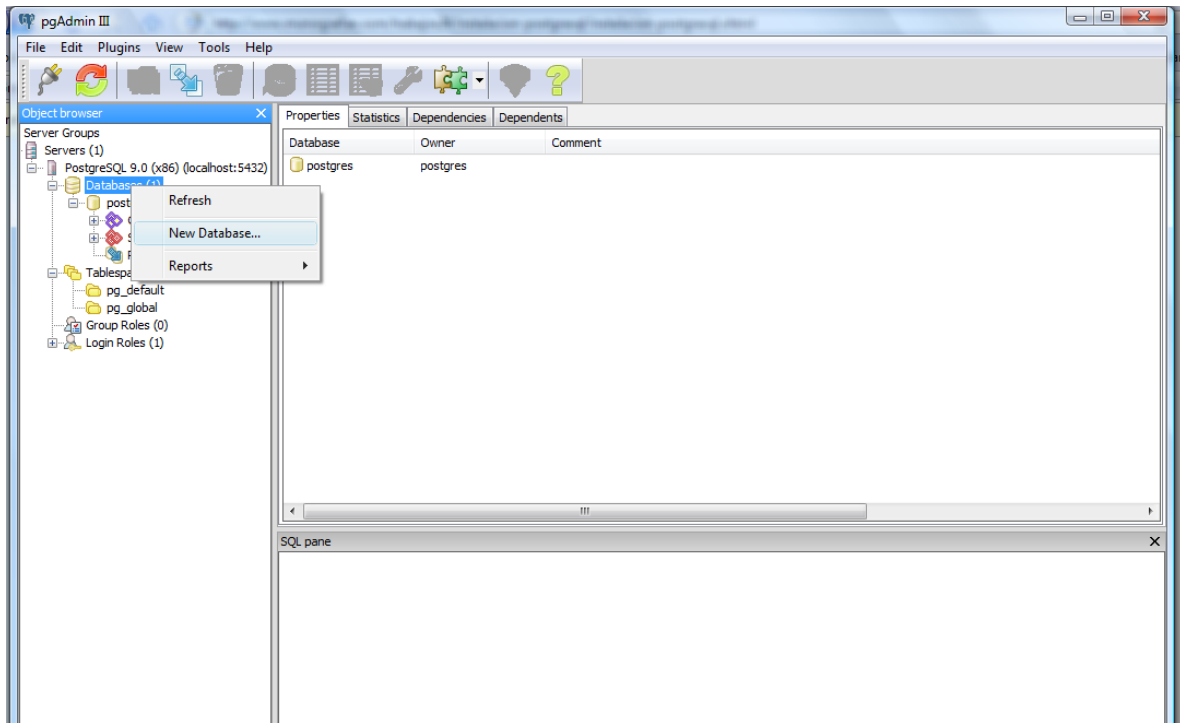
- Sentencia SQL para generar una **Tabla**:

CREATE TABLE nombre_tabla("campo1" int4, "campo2" char Varying(10), "clave1" int4 PRIMARY KEY);

