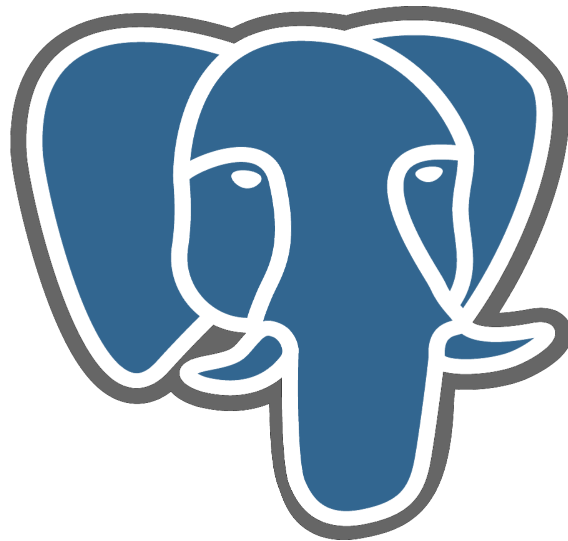
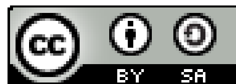
PROYECTO
FORTALECIMIENTO DE LAS CAPACIDADES TIC EN PYMES Y GOBIERNOS LOCALES MEDIANTE EL USO DE SOFTWARE LIBRE



MATERIAL DE APOYO
CURSO BÁSICO DE ADMINISTRACIÓN DEL SGBD POSTGRESQL

Elaborado por:
Ing. Patricia Araya Obando

Programa de las Naciones Unidas Para el Desarrollo
Universidad nacional



This document is licensed under the Attribution-ShareAlike 3.0 Unported license, available at http://creativecommons.org/licenses/by-sa/3.0/.

Tabla de contenidos

Tabla de contenidos....2
Justificación....3
Objetivo General....3
Objetivos Específicos....3
Metodología....4
Requisitos mínimos....4
Conociendo PostgreSQL....6
Instalación:....6
Utilizando apt-get install....6
Utilizando aptitude....7
Información acerca de la instalación....9
El servicio de PostgreSQL....9
Configurar postgresql por primer vez....10
Desinstalar PostgreSQL....15
Utilizaciòn básica de la terminal interactiva....17
Tipos de datos relevantes en PostgreSQL 18
Tipos de datos del estándar SQL3 en PostgreSQL....18
Tipos de datos extendidos por PostgreSQL....19
Tipos de datos de PostgreSQL....20
Operaciones sobre tablas y datos....22
Schemas....23
Ejemplos....23
Permisos y seguridad....24
Eliminar Esquemas....24
Funciones de Schemas....24
public,asis,clie,cont,cred,inve,masi,misc,mutu,pres,rrhh,segu,serv,sise,teso;....24
Limitaciones....24
Disparadores (triggers) en PostgreSQL....25
Características y reglas a seguir:....25
Variables especiales en PL/PGSQL....26
Ejemplos prácticos....27
Algunas anotaciones....36
Herencia....38
Particionamiento de Tablas:....41
Ejemplo....41
Consultas entre bases de datos....43
Tablas temporales....45
Archivos CSV....45
Herramientas gráficas para la administración de bases de datos en PostgreSQL....47
PGADMIN3....47
DbVisualizer....51
Consultas entre bases de datos:....57
Routine Vacuuming....57
Actualización de Planificación de Estadística 57
Autovacuum Daemon....58
Rutina de Indexación 58
Respaldos....59
PG_DUMP....59
Point In Time Recovery (PITR)....60
Replicación....64

Justificación

Hoy en día, casi la totalidad de los sistemas de información existentes consiste en el uso de Sistemas Gestores de Bases de Datos (SGBD) relacionales. En el mercado del software existen múltiples opciones propietarias bien conocidas para suplir esta necesidad, sin embargo en el presente también existen opciones libres que, por su escasa capacidad publicitaria, son notablemente desconocidas, al menos en nuestro entorno. Este curso pretende enseñar las características básicas del SGBD PostgreSQL como alternativa en prestaciones y robustez para cualquier tipo de demanda a los SGBD propietarios más extendidos.

Objetivo General

Brindar al estudiante los conocimientos necesarios para que sea capaz de gestionar bases de datos utilizando el SGBD Postgresql.

Objetivos Específicos

Al final del curso el estudiante estará en capacidad de:

1. Llevar a cabo el proceso de instalación y configuración de Postgresql 8.3.
2. Configurar el SGBD de acuerdo con los recursos del equipo.
3. Asegurar los datos mediante mecanismos de autenticación de clientes.
4. Realizar operaciones básicas de tablas y datos mediante el lenguaje PSQL.
5. Administrar gráficamente bases de datos.
6. Llevar a cabo rutinas de mantenimiento y monitoreo de las bases de datos.
7. Respaldar y recuperar los datos.
8. Utilizar algunas características avanzadas del SGBD como lo son dblik, herencia y particionamiento de tablas.
9. Hacer replicación de bases de datos.

Metodología

El objetivo principal de este manual es lograr transmitir al estudiante los conocimientos necesarios para que sea capaz de gestionar bases de datos utilizando el SGBD Postgresql.

Para ello se hará uso de otros materiales adicionales, los cuales deben ser proveídos al estudiante, ya que este documento hará referencia a ellos.

Requisitos mínimos

Este curso está dirigido a personas con conocimientos básicos de:

Computación, (especialmente en el sistema operativo GNU/Linux).

Lenguaje SQL Estándar.

La distribución GNU/Linux sobre la que está optimizado y probado este manual es Debian Lenny 5.0.3

CAPÍTULO 1

INSTALACIÓN Y CONFIGURACIÓN DEL SGBD

Conociendo PostgreSQL

Para conocer un poco de PostgreSQLy sus características el estudiante deberá iniciar leyendo el artículo “PostgreSQL la alternativa a los SGBD propietarios”, de Carlos Juan Martín Pérez, el cual se encuentra en los materiales adicionales del curso.

Instalación:

PostgreSQL se puede instalar de varias formas, a continuación presentaremos las dos mas comunes utilizando los repositorios, estas son: por medio de apt-get install y por medio de aptitude.

Utilizando apt-get install

Abrir una terminal.

1.Convertirse en usuario root:

```
$su
```

2.Escribir la contraseña de root.

Ya como usuario root:

```
#apt-get install postgresql
```

Se mostrará:
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
Se instalarán los siguientes paquetes extras:
  postgresql-8.3 postgresql-common
Paquetes sugeridos:
  oidentd ident-server
Se instalarán los siguientes paquetes NUEVOS:
  postgresql postgresql-8.3 postgresql-common
0 actualizados, 3 se instalarán, 0 para eliminar y 0 no actualizados.
Se necesita descargar 0B/5586kB de archivos.
Se utilizarán 14.8MB de espacio de disco adicional después de esta operación.
¿Desea continuar [S/n]?

Teclear s

Se mostrará información adicional donde se solicitará que se presione la tecla enter y listo, tenemos postgres instalado.

UTILIZANDO APTITUDE

Como usuario root:

```
#aptitude
```

Se abrirá la aplicación de instalación de paquetes:

```
 Acciones  Deshacer  Paquete  Solucionador  Buscar  Opciones  Vista→
C-T: Menú ?: Ayuda q: Salir u: Actualizar g: Descarga/Instala/Elimis
aptitude 0.4.11.11
--- Paquetes nuevos (673)
--- Paquetes instalados (1584)
--- Paquetes no instalados (20765)
--- Paquetes obsoletos y creados localmente (51)
--- Paquetes virtuales (2474)
--- Tareas (732)

Estos paquetes se han añadido a Debian desde la última vez que
borró la lista de paquetes «nuevos». (elija «Olvidar paquetes
nuevos» del menú Acciones para vaciar la lista).

Este grupo contiene 673 paquetes.
```

Esta aplicación utiliza varias teclas para realizar acciones, por ejemplo:

u = Actualizar el repositorio
U = Instalar actualizaciones de seguridad
/   = Para buscar un paquete hacia abajo
\   = Para buscar un paquete hacia arriba
+ = Para instalar un paquete y sus dependencias.
-   = Para desinstalar un paquete
_ = Para desinstalar un paquete y purgar los archivos.
q = Para salir

Entonces, para instalar el paquete de postgres teclearemos **"/",** se mostrará la caja de diálogo de búsqueda, seguidamente se escribe el nombre del paquete postgresql, conforme se va escribiendo el nombre del paquete se va realizando la búsqueda.

```
 Acciones  Deshacer  Paquete  Solucionador  Buscar  Opciones  Vista→
C-T: Menú ?: Ayuda q: Salir u: Actualizar g: Descarga/Instala/Elimis
aptitude 0.4.11.11
i     postgresql                             8.3.8-0len 8.3.8-0len
i A   postgresql-8.3                         8.3.8-0len 8.3.8-0len
i A   postgresql-client-8.3                  8.3.8-0len 8.3.8-0len
i
i  Buscar:
i  postgresql
Ba        [ Aceptar ]                  [ Cancelar ]
Po
objeto-relacionales. Admite gran parte del estándar SQL y, en
algunos aspectos, está diseñado para que sea extensible por los
usuarios. Algunas de estas características son: transacciones ACID,
claves ajenas, vistas, secuencias, subpeticiones, lanzadores, tipos
y funciones definidos por el usuario, uniones externas y control de
concurrencia multiversión. También están disponibles interfaces
```

Tecleamos enter para cerrar la caja de diálogo de búsqueda o hacemos clic en **"Aceptar"**.