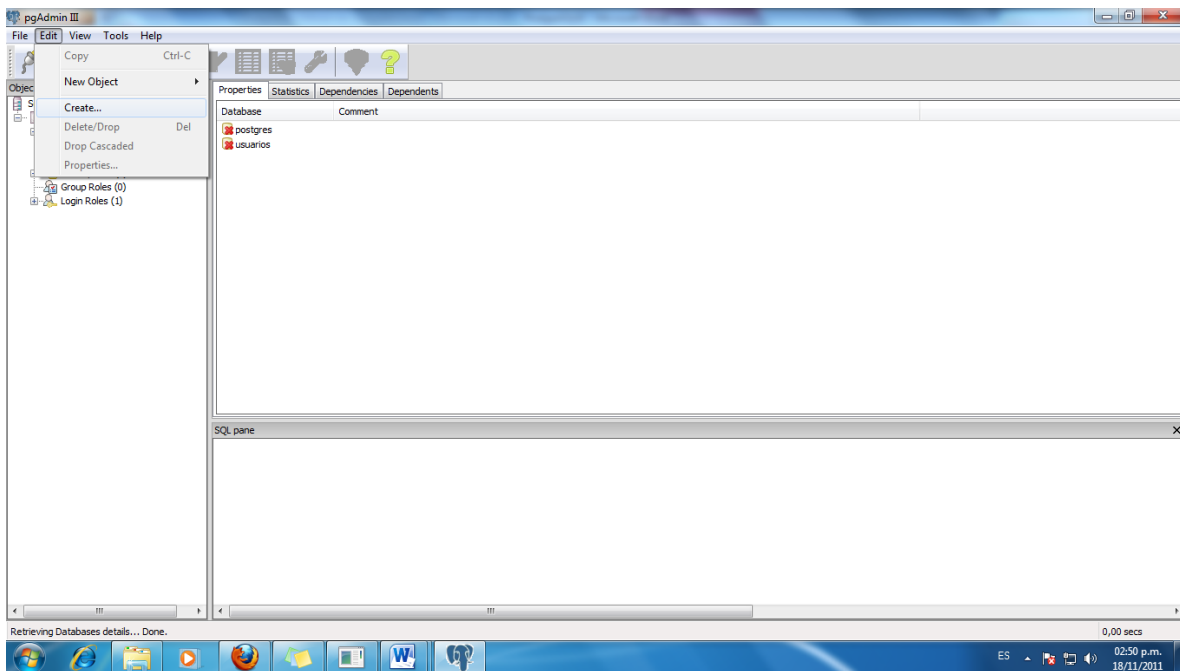
Otra forma de hacerlo es usando el ODBC con cualquier programa que trabaje con bases de datos, por ejemplo Access, PowerBuilder, etc.

En pgAdminIII

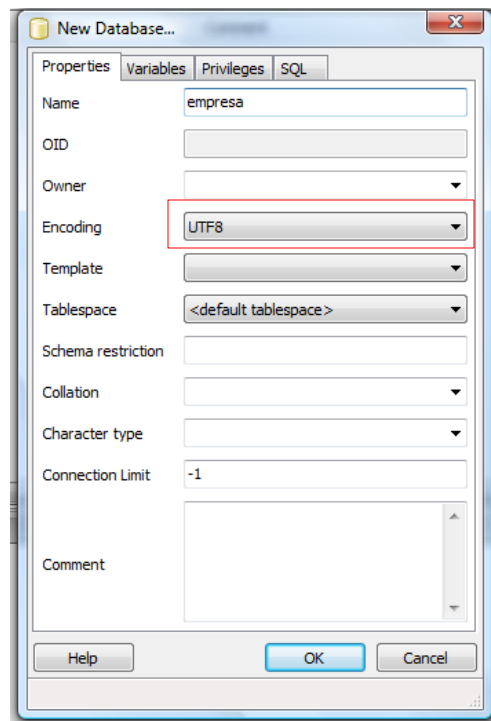
Desde la ventana de administración creamos una base de datos:



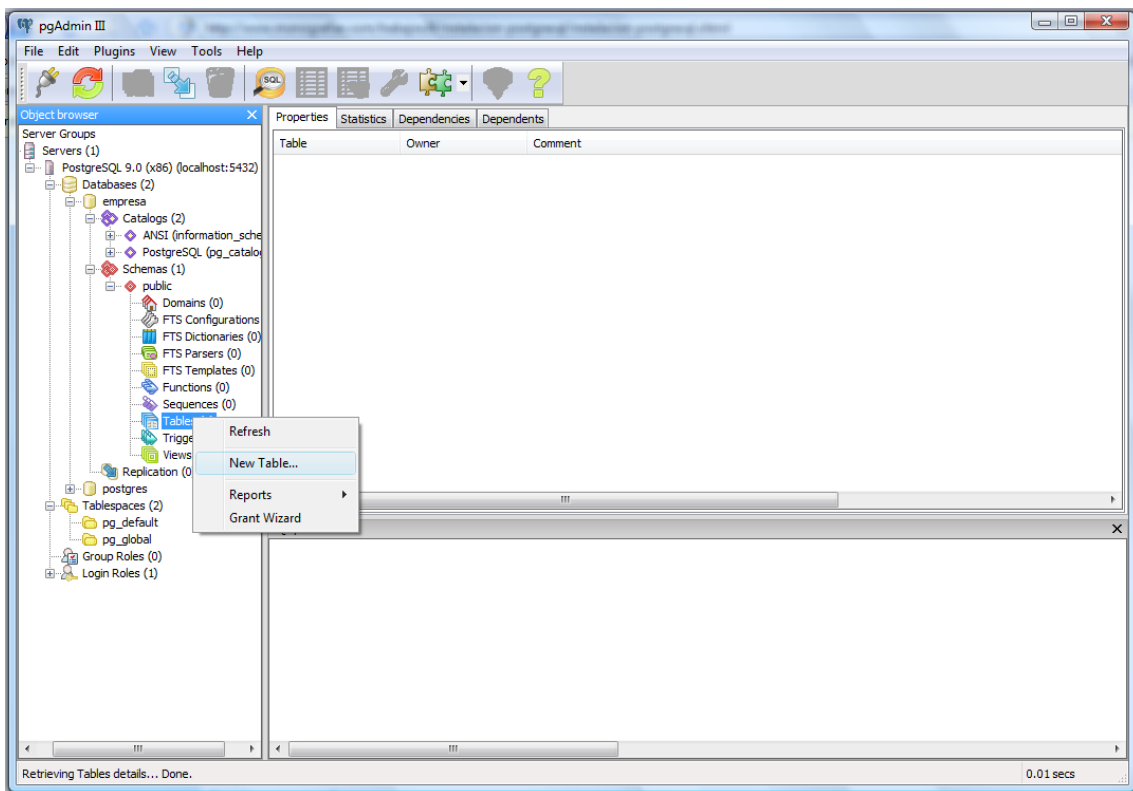
Otra manera:



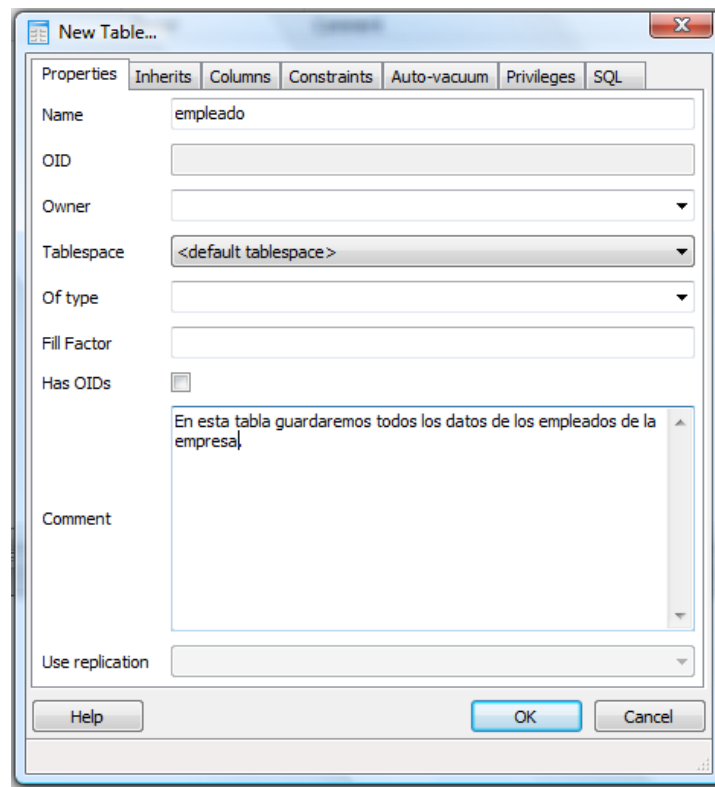
Elegimos el nombre y [codificación](#) (la deseada):



Creamos una tabla en la base de datos:



Le asignamos nombre:

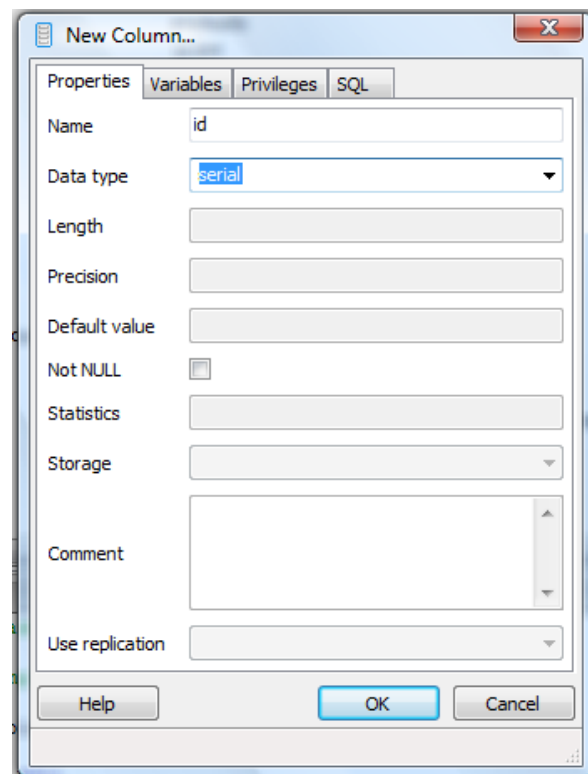


The 'New Table...' dialog box is shown with the following fields and values:

- Name: empleado
- OID: (empty)
- Owner: (empty)
- Tablespace: <default tablespace>
- Of type: (empty)
- Fill Factor: (empty)
- Has OIDs: ☐
- Comment: En esta tabla guardaremos todos los datos de los empleados de la empresa.
- Use replication: (empty)

Buttons: Help, OK, Cancel

Creamos los campos, seleccionando su tipo correspondiente:



The 'New Column...' dialog box is shown with the following fields and values:

- Name: id
- Data type: serial
- Length: (empty)
- Precision: (empty)
- Default value: (empty)
- Not NULL: ☒
- Statistics: (empty)
- Storage: (empty)
- Comment: (empty)
- Use replication: (empty)

Buttons: Help, OK, Cancel

Algunas operaciones con SQL

- **Insertar:** los registros pueden ser introducidos a partir de sentencias que emplean la instrucción *INSERT*.

INSERT into empleado (nombre) Values ('Pedro Meza');

- **Mostrar:** para mostrar los registros se utiliza la instrucción *SELECT*.

SELECT * from empleado;

- **Borrar:** Para borrar un registro se utiliza la instrucción *DELETE*. En este caso debemos especificar cual o cuales son los registros que queremos borrar. Es por ello necesario establecer una selección que se llevara a cabo mediante la cláusula *WHERE*.

DELETE from empleado WHERE id='1';

- **Actualizar:** para actualizar los registros se utiliza la instrucción *UPDATE*. Como para el caso de *DELETE*, necesitamos especificar por medio de *WHERE* cuáles son los registros en los que queremos hacer efectivas nuestras modificaciones. Además, tendremos que especificar cuáles son los nuevos valores de los campos que deseamos actualizar.

UPDATE empleado set nombre='Mario Meza' Where id='1';

Podemos utilizar el editor SQL para realizar estos QUERY



PHP-POSTGRES

Conexion

Antes de trabajar con una base de datos PostgreSQL, debes conectarte al servidor de base de datos, para hacerlo debes conocer la dirección, el nombre de usuario y la contraseña de dicha base de datos.

Normalmente, la dirección de tu base de datos será 'localhost', ya que la ejecutaremos desde el mismo servidor en la que está alojada.

Conexion a Postgres

```
<?php
```

```
    $conexion = pg_connect("host=localhost password=123456 user=postgres
dbname=base");
```

```
    if (pg_ErrorMessage($conexion)) {
```

```
echo "<p><b>Ocurrio un error conectando a la base de datos: .</b></p>";
exit;
```

```
}
```

```
?>
```

Una vez conectados y con una base de datos seleccionada, podemos empezar con las instrucciones de SQL de consulta, edición, inserción, eliminación...

Finalmente, no debemos olvidar la desconexión, indispensable para que no queden puertos abiertos en el servidor y en el servidor de base de datos.

Desconexión Postgres

```
<?php
```

```
    pg_close($conexion);
```

```
?>
```

Todos los códigos que contengan instrucciones de SQL deben estar entre **pg_connect** y **pg_close**.

Luego de conectarse la base de datos, podemos ejecutar comandos SQL, utilizando la función **pg_exec**, pasándole la cadena de conexión **\$conexion** y el comando que queremos ejecutar de la siguiente manera:

```
<?php
```

```
    $query = pg_exec($conexion, "Comando SQL" );
```

```
?>
```


La función devolverá 1 o 0 según se ha ejecutado correctamente o no. En el caso que la consulta requiera resultados devolverá un array con los datos.

```
<?php
    while($fila=pg_fetch_array($consulta)) {
        print "<br>".$fila[0];
    }

?>
```

Ejemplo CREATE TABLE

```
<?php

$sql= "CREATE TABLE 'agenda' ('nombre' varchar(30), 'telefono' int4, 'descripcion'
varchar(30) )";
pg_exec($conexion, $sql );

?>
```

Ejemplo INSERCIÓN

```
<?php

$sql= "INSERT INTO agenda VALUES('Maria Zurbaran', 2122889654, 'Los Caobos')";
pg_exec($conexion, $sql );

?>
```

Ejemplo CONSULTA

```
<?php

$sql= "SELECT * FROM agenda WHERE nombre='Maria Zurbaran' ";
pg_exec($conexion, $sql );

?>
```

CONSULTA e IMPRESIÓN

```
<?php

$sql= "SELECT * FROM agenda WHERE nombre LIKE 'M%' order by nombre limit 0,20 ";
$resultado= pg_exec( $conexion, $sql );
while($fila=pg_fetch_array($resultado)) {
    print "<br>".$fila["nombre"]."-". $fila["telefono"];
}
```

```
}
```

```
?>
```

Ejemplo lectura (consulta a una base de datos Postgres):

```
<?php
```

```
$resultado=pg_exec($conexion, "SELECT * FROM empleado");

if (!$resultado) { echo "<b>Error de busqueda</b>"; exit; }

$filas=pg_numrows($resultado);

if ($filas==0) { echo "No se encontro ningun registro\n"; exit; }

else {

    echo "<ul>";

    for($cont=0;$cont<$filas;$cont++) {

        $campo1=pg_result($resultado,$cont,0);

        $campo2=pg_result($resultado,$cont,1);

        echo " <li>$campo1 \n";

        echo " <li>$campo2 \n";

    }

}

pg_FreeResult($resultado);
```

```
?>
```

Ejemplo actualización de datos

```
<?php
```

```
$resultado=pg_exec($conexion, "UPDATE nombre_tabla SET campo1=valor1, campo2='valor2'
WHERE clave1=valor1 ");
```

```
?>
```

```
<?php
```

```
$resultado=pg_exec($conexion,"UPDATE agenda SET telefono=2396547 WHERE nombre='Maria Zurbaran' ");
```

?>

Interfaces genéricas:

***Insertar:**

Registrar Usuario	
Cédula	<input type="text"/>
Nombre	<input type="text"/>
Dirección	<input type="text"/>
Teléfono	<input type="text"/>
Sexo	<input type="radio"/> Masculino <input type="radio"/> Femenino
Nacionalidad	Venezolano ▼
Ciudad	<input type="text"/>
<input type="button" value="Registrar"/> <input type="button" value="Limpiar"/>	

***Consultar:**

Consulta general de usuario						
Cedula	Nombre	Direccion	Telefono	Sexo	Nacionalidad	Ciudad

***Buscar:**

Buscar Usuario	
Cedula	<input type="text"/>
<input type="button" value="Buscar"/>	

El php+postgres esta instalado en legba y en inter3

----- Funciones agregadas de SQL

Max, Min

Devuelven el mínimo o el máximo de un conjunto de valores contenidos en un campo específico de una consulta. Su sintaxis es:

```
Min(expr)
Max(expr)
```

En donde expr es el campo sobre el que se desea realizar el cálculo. Expr puede incluir el nombre de un campo de una tabla, una constante o una función (la cual puede ser intrínseca o definida por el usuario pero no otras de las funciones agregadas de SQL).

```
SELECT Min(Gastos) FROM Pedidos WHERE Pais = 'España';
SELECT Max(Gastos) FROM Pedidos WHERE Pais = 'España';
```