Bien, ahora que hemos encontrado el paquete a instalar lo marcamos tecleando un **“+”,** lo cual marcará el paquete en verde, así como todas las dependencias a instalar.

```
 Acciones  Deshacer  Paquete  Solucionador  Buscar  Opciones  Vista→
C-T: Menú ?: Ayuda q: Salir u: Actualizar g: Descarga/Instala/Elimis
aptitude 0.4.11.11           Se usará 19.7MB de espacio  TamDesc: 72
pi     postgresql                       +279kB  <ninguno>  8.3.8-0len
ciA    postgresql-8.3                   +14.1MB <ninguno>  8.3.8-0len
p      postgresql-8.3-ip4r                      <ninguno>  1.03-2
p      postgresql-8.3-orafce                    <ninguno>  2.1.4-1
p      postgresql-8.3-pljava-gcj                <ninguno>  1.4.0-1.1
p      postgresql-8.3-plproxy                   <ninguno>  2.0.5-2
Base de datos SQL objeto-relacional (versión con soporte)
PostgreSQL es un completo sistema de gestión de bases de datos
objeto-relacionales. Admite gran parte del estándar SQL y, en
algunos aspectos, está diseñado para que sea extensible por los
usuarios. Algunas de estas características son: transacciones ACID,
claves ajenas, vistas, secuencias, subpeticiones, lanzadores, tipos
y funciones definidos por el usuario, uniones externas y control de
concurrencia multiversión. También están disponibles interfaces
```

A continuación tecleamos **“g”** para que se muestren todos los paquetes a instalar

```
 Acciones  Deshacer  Paquete  Solucionador  Buscar  Opciones  Vista→
C-T: Menú ?: Ayuda q: Salir u: Actualizar g: Descarga/Instala/Elimis
             Paquetes                       Previsualizar
aptitude 0.4.11.11           Se usará 19.7MB de espacio  TamDesc: 72
--\ Paquetes automáticamente instalados para satisfacer las dependen
ciA  postgresql-8.3                    +14.1MB <ninguno>  8.3.8-0len
piA  postgresql-client-8.3             +4686kB <ninguno>  8.3.8-0len
piA  postgresql-client-common          +176kB  <ninguno>  94lenny1
ciA  postgresql-common                 +496kB  <ninguno>  94lenny1
--\ Paquetes a instalar (1)

Estos paquetes se instalarán porque algún paquete seleccionado para
instalar los necesita.
```

y por último tecleamos **“g”** nuevamente para que se dé la instalación. En la consola se mostrará:

Preconfigurando paquetes ...
Seleccionando el paquete postgresql-client-common previamente no seleccionado.
(Leyendo la base de datos ...
190283 archivos y directorios instalados actualmente.)
Desempaquetando postgresql-client-common (de .../postgresql-client-common_94lenny1_all.deb) ...
Seleccionando el paquete postgresql-client-8.3 previamente no seleccionado.
Desempaquetando postgresql-client-8.3 (de .../postgresql-client-8.3_8.3.8-0lenny1_i386.deb) ...
Seleccionando el paquete postgresql-common previamente no seleccionado.
Desempaquetando postgresql-common (de .../postgresql-common_94lenny1_all.deb) ...
Seleccionando el paquete postgresql-8.3 previamente no seleccionado.
Desempaquetando postgresql-8.3 (de .../postgresql-8.3_8.3.8-0lenny1_i386.deb) ...
Seleccionando el paquete postgresql previamente no seleccionado.
Desempaquetando postgresql (de .../postgresql_8.3.8-0lenny1_all.deb) ...
Procesando disparadores para man-db ...
Configurando postgresql-client-common (94lenny1) ...
Configurando postgresql-client-8.3 (8.3.8-0lenny1) ...
Configurando postgresql-common (94lenny1) ...
supported_versions: WARNING: Unknown Debian release: 5.0.3
Configurando postgresql-8.3 (8.3.8-0lenny1) ...
Starting PostgreSQL 8.3 database server: main.
Configurando postgresql (8.3.8-0lenny1) ...
Presione ENTER para continuar.

Tecleamos **enter** y listo!! tenemos postgres instalado.
Para salir de aptitude tecleamos **“q”** y hacemos click en **“Sí”**

INFORMACIÓN ACERCA DE LA INSTALACIÓN

Una vez que se instaló PostgreSQLse han creado algunas carpetas que debemos conocer:

Todos los archivos de configuración se encuentran en la carpeta[1]

- /etc/postgresql/8.3/main/

El cluster como tal(programas que se ejecutan y en donde viven las bases de datos) se encuentra en:

- /var/lib/postgresql/8.3/main//var/lib/postgresql/8.3/main/

En caso de errores es necesario consultar el log del SGBD, éste se encuentra en:

- /var/log/postgresql/postgresql-8.3-main.log

y puede ser consultado mediante el comando:

```
tail -f /var/log/postgresql/postgresql-8.3-main.log
```

Este comando irá mostrando en la terminal todos los cambios que se lleven a cabo en el log.

EL SERVICIO DE POSTGRESQL

Postgres ha sido instalado como un servicio, el cual se iniciará automaticamente una vez que se arranque el sistema operativo.

Para detener el servicio:

```
/etc/init.d/postgresql-8.3 stop
```

Para iniciar el servicio:

```
/etc/init.d/postgresql-8.3 start
```

Para reiniciar el servicio:

```
/etc/init.d/postgresql-8.3 restart
```

Para recargar la configuración del servicio:

```
/etc/init.d/postgresql-8.3 reload
```

Cualquier modificación que se desee realizar sobre éstas carpetas, deberá realizarse como usuario root.

1 EL número 8.3 corresponde a la versión de postgres que se ha instalado, por lo que puede variar. En caso de otras distribuciones de postgres la ruta sería /usr/local/pgsql/data

CONFIGURAR POSTGRESQL POR PRIMER VEZ

Al instalarse postgres se crea el usuario postgres. Solamente el usuario root puede convertirse en usuario postgres, para lo cual se debe:

En una terminal:

1. Convertirse en usuario root:

```
$su
```

2. Escribir la contraseña de root.

3. Convertirse en usuario postgres:

```
#su postgres
```

El shell ahora se verá: postgres@nombredelamaquina.

Probando postgres:

Al escribir psql se entra a una terminal interactiva de postgres, en donde se pueden llevar a cabo todas las funcionalidades de postgres.

```
$psql template1
```

Los templates son para que cuando se crea una base de datos tenga ciertas cosas por defecto, es decir, todo lo que se aplique sobre template1 lo tendrán las bases de datos que se creen a partir de ese momento, ya que en postgres cuando se crea una nueva base de datos, lo que se hace es una copia de template1.

Se verá como se muestra a continuación:

```
tita@Tita:~$ su
Contraseña:
Tita:/home/tita# su postgres
postgres@Tita:/home/tita$ psql template1
Bienvenido a psql 8.3.8, la terminal interactiva de PostgreSQL.

Digite:  \copyright para ver los términos de distribución
       \h para ayuda de órdenes SQL
       \? para ayuda de órdenes psql
       \g o punto y coma («;») para ejecutar la consulta
       \q para salir

template1=#
```

A continuación vamos a configurar postgres para poder entrar a la terminal interactiva con el usuario postgres con contraseña, sin tener que convertirse en root.

```
template1=#alter user postgres with password 'password';
```

Para crear un usuario administrador que pueda crear bases de datos:

```
template1=#create user admin with password 'admin' createdb;
```

Para salir del editor psql:

```
temptate1=#\q
```

Para convertirse en usuario root nuevamente:

```
$exit
```

A continuación haremos unas pruebas:

```
#psql template1 -U admin
```

Lo anterior dará un error debido a la configuración por defecto que tiene postgres, para corregirlo vamos a editar editar el archivo pg_hba.conf, que se encuentra en /etc/postgresql/8.3/main
Buscaremos las línea:

local all postgres ident sameuser

y la cambiaremos por:

local all postgres md5

Al igual con:

local all all ident sameuser

Cambiarla por

local all all md5

Hay que recargar la configuración de postgres lo cual se hace con el siguiente comando, como usuario root:

```
#/etc/init.d/postgresql-8.3 reload
```

Haremos otra prueba:

```
 psql template1 -U admin
```

Ahora si nos solicitará la contraseña del usuario admin!!