Sum

Devuelve la suma del conjunto de valores contenido en un campo específico de una consulta. Su sintaxis es:

```
Sum(expr)
```

En donde expr representa el nombre del campo que contiene los datos que desean sumarse o una expresión que realiza un cálculo utilizando los datos de dichos campos.

```
SELECT Sum(PrecioUnidad * Cantidad) AS Total FROM DetallePedido;
```

Count

Calcula el número de registros devueltos por una consulta. Su sintaxis es la siguiente:

```
Count(expr)
```

En donde expr contiene el nombre del campo que desea contar. Puede contar cualquier tipo de datos incluso texto.

```
SELECT Count(*) AS Total FROM Pedidos;
```

EJERCICIO FORO. Hacer Base de Datos. Junto a clave Foránea

- Temas

```
`id` int4 NOT NULL serial,  
`nombre` char varying(30) NOT NULL,  
`tema` char varying(40) NOT NULL,  
`fecha` date NOT NULL,  
`hora` char varying(10) NOT NULL,  
PRIMARY KEY (`id`)
```

- Comentarios

```
`id` int4 NOT NULL serial,  
`nombre` char varying(30) NOT NULL,  
`tema` char varying(40) NOT NULL,  
`fecha` date NOT NULL,  
`hora` char varying(10) NOT NULL,  
`id_tema` int4 NOT NULL,  
PRIMARY KEY (`id`)
```

Disparadores (triggers) en PostgreSQL

Un disparador no es otra cosa que una acción definida en una tabla de nuestra base de datos y ejecutada automáticamente por una función programada por nosotros. Esta acción se activará, según la definamos, cuando realicemos un INSERT, un UPDATE ó un DELETE en la tabla.

Un disparador se puede definir de las siguientes maneras:

- Para que ocurra ANTES de cualquier INSERT, UPDATE ó DELETE
- Para que ocurra DESPUES de cualquier INSERT, UPDATE ó DELETE
- Para que se ejecute una sola vez por comando SQL (statement-level trigger)
- Para que se ejecute por cada linea afectada por un comando SQL (row-level trigger)

Esta es la definición del comando SQL que se puede utilizar para definir un disparador en una tabla:

```
CREATE TRIGGER nombre { BEFORE | AFTER } { INSERT | UPDATE | DELETE [ OR ... ] }  
ON tabla [ FOR [ EACH ] { ROW | STATEMENT } ]  
EXECUTE PROCEDURE nombre de funcion ( argumentos )
```

Antes de definir el disparador tendremos que definir el procedimiento almacenado que se ejecutará cuando nuestro disparador se active.

El procedimiento almacenado usado por nuestro disparador se puede programar en cualquiera de los lenguajes de procedimientos disponibles, entre ellos, el proporcionado por defecto cuando se instala PostgreSQL, PL/pgSQL.

Ejemplos prácticos

Creemos una base de datos para utilizarla con nuestros ejemplos:

```
postgres=# CREATE DATABASE prueba1;  
CREATE DATABASE  
postgres=# \c prueba1  
You are now connected to database "prueba1".  
prueba1=#
```

Lo primero que tenemos que hacer es instalar el lenguaje plpgsql si no lo tenemos instalado.

```
CREATE PROCEDURAL LANGUAGE plpgsql;
```

Ahora creamos una tabla para poder definir nuestro primer disparador:

```
CREATE TABLE numeros(  
    numero bigint NOT NULL,  
    cuadrado bigint,  
    cubo bigint,  
    raiz2 real,  
    raiz3 real,
```

PRIMARY KEY (numero)

);

Después tenemos que crear una función en PL/pgSQL para ser usada por nuestro disparador. Nuestra primera función es la más simple que se puede definir y lo único que hará será devolver el valor NULL:

```
CREATE OR REPLACE FUNCTION proteger_datos() RETURNS TRIGGER AS
$proteger_datos$
DECLARE
BEGIN

--
-- Esta funcion es usada para proteger datos en un tabla
-- No se permitira el borrado de filas si la usamos
-- en un disparador de tipo BEFORE / row-level
--

RETURN NULL;
END;
$proteger_datos$ LANGUAGE plpgsql;
```

A continuación definimos en la tabla números un disparador del tipo BEFORE / row-level para la operación DELETE. Más adelante veremos como funciona:

```
CREATE TRIGGER proteger_datos BEFORE DELETE
ON numeros FOR EACH ROW
EXECUTE PROCEDURE proteger_datos();
```

La definición de nuestra tabla ha quedado así:

```
prueba1=# \d numeros
```

```
Table "public.numeros"
```

```
Column | Type | Modifiers
```

```
-----+-----+-----
```

```
numero | bigint | not null
```

```
cuadrado | bigint |
```

```
cubo | bigint |
```

```
raiz2 | real |
```

```
raiz3 | real |
```

Indexes:

"numeros_pkey" PRIMARY KEY, btree (numero)

Triggers:

proteger_datos BEFORE DELETE ON numeros
FOR EACH ROW EXECUTE PROCEDURE proteger_datos()

Ahora vamos a definir una nueva función un poco más complicada y un nuevo disparador en nuestra tabla numeros:

```
CREATE OR REPLACE FUNCTION rellenar_datos() RETURNS TRIGGER AS $rellenar_datos$
DECLARE
BEGIN
```

```
    NEW.cuadrado := power(NEW.numero,2);
    NEW.cubo := power(NEW.numero,3);
    NEW.raiz2 := sqrt(NEW.numero);
    NEW.raiz3 := cbrt(NEW.numero);
```

```
    RETURN NEW;
END;
```

```
$rellenar_datos$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER rellenar_datos BEFORE INSERT OR UPDATE
ON numeros FOR EACH ROW
EXECUTE PROCEDURE rellenar_datos();
```

La definición de nuestra tabla ha quedado así:

```
prueba1=# \d numeros
      Table "public.numeros"
  Column | Type          | Modifiers
-----+-----+-----
 numero | bigint        | not null
 cuadrado | bigint
 cubo    | bigint
 raiz2   | real
 raiz3   | real
```

Indexes:

"numeros_pkey" PRIMARY KEY, btree (numero)

Triggers:

proteger_datos BEFORE DELETE ON numeros


```
FOR EACH ROW EXECUTE PROCEDURE proteger_datos()
rellenar_datos BEFORE INSERT OR UPDATE ON numeros
FOR EACH ROW EXECUTE PROCEDURE rellenar_datos()
```

Ahora vamos a ver como los disparadores que hemos definido en la tabla numeros funcionan:

```
prueba1=# SELECT * from numeros;
numero | cuadrado | cubo | raiz2 | raiz3
-----+-----+-----+-----+-----
(0 rows)
```

```
prueba1=# INSERT INTO numeros (numero) VALUES (2);
INSERT 0 1
```

```
test001=# SELECT * from numeros;
numero | cuadrado | cubo | raiz2 | raiz3
-----+-----+-----+-----+-----
2 | 4 | 8 | 1.41421 | 1.25992
(1 rows)
```

```
prueba1=# INSERT INTO numeros (numero) VALUES (3);
INSERT 0 1
```

```
prueba1=# SELECT * from numeros;
numero | cuadrado | cubo | raiz2 | raiz3
-----+-----+-----+-----+-----
2 | 4 | 8 | 1.41421 | 1.25992
3 | 9 | 27 | 1.73205 | 1.44225
(2 rows)
```

```
prueba1=# UPDATE numeros SET numero = 4 WHERE numero = 3;
UPDATE 1
```

```
prueba1=# SELECT * from numeros;
numero | cuadrado | cubo | raiz2 | raiz3
-----+-----+-----+-----+-----
2 | 4 | 8 | 1.41421 | 1.25992
4 | 16 | 64 | 2 | 1.5874
```

Hemos realizado 2 INSERT y 1 UPDATE. Esto significa que por cada uno de estos comandos el sistema ha ejecutado la función rellenar_datos(), una vez por cada fila afectada y antes de actualizar la tabla numeros.

Como pueden comprobar, nosotros solamente hemos actualizado la columna numero, pero al listar el contenido de nuestra tabla vemos como el resto de columnas (cuadrado, cubo, raiz2 y raiz3) también contienen valores.