PostgreSQL se puede empezar a utilizar sin necesidad de configurarlo. Si vamos a utilizar PostgreSQL para algo importante y con cierto volumen de datos y usuarios es imprescindible que lo configuremos para dicho trabajo. La configuración inicial de postgres no toma en cuenta la cantidad de memoria RAM de la máquina, por lo que hay que ajustar algunos valores para que lo haga.

Configuración[2]

El comportamiento de PostgreSQL en nuestro sistema es controlado por medio de tres archivos de configuración que se encuentran en el directorio de datos donde se inicializa el cluster PostgreSQL. Dicho directorio es /etc/postgresql/8.3/main/ . Estos tres archivos son:

- **pg_hba.conf**: que se utiliza para definir los diferentes tipos de accesos que un usuario tiene en el cluster.
- **pg_ident.conf**: que se utiliza para definir la información necesaria en el caso que utilicemos un acceso del tipo ident en pg_hba.conf .
- **postgresql.conf**: En este archivo podemos cambiar todos los parámetros de configuración que afectan al funcionamiento y al comportamiento de PostgreSQL en nuestra maquina.

Pasamos a continuación a explicar los cambios mas importantes que podemos hacer en algunos de estos archivos.

Archivo pg_hba.conf

Este archivo se utiliza para definir como, donde y desde que sitio un usuario puede utilizar el cluster PostgreSQL. Todas las lineas que empiecen con el caracter **#** se interpretan como comentarios. El resto debe de tener el siguiente formato:

```
[Tipoconexión][database][usuario][IP][Netmask][Tipoautenticación][opciones]
```

Dependiendo del tipo de conexión y del tipo de autenticación, [IP],[Netmask] y [opciones] pueden ser opcionales. Vamos a explicar un poco como definir las reglas de acceso.

El tipo de conexión puede tener los siguientes valores, local, host, hostssl y hostnossl. El tipo de método puede tener los siguientes valores, trust, reject, md5, crypt, password, krb5, ident, pam o ldap.

Una serie de ejemplos nos ayudarán a comprender mejor como podemos configurar diferentes accesos al cluster PostgreSQL.

Ejemplo 1 .- Acceso por tcp/ip (red) a la base de datos test001, como usuario test desde la computadora con IP 10.0.0.100, y método de autenticación md5:

host test001 test 10.0.0.100 255.255.255.255 md5

Esta misma entrada se podría escribir también con la mascara de red en notación CIDR:

host test001 test 10.0.0.100/32 md5

Ejemplo 2 .- Acceso por tcp/ip (red) a la base de datos test001, como usuario test desde todos los ordenadores de la red 10.0.0.0, con mascara de red 255.255.255.0 (254 ordenadores en total) y método de autentificar md5:

host test001 test 10.0.0.0 255.255.255.0 md5

Esta misma entrada se podría escribir también con la mascara de red en notación CIDR:

host test001 test 10.0.0.0/24 md5

2 Tomado de http://www.linux-es.org/node/660, Introducción a PostgresQL-Configuración, El rincón de Linux

Ejemplo 3 .- Acceso por tcp/ip (red), encriptado, a todas las bases de datos de nuestro cluster, como usuario test desde la computadora con IP 10.0.0.100, y la computadora 10.1.1.100 y método de autenticación md5 (necesitamos dos entradas en nuestro archivo pg_hba.conf:

hostssl all test 10.0.0.100 255.255.255.255 md5
hostssl all test 10.1.1.100 255.255.255.255 md5

Ejemplo 4.- Denegar el acceso a todos las bases de datos de nuestro cluster al usuario test, desde todos los ordenadores de la red 10.0.0.0/24 y dar acceso al resto del mundo con el método md5:

host all test 10.0.0.0/24 reject
host all all 0.0.0.0/0 md5

Así podríamos seguir jugando con todas las posibilidades que nos brinda este archivo de configuración. Por supuesto que las bases de datos y usuarios usados en este archivo tienen que existir en nuestro cluster para que todo funcione.

Para poner en producción los cambios en este archivo tendremos que decirle a PostgreSQL que vuelva a leerlo. Basta con un simple *'reload'* (/etc/init.d/postgresql-8.3 reload) desde la línea de comandos o con la función pg_reload_conf() como usuario postgres desde psql, el cliente PostgreSQL.(postgres=# SELECT pg_reload_conf();)

Para una documentación detallada sobre el archivo pg_hba.conf, consultar la sección Chapter 20. Client Authentication de la documentación oficial de PostgreSQL.

Archivo postgresql.conf

Los cambios que se realicen en este archivo afectarán a todas las bases de datos que tengamos definidas en nuestro cluster PostgreSQL. La mayoría de los cambios se pueden poner en producción con un simple *'reload'* (/etc/init.d/postgresql-8.3 reload), otros cambios necesitan que arranquemos de nuevo nuestro cluster (/etc/init.d/postgresql-8.3 restart)

Mas información sobre todos los parámetros que podemos cambiar en este archivo, que afectan y como se pueden poner en producción se puede encontrar en la sección 17. Server Configuration de la documentación oficial de PostgreSQL.

A continuación vamos a ver los parámetros mas importantes que deberíamos cambiar si empezamos a usar PostgreSQL para un uso serio y si queremos sacarle el máximo partido a nuestra máquina.

max_connections: Número máximo de clientes conectados a la vez a nuestras bases de datos. Deberíamos de incrementar este valor en proporción al numero de clientes concurrentes en nuestro cluster PostgreSQL. Un buen valor para empezar es el 100:
max_connections = 100

shared_buffers: Este parámetro es importantísimo y define el tamaño del buffer de memoria utilizado por PostgreSQL. No por aumentar este valor mucho tendremos mejor respuesta. En un servidor dedicado podemos empezar con un 25% del total de nuestra memoria. Nunca mas de 1/3 (33%) del total. Por ejemplo, en un servidor con 4Gbytes de memoria, podemos usar 1024MB como valor inicial.
shared_buffers = 1024MB

work_mem: Usada en operaciones que contengan ORDER BY, DISTINCT, joins, ... En un servidor dedicado podemos usar un 2-4% del total de nuestra memoria si tenemos solamente unas pocas sesiones (clientes) grandes. nunca mas de RAM/num.conexiones. Como valor inicial podemos usar 8 Mbytes, para aplicaciones web, y hasta 128MB para una aplicación de datawarehouse.

work_mem = 8MB

maintenance_work_mem: Usada en operaciones del tipo VACUUM, ANALYZE, CREATE INDEX, ALTER TABLE, ADD FOREIGN KEY. Su valor dependerá mucho del tamaño de nuestras bases de datos. Por ejemplo, en un servidor con 4Gbytes de memoria, podemos usar 256MB como valor inicial. La fórmula es 1/16 de nuestra memoria RAM.

maintenance_work_mem = 256MB

effective_cache_size: Parámetro usado por el 'query planner' de nuestro motor de bases de datos para optimizar la lectura de datos. En un servidor dedicado podemos empezar con un 50% del total de nuestra memoria. Como máximo unos 2/3 (66%) del total. Por ejemplo, en un servidor con 4Gbytes de memoria, podemos usar 2048MB como valor inicial.

effective_cache_size = 2048MB

checkpoint_segments: Este parámetro es muy importante en bases de datos con numerosas operaciones de escritura (insert,update,delete). Para empezar podemos empezar con un valor de 64. En grandes bases de datos con muchos Gbytes de datos escritos podemos aumentar este valor hasta 128-256.

checkpoint_segments = 64

max_stack_depth (integer): Especifica el tamaño máximo de profundidad de la pila de ejecución del servidor .El tamanño ideal debe ser el resultado del comando ulimit -s - 1MB

Parámetros del kernel:

Es muy importante tener en cuenta que al aumentar los valores por defecto de muchos de estos parámetros, tendremos que aumentar los valores por defecto de algunos parámetros del kernel de nuestro sistema. Información detallada de como hacer esto se encuentra en la sección 16.4. Managing Kernel Resources de la documentación oficial de PostgreSQL.

Uno de los valores más importantes es shmmax, el valor que le vamos a asignar es igual a 1/3 de la RAM disponible en Bytes.
En Linux 2.4 y posteriores, el comportamiento por defecto de la memoria virtual no es óptimo para postgres, para ello modificaremos el parámetro: overcommit_memory[3].

En el archivo /etc/sysctl agregar:

kernel.shmmax = 1/3 de la RAM disponible en bytes.
vm.overcommit_memory=2

En consola, como usuario root:

```
#sysctl -w kernel.shmmax = 1/3 de la RAM disponible en bytes.
#sysctl -w vm.overcommit_memory=2
```

3 http://www.network-theory.co.uk/docs/postgresql/vol3/LinuxMemoryOvercommit.html

DESINSTALAR POSTGRESQL

En caso que se desee desintalar postgresql, como usuario root:

```
#aptitude purge postgresql
```

Para borrar los archivos de configuración:

```
# aptitude search ~c
# aptitude purge ~c
```

CAPÍTULO 2

SQL

Y

CARACTERÍSTICAS PARTICULARES DEL SGBD

Utilizaciòn básica de la terminal interactiva

Para ingresar a la terminal con el usuario user a la base de datos base:

```
$psql -U user  base
```

Para ingresar a una base de datos en otro equipo:

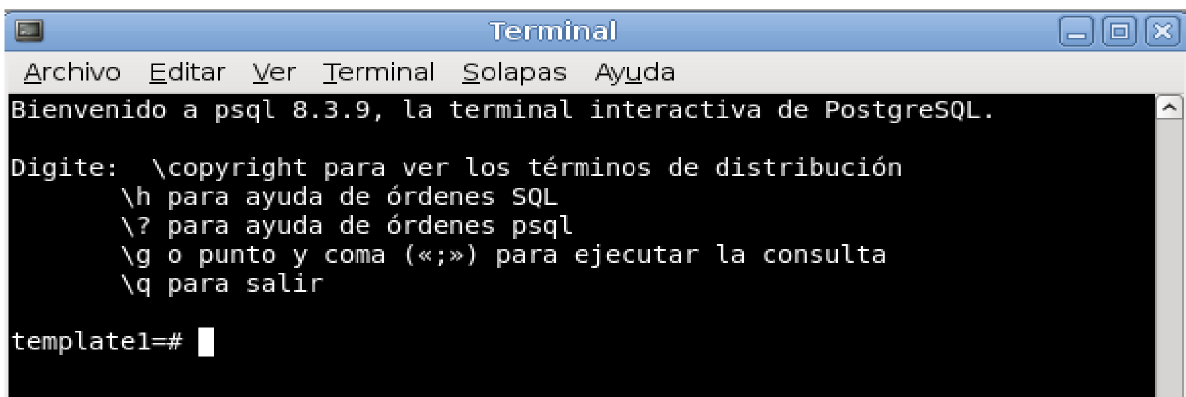
```
$psql -U user -h host -d basededatos
```

donde host es la dirección ip del equipo remoto o el nombre de un host que esté registrado en el archivo /etc/hosts

Para cargar un script sql en una base de datos:

```
$psql -U user basededatos < archivo
```

Una vez en la terminal:



la terminal nos indica que estamos en la base de datos template1.

Para conectarse a otra base de datos utilizamos **\c**:

```
template1=# \c ejemplo admin
Contraseña para usuario admin:
Ahora está conectado a la base de datos «ejemplo» como el usuario «admin»
.
ejemplo=>
```

Para cargar un script sql desde dentro de la terminal:

```
\i /home/usuario/archivo
```

En esta terminal podemos escribir los comandos sql, solamente se van a ejecutar cuando se escriba un **“;”**:

```
ejemplo=> select * from cities
ejemplo-> where name like '%San%';
     name      | location
---------------+-----------
 San Francisco | (-194,53)
(1 fila)
```

Para conocer mas comandos de la terminal interactiva

```
 \?
```

Tipos de datos relevantes en PostgreSQL

Como todos los manejadores de bases de datos, PostgreSQL implementa los tipos de datos definidos para el estándar SQL3 y aumenta algunos otros. A continuación se presentan los tipos de datos de postgres.

Tipos de datos del estándar SQL3 en PostgreSQL

Tipos de datos del estándar SQL3 en PostgreSQL		
Tipo en Postgres	Correspondiente en SQL3	Descripción
bool	boolean	valor lógico o booleano (true/false)
char(n)	character(n)	cadena de carácteres de tamaño fijo
date	date	fecha (sin hora)
float4/8	float(86#86)	número de punto flotante con precisión 86#86
float8	real, double precision	número de punto flotante de doble precisión
int2	smallint	entero de dos bytes con signo
int4	int, integer	entero de cuatro bytes con signo
int4	decimal(87#87)	número exacto con 88#88
int4	numeric(87#87)	número exacto con 89#89
money	decimal(9,2)	cantidad monetaria
time	time	hora en horas, minutos, segundos y centésimas
timespan	interval	intervalo de tiempo
timestamp	timestamp with time zone	fecha y hora con zonificación
varchar(n)	character varying(n)	cadena de carácteres de tamaño variable

TIPOS DE DATOS EXTENDIDOS POR POSTGRESQL

Tipos de datos extendidos en PostgreSQL	
Tipo	Descripción
box	caja rectangular en el plano
cidr	dirección de red o de *host* en IP versión 4
circle	círculo en el plano
inet	dirección de red o de *host* en IP versión 4
int8	entero de ocho bytes con signo
line	línea infinita en el plano
lseg	segmento de línea en el plano
path	trayectoria geométrica, abierta o cerrada, en el plano
point	punto geométrico en el plano
polygon	trayectoria geométrica cerrada en el plano
serial	identificador numérico único

Tipos de datos Autoincrementales(serial):

Para declarar un campo como autoincremental:
create table tabla (
campo serial
campo2 ….
campo3...
)

Para hacer un insert:
Insert into table (campo2,campo3) values(... ,...)
Especificar los demas campos de la tabla a insertar, no incluir el campo tipo serial.

Para agregarle un campo incremental a una tabla:
No existe alter table add column campo serial, por lo que hay que hacer:

ALTER TABLE tabla ADD campo int8;

CREATE SEQUENCE nombredelasecuencia_seq;
ALTER TABLE tabla ALTER COLUMN campo SET DEFAULT nextval(nombredelasecuencia_seq);
UPDATE tabla SET campo=nextval('nombredelasecuencia_seq');

TIPOS DE DATOS DE POSTGRESQL

Tipo	Descripción
`SET`	conjunto de tuplas
`abstime`	fecha y hora absoluta de rango limitado (Unix system time)
`aclitem`	lista de control de acceso
`bool`	booleano '*true*'/'*false*'
`box`	rectángulo geométrico '(izquierda abajo, derecha arriba)'
`bpchar`	carácteres rellenos con espacios, longitud especificada al momento de creación
`bytea`	arreglo de bytes de longitud variable
`char`	un sólo carácter
`cid`	*command identifier type*, identificador de secuencia en transacciones
`cidr`	dirección de red
`circle`	círculo geométrico '(centro, radio)'
`Date`	fecha ANSI SQL 'aaaa-mm-dd'
`datetime`	fecha y hora 'aaaa-mm-dd hh:mm:ss'
`filename`	nombre de archivo usado en tablas del sistema
`float4`	número real de precisión simple de 4 bytes
`float8`	número real de precisión doble de 8 bytes
`inet`	dirección de red
`int2`	número entero de dos bytes, de -32k a 32k
`int28`	8 numeros enteros de 2 bytes, usado internamente
`int4`	número entero de 4 bytes, -2B to 2B
`int8`	número entero de 8 bytes, 90#9018 dígitos
`line`	línea geométrica '(pt1, pt2)'
`lseg`	segmento de línea geométrica '(pt1, pt2)'
`macaddr`	dirección MAC
`money`	unidad monetaria '$d,ddd.cc'
`name`	tipo de 31 carácteres para guardar identificadores del sistema
`numeric`	número de precisión múltiple
`oid`	tipo de identificación de objetos
`oid8`	arreglo de 8 *oid*s, utilizado en tablas del sistema
`path`	trayectoria geométrica '(pt1, ...)'
`point`	punto geométrico '(x, y)'

Tipo	Descripción
polygon	polígono geométrico '(pt1, ...)'
regproc	procedimiento registrado
reltime	intervalo de tiempo de rango limitado y relativo (Unix delta time)
smgr	manejador de almacenamiento (*storage manager*)
text	cadena de carácteres nativa de longitud variable
tid	tipo de identificador de tupla, localización física de tupla
time	hora ANSI SQL 'hh:mm:ss'
timespan	intervalo de tiempo '@ <number> <units>'
timestamp	fecha y hora en formato ISO de rango limitado
tinterval	intervalo de tiempo '(abstime, abstime)'
unknown	tipo desconocido
varchar	cadena de carácteres sin espacios al final, longitud especificada al momento de creación
xid	identificador de transacción

OPERACIONES SOBRE TABLAS Y DATOS

En los materiales adicionales del curso se encuentra una carpeta llamada “Base de Datos de Ejemplo”, la cual contiene scripts con ejemplos de operaciones básicas sobre tablas.