De esta actualización se ha encargado la función rellenar_datos() llamada por nuestro disparador.

Vamos a analizar lo que hace esta función:

```
NEW.cuadrado := power(NEW.numero,2);
NEW.cubo := power(NEW.numero,3);
NEW.raiz2 := sqrt(NEW.numero);
NEW.raiz3 := cbrt(NEW.numero);
```

RETURN NEW;

- Cuando ejecutamos el primer INSERT (numero = 2), el disparador rellenar_datos llama a la función rellenar_datos() una vez.
- El valor de la variable NEW al empezar a ejecutarse rellenar_datos() es numero=2, cuadrado=NULL, cubo=NULL, raiz2=NULL, raiz3=NULL.
- Nuestra tabla todavía no contiene ninguna fila.
- A continuación calculamos el cuadrado, el cubo, la raíz cuadrada y la raíz cubica de 2 y asignamos estos valores a NEW.cuadrado, NEW.cubo, NEW.raiz2 y NEW.raiz3.
- El valor de la variable NEW antes de la sentencia RETURN NEW es ahora numero=2, cuadrado=4, cubo=8, raiz2=1.41421, raiz3=1.25992.
- Con la sentencia RETURN NEW, retornamos la fila (RECORD) almacenada en la variable NEW, y salimos de la función rellenar_datos(). El sistema almacena entonces el RECORD contenido en NEW en la tabla numeros

De la misma manera funciona el disparador proteger_datos cuando ejecutamos una sentencia DELETE. Antes de borrar nada ejecutará la función proteger_datos().

Vistas

Una vista es una alternativa para mostrar datos de varias tablas. Una vista es como una tabla virtual que almacena una consulta. Los datos accesibles a través de la vista no están almacenados en la base de datos como un objeto.

Entonces, una vista almacena una consulta como un objeto para utilizarse posteriormente. Las tablas consultadas en una vista se llaman tablas base. En general, se puede dar un nombre a cualquier consulta y almacenarla como una vista.

Una vista suele llamarse también tabla virtual porque los resultados que retorna y la manera de referenciarlas es la misma que para una tabla.

Una vista se define usando un "select".

La sintaxis básica parcial para crear una vista es la siguiente:

```
create view NOMBREVISTA as
SENTENCIAS SELECT
from TABLA;
```

El contenido de una vista se muestra con un "select":

select * from NOMBREVISTA;

Ejemplo:

```
CREATE TABLE ciudades (
ciudad varchar(80) primary key,
localizacion point
);
```

```
ALTER TABLE ciudades
ADD CONSTRAINT ciudad_pk
PRIMARY KEY (ciudad);
```

```
CREATE TABLE climas (
ciudad varchar(80) references
ciudades(ciudad),
temp_baja int,
temp_alta int,
prcp real,
fecha date);
```

```
ALTER TABLE climas
ADD CONSTRAINT ciudad_fk FOREIGN
KEY (ciudad) REFERENCES ciudades
(ciudad) ON DELETE CASCADE;
```

```
CREATE VIEW MiVista AS
SELECT temp_baja, temp_alta, prcp, fecha,
localizacion
FROM climas, ciudades
WHERE ciudad = nombre;
```

Algunos tipos de datos importantes

char (rango) :	Dato alfanumérico de longitud fija de 30 bytes.
varchar (rango) :	Dato alfanumérico de longitud variable de hasta 30 bytes.
int2:	Dato numérico binario de 2 bytes : $2^{**}-15$ hasta $2^{**}15$
int4:	Dato numérico binario de 4 bytes : $2^{**}-31$ - $2^{**}31$
money:	Dato numérico de coma fija, ej: money(6,3), dato numérico de seis dígitos de los cuales 3 son decimales (3 enteros y tres decimales).
time:	Dato de tiempo que contendrá horas, minutos, segundos, centésimas, HH:MM:SS:CCC
date:	Dato de fecha que contendrá año, mes, día, AAAA/MM/DD
timestamp:	Dato fecha y hora, AAAA/MM/DD:HH:MM:SS:CCC
float (n) :	Dato real de precisión
float3:	Dato real de doble precisión