El archivo **diagrama_EER.pdf** contiene el modelo Entidad-Relación de la Base de Datos a utilizar como ejemplo.

Para repasar las operaciones básicas sobre tablas el estudiante debe estudiar el archivo llamado:

- **base_de_datos.sql**
- **consultas_resueltas.sql**

Para ver un resúmen de las funciones de PostgreSQL, el estudiante debe estudiar el archivo:

- **postgresql_cheat_sheet.pdf**

SCHEMAS

Un esquema es esencialmente un espacio de nombres: contiene el nombre de objetos (tablas, tipos de datos, funciones y operadores), cuyos nombres pueden duplicar los de otros objetos existentes en otros esquemas. Los objetos con nombre se accede bien por "calificación" de sus nombres con el nombre de esquema como prefijo, o mediante el establecimiento de una ruta de búsqueda que incluye el esquema deseado . Un comando CREATE especifica un nombre de objeto no calificado, crea el objeto en el esquema actual .

EJEMPLOS

Funcionalidad básica:

```
test=# CREATE SCHEMA foo;
```

CREATE SCHEMA

```
test=# CREATE TABLE foo.info (id INT, txt TEXT);
```

CREATE TABLE

```
test=# INSERT INTO foo.info VALUES(1, 'This is schema foo');
```

INSERT 23062 1

```
test=# CREATE SCHEMA bar;
```

CREATE SCHEMA

```
test=# CREATE TABLE bar.info (id INT, txt TEXT);
```

CREATE TABLE

```
test=# INSERT INTO bar.info VALUES(1, 'This is schema bar');
```

INSERT 23069 1

```
test=# SELECT foo.info.txt, bar.info.txt
test-#   FROM foo.info, bar.info
test-#  WHERE foo.info.id=bar.info.id;
```

```
       txt         |       txt
--------------------+--------------------
 This is schema foo | This is schema bar
(1 row)
```

Cambiar de schema:

```
test=# SET search_path TO foo;
```

SET

Para cambiar permanentemente el schema en cada conexión:

```
ALTER USER test SET search_path TO bar,foo;
```

El cambio surtirá efecto solamente después de conectarse de nuevo.

```
test=# SET search_path TO bar, foo;
```

SET

```
test=# SELECT txt FROM info;
       txt
--------------------
 This is schema bar
(1 row)
```

```
test=# SELECT * FROM info_view;
ERROR:  Relation "info_view" does not exist
test=# SELECT * FROM public.info_view;
       foo          |       bar
--------------------+--------------------
 This is schema foo | This is schema bar
```

PERMISOS Y SEGURIDAD

Los esquemas solamente pueden ser creados por los superusuarios.

Para crear un esquema para otro usuario use:

CREATE SCHEMA tarzan AUTHORIZATION tarzan;

o

CREATE SCHEMA AUTHORIZATION tarzan;

ELIMINAR ESQUEMAS

DROP SCHEMA tarzan;

Si ya existen datos en el esquema:

DROP SCHEMA tarzan CASCADE;

FUNCIONES DE SCHEMAS

- current_schema()
  Retorna el nombre del esquema actual.
- current_schemas(boolean)
  Retorna todos los esquemas en search path

SET search_path TO

PUBLIC,ASIS,CLIE,CONT,CRED,INVE,MASI,MISC,MUTU,PRES,RRHH,SEGU,SERV,SISE,TESO;

LIMITACIONES

no es posible transferir objetos entre esquemas, para ello:

CREATE TABLE new_schema.mytable AS SELECT * FROM old_schema.mytable

INSERT INTO new_schema.mytable SELECT * FROM old_schema.mytable;

Disparadores (triggers) en PostgreSQL[4]

Una de las funcionalidades disponibles en PostgreSQL son los denominados disparadores (triggers). 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 susodicha 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. Este lenguaje es el que utilizaremos en todos los ejemplos de este artículo.

Características y reglas a seguir:

Las reglas más importantes a tener en cuenta, cuando definamos un disparador y/ó programemos un procedimiento almacenado que se vaya a utilizar por un disparador:

- El procedimiento almacenado que se vaya a utilizar por el disparador debe de definirse e instalarse antes de definir el propio disparador.
- Un procedimiento que se vaya a utilizar por un disparador no puede tener argumentos y tiene que devolver el tipo "trigger".
- Un mismo procedimiento almacenado se puede utilizar por múltiples disparadores en diferentes tablas.
- Procedimientos almacenados utilizados por disparadores que se ejecutan una sola vez per comando SQL (statement-level) tienen que devolver siempre NULL.
- Procedimientos almacenados utilizados por disparadores que se ejecutan una vez per linea afectada por el comando SQL (row-level) pueden devolver una fila de tabla.
- Procedimientos almacenados utilizados por disparadores que se ejecutan una vez per fila afectada por el comando SQL (row-level) y ANTES de ejecutar el comando SQL que lo lanzó, pueden:

  -Retornar NULL para saltarse la operación en la fila afectada.
  -Ó devolver una fila de tabla (RECORD)

4 Tomado de http://www.postgresql-es.org/node/301, © Copyright 2009 PosgreSQL-es.org - Rafael Martinez

- Procedimientos almacenados utilizados por disparadores que se ejecutan DESPUES de ejecutar el comando SQL que lo lanzó, ignoran el valor de retorno, asi que pueden retornar NULL sin problemas.

En resumen, independendientemente de como se defina un disparador, el procedimiento almacenado utilizado por dicho disparador tiene que devolver ó bien NULL, ó bien un valor RECORD con la misma estructura que la tabla que lanzó dicho disparador.

Si una tabla tiene más de un disparador definido para un mismo evento (INSERT,UPDATE,DELETE), estos se ejecutarán en orden alfabético por el nombre del disparador. En el caso de disparadores del tipo ANTES / row-level, la file retornada por cada disparador, se convierte en la entrada del siguiente. Si alguno de ellos retorna NULL, la operación será anulada para la fila afectada.

Procedimientos almacenados utilizados por disparadores pueden ejecutar sentencias SQL que a su vez pueden activar otros disparadores. Esto se conoce como disparadores en cascada. No existe límite para el número de disparadores que se pueden llamar pero es responsabilidad del programador el evitar una recursión infinita de llamadas en la que un disparador se llame así mismo de manera recursiva.

Otra cosa que tenemos que tener en cuenta es que, por cada disparador que definamos en una tabla, nuestra base de datos tendrá que ejecutar la función asociada a dicho disparador. El uso de disparadores de manera incorrecta ó inefectiva puede afectar significativamente al rendimiento de nuestra base de datos. Los principiantes deberian de usar un tiempo para entender como funcionan y asi poder hacer un uso correcto de los mismos antes de usarlos en sistemas en producción.

Variables especiales en PL/PGSQL

Cuando una función escrita en PL/pgSQL es llamada por un disparador tenemos ciertas variable especiales disponibles en dicha función. Estas variables son las siguientes:

NEW

Tipo de dato RECORD; Variable que contiene la nueva fila de la tabla para las operaciones INSERT/UPDATE en disparadores del tipo row-level. Esta variable es NULL en disparadores del tipo statement-level.

OLD

Tipo de dato RECORD; Variable que contiene la antigua fila de la tabla para las operaciones UPDATE/DELETE en disparadores del tipo row-level. Esta variable es NULL en disparadores del tipo statement-level.

TG_NAME

Tipo de dato name; variable que contiene el nombre del disparador que está usando la función actualmente.

TG_WHEN

Tipo de dato text; una cadena de texto con el valor BEFORE o AFTER dependiendo de como el disparador que está usando la función actualmente ha sido definido

TG_LEVEL
Tipo de dato text; una cadena de texto con el valor ROW o STATEMENT dependiendo de como el disparador que está usando la función actualmente ha sido definido

TG_OP
Tipo de dato text; una cadena de texto con el valor INSERT, UPDATE o DELETE dependiendo de la operación que ha activado el disparador que está usando la función actualmente.

TG_RELID
Tipo de dato oid; el identificador de objeto de la tabla que ha activado el disparador que está usando la función actualmente.

TG_RELNAME
Tipo de dato name; el nombre de la tabla que ha activado el disparador que está usando la función actualmente. Esta variable es obsoleta y puede desaparacer en el futuro. Usar TG_TABLE_NAME.

TG_TABLE_NAME
Tipo de dato name; el nombre de la tabla que ha activado el disparador que está usando la función actualmente.

TG_TABLE_SCHEMA
Tipo de dato name; el nombre de la schema de la tabla que ha activado el disparador que está usando la función actualmente.

TG_NARGS
Tipo de dato integer; el número de argumentos dados al procedimiento en la sentencia CREATE TRIGGER.

TG_ARGV[ ]
Tipo de dato text array; los argumentos de la sentencia CREATE TRIGGER. El índice empieza a contar desde 0. Indices inválidos (menores que 0 ó mayores/iguales que tg_nargs) resultan en valores nulos.

Ejemplos prácticos

Una vez que hemos visto la teoria básica de disparadores nada mejor que unos cuantos ejemplos prácticos para ver como se usan y defininen los disparadores en PostgreSQL. (estos ejemplos han sido comprobados en postgreSQL 8.3.7).

Creamos una base de datos para utilizarla con nuestros ejemplos:

```
postgres@server:~$ psql
```

Welcome to psql 8.3.7, the PostgreSQL interactive terminal.
Type: \copyright for distribution terms
 \h for help with SQL commands
 \? for help with psql commands
 \g or terminate with semicolon to execute query
 \q to quit

```
postgres=# CREATE DATABASE test001;
```
CREATE DATABASE

```
postgres=# \c test001
```
You are now connected to database "test001".

```
test001=#
```

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 *numeros* 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:

```
test001=# \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:

```
test001=# \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:

```
test001=# SELECT * from numeros;
 numero | cuadrado | cubo | raiz2 | raiz3
--------+----------+------+-------+-------
(0 rows)

test001=# 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)

test001=# INSERT INTO numeros (numero) VALUES (3);
INSERT 0 1

test001=# SELECT * from numeros;
 numero | cuadrado | cubo |  raiz2  |  raiz3
--------+----------+------+---------+---------
      2 |        4 |    8 | 1.41421 | 1.25992
      3 |        9 |   27 | 1.73205 | 1.44225
(2 rows)

test001=# UPDATE numeros SET numero = 4 WHERE numero = 3;
UPDATE 1

test001=# SELECT * from numeros;
 numero | cuadrado | cubo |  raiz2  |  raiz3
--------+----------+------+---------+---------
      2 |        4 |    8 | 1.41421 | 1.25992
      4 |       16 |   64 |       2 |  1.5874
(2 rows)
```

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 se puede 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 raiz cuadrada y la raiz 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()*.

Esta función retorna el valor NULL y esto significa, segun la regla 6.1 definida en este artículo, que para la fila afectada no se ejecutará el comanado DELETE. Por eso y mientras este disparador este instalado será imposible de borrar nada de la tabla *numeros*.

```
test001=# DELETE FROM numeros;
```
DELETE 0

```
test001=# SELECT * from numeros;
```
```
 numero | cuadrado | cubo |  raiz2  |  raiz3
--------+----------+------+---------+---------
      2 |        4 |    8 | 1.41421 | 1.25992
      4 |       16 |   64 |       2 |  1.5874
(2 rows)
```

Vamos a continuar complicando las cosas. Primero, vamos a desinstalar nuestros dos disparadores *proteger_datos* y *rellenar_datos*.

```
test001=# DROP TRIGGER proteger_datos ON numeros;
```
DROP TRIGGER

```
test001=# DROP TRIGGER rellenar_datos ON numeros;
```
DROP TRIGGER

A continuación crearemos un disparador único para las sentencias INSERT, UPDATE y DELETE. Este nuevo disparador utilizará una nueva función en la que tendremos que tener en cuenta que tipo de comando ha activado el disparador, si queremos retornar el valor correcto. Para ello utilizaremos la variable TG_OP.

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

   IF (TG_OP = 'INSERT' OR TG_OP = 'UPDATE' ) THEN

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

   ELSEIF (TG_OP = 'DELETE') THEN
     RETURN NULL;

   END IF;
 END;

$proteger_y_rellenar_datos$ LANGUAGE plpgsql;

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

La definición de nuestra tabla ha quedado asi:

```
test001=# \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:
   rellenar_datos BEFORE INSERT OR DELETE OR UPDATE ON numeros
      FOR EACH ROW EXECUTE PROCEDURE proteger_y_rellenar_datos()
```

Y todo seguirá funcionando de la misma manera que con los dos disparadores del comienzo:

```
test001=# SELECT * from numeros;
 numero | cuadrado | cubo |  raiz2  |  raiz3
--------+----------+------+---------+---------
      2 |        4 |    8 | 1.41421 | 1.25992
      4 |       16 |   64 |       2 |  1.5874
(2 rows)
```

```
test001=# INSERT INTO numeros (numero) VALUES (5);
INSERT 0 1
```

```
test001=# INSERT INTO numeros (numero) VALUES (6);
INSERT 0 1
```

```
test001=# SELECT * from numeros;
 numero | cuadrado | cubo |  raiz2  |  raiz3
--------+----------+------+---------+---------
      2 |        4 |    8 | 1.41421 | 1.25992
      4 |       16 |   64 |       2 |  1.5874
      5 |       25 |  125 | 2.23607 | 1.70998
      6 |       36 |  216 | 2.44949 | 1.81712
(4 rows)
```

```
test001=# UPDATE numeros SET numero = 10 WHERE numero = 6;
UPDATE 1
```

```
test001=# SELECT * from numeros ;
 numero | cuadrado | cubo |  raiz2  |  raiz3
--------+----------+------+---------+---------
      2 |        4 |    8 | 1.41421 | 1.25992
      4 |       16 |   64 |       2 |  1.5874
      5 |       25 |  125 | 2.23607 | 1.70998
     10 |      100 | 1000 | 3.16228 | 2.15443
(4 rows)
```

```
test001=# DELETE FROM numeros where numero =10;
DELETE 0
```

```
test001=# SELECT * from numeros;
 numero | cuadrado | cubo |  raiz2  |  raiz3
--------+----------+------+---------+---------
      2 |        4 |    8 | 1.41421 | 1.25992
      4 |       16 |   64 |       2 |  1.5874
      5 |       25 |  125 | 2.23607 | 1.70998
     10 |      100 | 1000 | 3.16228 | 2.15443
(4 rows)
```

Por último y antes de terminar, vamos a definir un disparador del tipo statement-level que se ejecute despues de nuestras sentencias INSERT, UPDATE y DELETE. La función ejecutada por este disparador grabará datos de la ejecución en la tabla *cambios* (esto no sirve para mucho en la vida real, pero como ejemplo esta bien para ver como funciona).

Para demostrar como podemos utilizar esto vamos a definir una nueva tabla:

```
CREATE TABLE cambios(
  timestamp_ TIMESTAMP WITH TIME ZONE default NOW(),
  nombre_disparador text,
  tipo_disparador text,
  nivel_disparador text,
  comando text
);
```

La función la podemos definir asi:

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

    INSERT INTO cambios (
            nombre_disparador,
            tipo_disparador,
            nivel_disparador,
            comando)
       VALUES (
            TG_NAME,
            TG_WHEN,
            TG_LEVEL,
            TG_OP
           );

    RETURN NULL;
  END;

$grabar_operaciones$ LANGUAGE plpgsql;
```

Y el disparador lo instalaríamos de la siguiente forma:

```
CREATE TRIGGER grabar_operaciones AFTER INSERT OR UPDATE OR DELETE
    ON numeros FOR EACH STATEMENT
    EXECUTE PROCEDURE grabar_operaciones();
```

La definición de nuestra tabla quedaría asi:

```
test001=# \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:
    grabar_operaciones AFTER INSERT OR DELETE OR UPDATE ON numeros
        FOR EACH STATEMENT EXECUTE PROCEDURE grabar_operaciones()
    proteger_y_rellenar_datos BEFORE INSERT OR DELETE OR UPDATE ON numeros
        FOR EACH ROW EXECUTE PROCEDURE proteger_y_rellenar_datos()
```

A continuación se puede ver como funcionaría:

```
test001=# SELECT * from cambios ;
 timestamp_ | nombre_disparador | tipo_disparador | nivel_disparador | comando
------------+-------------------+-----------------+------------------+---------
(0 rows)

test001=# INSERT INTO numeros (numero) VALUES (100);
INSERT 0 1

test001=# SELECT * from numeros ;
 numero | cuadrado |  cubo   |  raiz2  |  raiz3
--------+----------+---------+---------+---------
      2 |        4 |       8 | 1.41421 | 1.25992
      4 |       16 |      64 |       2 |  1.5874
      5 |       25 |     125 | 2.23607 | 1.70998
     10 |      100 |    1000 | 3.16228 | 2.15443
    100 |    10000 | 1000000 |      10 | 4.64159
(5 rows)

test001=# SELECT * from cambios ;
          timestamp_           | nombre_disparador  | tipo_disparador | nivel_disparador | comando
-------------------------------+--------------------+-----------------+------------------+---------
 2009-06-11 23:05:29.794534+02 | grabar_operaciones | AFTER           | STATEMENT        | INSERT
(1 row)

test001=# UPDATE numeros SET numero = 1000 WHERE numero = 100;
UPDATE 1

test001=# SELECT * from numeros ;
 numero | cuadrado |    cubo    |  raiz2  |  raiz3
--------+----------+------------+---------+---------
      2 |        4 |          8 | 1.41421 | 1.25992
      4 |       16 |         64 |       2 |  1.5874
      5 |       25 |        125 | 2.23607 | 1.70998
     10 |      100 |       1000 | 3.16228 | 2.15443
   1000 |  1000000 | 1000000000 | 31.6228 |      10
(5 rows)

test001=# SELECT * from cambios ;
          timestamp_           | nombre_disparador  | tipo_disparador | nivel_disparador | comando
-------------------------------+--------------------+-----------------+------------------+---------
 2009-06-11 23:05:29.794534+02 | grabar_operaciones | AFTER           | STATEMENT        | INSERT
 2009-06-11 23:06:08.259421+02 | grabar_operaciones | AFTER           | STATEMENT        | UPDATE
(2 rows)
```

```
test001=# DELETE FROM numeros where numero =1000;
```
DELETE 0

```
test001=# SELECT * from numeros ;
```

```
 numero | cuadrado |    cubo    |  raiz2  |  raiz3
--------+----------+------------+---------+---------
      2 |        4 |          8 | 1.41421 | 1.25992
      4 |       16 |         64 |       2 |  1.5874
      5 |       25 |        125 | 2.23607 | 1.70998
     10 |      100 |       1000 | 3.16228 | 2.15443
   1000 |  1000000 | 1000000000 | 31.6228 |      10
(5 rows)
```

```
test001=# SELECT * from cambios ;
```

```
          timestamp_           | nombre_disparador  | tipo_disparador | nivel_disparador | comando
-------------------------------+--------------------+-----------------+------------------+---------
 2009-06-11 23:05:29.794534+02 | grabar_operaciones | AFTER           | STATEMENT        | INSERT
 2009-06-11 23:06:08.259421+02 | grabar_operaciones | AFTER           | STATEMENT        | UPDATE
 2009-06-11 23:06:26.568632+02 | grabar_operaciones | AFTER           | STATEMENT        | DELETE
(3 rows)
```

ALGUNAS ANOTACIONES

Cuando se crea la función del trigger, ésta no puede tener parámetros, sin embargo el triger como tal sí.

Cuando un disparador se define, los argumentos pueden ser especificados para ello. El propósito de la inclusión de argumentos en la definición del trigger es permitir a los diferentes factores desencadenantes con requisitos similares llamar a la misma función. Como ejemplo, podría ser una función de activación generalizada que toma como argumentos dos nombres de columna y pone el usuario actual en uno y la marca de tiempo actual en la otra.

Para tener acceso a los parámetros del trigger se utiliza la variable TG_ARGV[ ], la cual es un arreglo de texto como se mencionó anteriormente.

Ejemplo:

```
-- Crea una tabla para almacenar los datos
CREATE TABLE testing(
  valor BIGINT NOT NULL,
  argumento1 VARCHAR,
  argumento2 VARCHAR,
  PRIMARY KEY (valor)
);
```

```
-- Crea el stored procedure
CREATE OR REPLACE FUNCTION sp_con_argumentos() RETURNS TRIGGER AS
$sp_con_argumentos$
 DECLARE
 BEGIN

   IF (TG_OP = 'INSERT' OR TG_OP = 'UPDATE' ) THEN

     NEW.argumento1 := TG_ARGV[0];
     NEW.argumento2 := TG_ARGV[1];
     RETURN NEW;

   ELSEIF (TG_OP = 'DELETE') THEN
     RETURN NULL;

   END IF;
 END;

$sp_con_argumentos$ LANGUAGE plpgsql;

-- Crea el trigger
CREATE TRIGGER sp_con_argumentos BEFORE INSERT OR UPDATE OR DELETE
   ON testing FOR EACH ROW
   EXECUTE PROCEDURE sp_con_argumentos('LOOK1', 'LOOK2');

-- Inserta el primer valor, por ende se llama al triggers
INSERT INTO testing(valor) VALUES(1);

-- Se observan los valores de la tabla
SELECT * FROM testing;
```

Ejemplos de triggers sobre la base de datos de ejemplo en el archivo trigger.sql de los materiales adicionales.

HERENCIA

Herencia sirve para modelar casos en que una tabla es una especialización de otra(s).

Para estudiar este tema utilizaremos las siguientes tablas:

Hay ciudades que no son capitales:

```
CREATE TABLE cities (
   name         text,
   population     float,
   altitude       int     -- in feet
);

CREATE TABLE capitals (
   state         char(2)
) INHERITS (cities);

INSERT INTO cities VALUES ('San Francisco', 7.24E+5, 63);
INSERT INTO cities VALUES ('Las Vegas', 2.583E+5, 2174);
INSERT INTO cities VALUES ('Mariposa', 1200, 1953);

INSERT INTO capitals VALUES ('Sacramento', 3.694E+5, 30, 'CA');
INSERT INTO capitals VALUES ('Madison', 1.913E+5, 845, 'WI');

SELECT * FROM cities;
SELECT * FROM capitals;
```

En PostgreSQL, una tabla puede heredar de cero o mas tablas, y una consulta puede referenciar ya sea, todas la filas de una tabla así como todas las filas de una tabla mas todas la filas de sus tablas descendientes.

Ejemplo:

```
SELECT name, altitude
    FROM cities
    WHERE altitude > 500;
```

Esto retorna:

```
   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
```

La siguiente consulta encuentra todas las ciudades que no son capitales de estados y que se situan a una altitud de 500 pies:

```
SELECT name, altitude
    FROM ONLY cities
    WHERE altitude > 500;

   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
```

ONLY:

La opción only solo puede ser utilizada en comandos Insert, Update o Delete.

Para saber de cual tabla se origina el resultado de la consulta:

```
SELECT c.tableoid, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
 tableoid |   name    | altitude
----------+-----------+----------
   139793 | Las Vegas |     2174
   139793 | Mariposa  |     1953
   139798 | Madison   |      845
```

Para ver los nombres de las tablas:

```
SELECT p.relname, c.name, c.altitude
FROM cities c, pg_class p
WHERE c.altitude > 500 and c.tableoid = p.oid;
relname  |   name    | altitude
----------+-----------+----------
 cities   | Las Vegas |     2174
 cities   | Mariposa  |     1953
 capitals | Madison   # apt-get install postgresql-contrib|      845
```

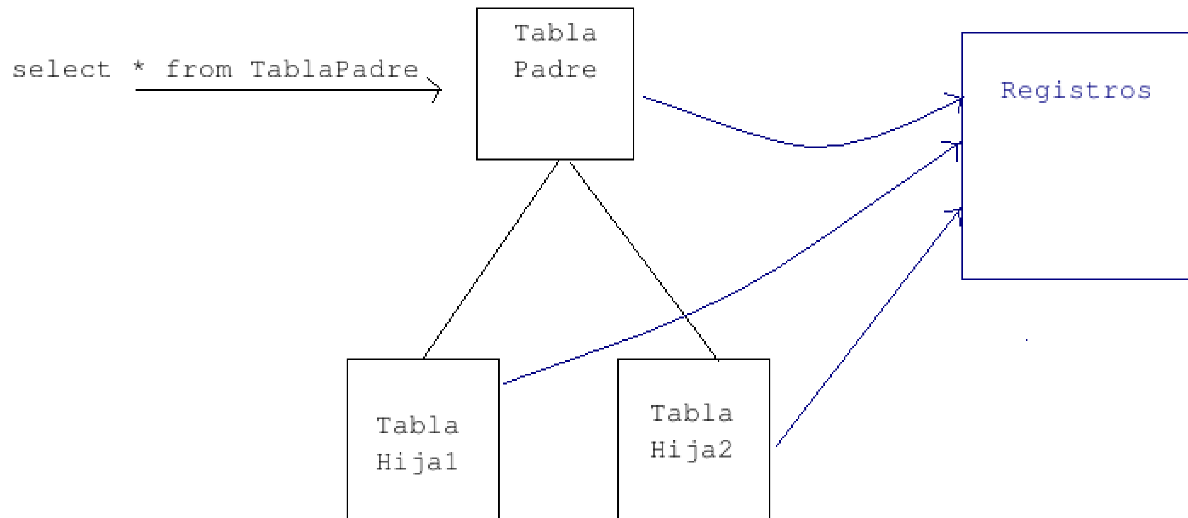
No propaga los inserts automáticamente:

```
INSERT INTO cities (name, population, altitude, state)
VALUES ('New York', NULL, NULL, 'NY');
```

Todos los constrains check y not-null constraints son heredados automáticamente.otros tipos de constraints (unique, primary key, and foreign key constraints) no.

Una tabla puede heredar de más de una tabla padre, en cuyo caso se tiene la unión de las columnas definidas por las tablas padres. Cualquier columna declarada en la definición de la tabla hija se suma a estas. Si el mismo nombre de columna aparece en las columnas de los padres, o en ambas una tabla padre y la tabla hija, entonces, estas columnas se combinan para que sólo haya una columna de este tipo en la tabla hija. Para que se combinen, las columnas deben tener el mismo tipo de datos, de lo contrario se produce un error. La columna fusionada tendrá copias de todas las restricciones check procedentes de cualquiera de las definiciones de las columnas de donde viene la columna, y será marcada not-null si alguna de las definiciones está así marcada.

Como se puede observar del ejemplo anterior postgres maneja la herencia de la siguiente manera:



Cuando se hace un select de la tabla padre, postgres devolverá todos los registros que coincidan con la cláusula select tanto de la tabla padre como de las tablas hijas que ésta tenga.

Si se usa Only, solamente devolverá los registros de la tabla padre.

Si un registro se encuentra en la tabla padre y en la tabla hija, postgres lo devolverá dos veces en los resultados del select.

Particionamiento de Tablas:

Se basa en herencia.

Se tiene una tabla padre, que por lo general no tiene datos y solamente se usa para unir las tablas particionadas, las cuales consisten en sus tablas hijas.

Pasos:

1. Crear las tablas hijas, agregarles un constraint de check para que no permitan insertar datos que no cumplan con el criterio de particionado.
2. Por medio de select into, poblar las tablas hijas uitlizando en where el constraint check de la tabla.
3. Crear índices en las tablas hijas sobre los campos utilizados como criterios de partición.
4. Crear constraints de llave primaria y otros si así se desea sobre las tablas hijas, ya que estos **no** se heredan.
5. Borrar con "only" los datos de la tabla padre para que no estén duplicados.
6. Crear un trigger que se encarge de redireccionar los inserts de la tabla padre a la tabla hija correspondiente.

Importante:
Para optimizar las consultas verificar que en el archivo postgres.conf esté activada la opción ***constraint_exclusion,*** la cual se encarga de antes de hacer una búsqueda sobre una tabla, verificar que los criterios de búsqueda coincidan con los contraint check de la tabla.[5]

Ejemplo

Particionamiento de pagos y desembolsos

Crear tablas hijas:

```
create table clie.pagos_y_desembolsos_men2000_cl ( check (fec_pago < '1/1/2000')) inherits (clie.pagos_y_desembolsos_cl);
create table clie.pagos_y_desembolsos_2000_cl ( check (fec_pago between '1/1/2000' and '31/12/2000')) inherits (clie.pagos_y_desembolsos_cl);
create table clie.pagos_y_desembolsos_2001_cl ( check (fec_pago between '1/1/2001' and '31/12/2001')) inherits (clie.pagos_y_desembolsos_cl);
create table clie.pagos_y_desembolsos_2002_cl ( check (fec_pago between '1/1/2002' and '31/12/2002')) inherits (clie.pagos_y_desembolsos_cl);
create table clie.pagos_y_desembolsos_2003_cl ( check (fec_pago between '1/1/2003' and '31/12/2003')) inherits (clie.pagos_y_desembolsos_cl);
```

Trasladar los datos de la tabla padre a las tablas hijas.

```
insert into clie.pagos_y_desembolsos_men2000_cl select * from clie.pagos_y_desembolsos_cl where fec_pago < '1/1/2000';
insert into clie.pagos_y_desembolsos_2000_cl select * from clie.pagos_y_desembolsos_cl where fec_pago between '1/1/2000' and '31/12/2000';
insert into clie.pagos_y_desembolsos_2001_cl select * from clie.pagos_y_desembolsos_cl where fec_pago between '1/1/2001' and '31/12/2001';
insert into clie.pagos_y_desembolsos_2002_cl select * from clie.pagos_y_desembolsos_cl where fec_pago between '1/1/2002' and '31/12/2002';
insert into clie.pagos_y_desembolsos_2003_cl select * from clie.pagos_y_desembolsos_cl where fec_pago between '1/1/2003' and '31/12/2003';
```

5 Para la versión 8.4 de postgres la opción que hay que activar es **partition.**

Crear índices en las tablas hijas:

```
CREATE INDEX pagos_y_desembolsos_men2000_fec_pago ON clie.pagos_y_desembolsos_men2000_cl (fec_pago);
CREATE INDEX pagos_y_desembolsos_2000_fec_pago ON clie.pagos_y_desembolsos_2000_cl (fec_pago);
CREATE INDEX pagos_y_desembolsos_2001_fec_pago ON clie.pagos_y_desembolsos_2001_cl (fec_pago);
CREATE INDEX pagos_y_desembolsos_2002_fec_pago ON clie.pagos_y_desembolsos_2002_cl (fec_pago);
CREATE INDEX pagos_y_desembolsos_2003_fec_pago ON clie.pagos_y_desembolsos_2003_cl (fec_pago);
CREATE INDEX pagos_y_desembolsos_2004_fec_pago ON clie.pagos_y_desembolsos_2004_cl (fec_pago);
```

Borrar los datos de la tabla padre:

```
delete from only clie.pagos_y_desembolsos_cl;
```

Crear trigger de inserciones:

```
CREATE OR REPLACE FUNCTION clie.pagos_y_desembolsos_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
   IF ( NEW.fec_pago < '1/1/2000') THEN
      INSERT INTO clie.pagos_y_desembolsos_men2000_cl VALUES (NEW.*);
   ELSIF ( NEW.fec_pago between '1/1/2000' AND '31/12/2000' ) THEN
      INSERT INTO clie.pagos_y_desembolsos_2000_cl VALUES (NEW.*);
   ELSIF ( NEW.fec_pago between '1/1/2001' AND '31/12/2001' ) THEN
      INSERT INTO clie.pagos_y_desembolsos_2000_cl VALUES (NEW.*);
   ELSIF ( NEW.fec_pago between '1/1/2002' AND '31/12/2002' ) THEN
      INSERT INTO clie.pagos_y_desembolsos_2000_cl VALUES (NEW.*);
   ELSIF ( NEW.fec_pago between '1/1/2003' AND '31/12/2003' ) THEN
      INSERT INTO clie.pagos_y_desembolsos_2000_cl VALUES (NEW.*);
   ELSE
      RAISE NOTICE 'Fecha fuera de rango %. Ojo porque no se introduce',NEW.fec_pago;
   END IF;
   RETURN NULL;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER clie.insert_pagos_y_desembolsos_trigger
   BEFORE INSERT ON clie.pagos_y_desembolsos_cl
   FOR EACH ROW EXECUTE PROCEDURE clie.pagos_y_desembolsos_insert_trigger();
```

Ejemplos de particionamiento sobre la base de datos de ejemplo en el archivo particionamiento.sql de los materiales adicionales.

Consultas entre bases de datos

Las consultas entre dos bases de datos se llevan a cabo por medio de dblink.

Para instalar esta característica:
Instalar postgresql-contrib :

```
# apt-get install postgresql-contrib
```

Correr en la base de datos (o mejor en la template) el script

```
#psql template1 -U postgres
template1# \i /usr/share/postgresql/8.3/contrib/dblink.sql
```

Ejemplos de consultas:

El siguiente ejemplo inserta en la tabla carrera de la actual base de datos los resultados del select sobre la tabla carrera de la base de datos hostorico del host **adacad.academica.ues.edu.sv**.

Nótese que se debe indicar el usuario y password para realizar la conexión.

INSERT INTO carrera SELECT *
FROM dblink(
'host=adacad.academica.ues.edu.sv
dbname=historico user=publico password=publico',
'select * from carrera where codigo=\'P70431\' and plan_estudios=1998'
)
AS resultados(codigo text, plan smallint, tipo text, nombre text,
cod_uacad text, cod_uadmin text) ;

Es importante destacar que para que el usuario pueda acceder a la base de datos remota, éste debe tener la entrada de acceso correspondiente en el archivo de configuración pg_hba.conf.

I**NSERT INTO imagen**
SELECT *
FROM dblink(
'host=adacad.academica.ues.edu.sv
dbname=ni_2006 user=publico password=publico',
'
select carnet,\'foto\',\'jpg\',fotografia,\'Fotografía de el/la alumno/a\',now() from aspirante_ingreso,solicita_primer_ingreso where numero_formulario=numero_formulario_aspirante_ing and estado_seleccion=\'S\' and fotografia is not null
'
)
AS resultados(carnet text, denominacion text, formato text, datos bytea, descripcion text, fecha date)
WHERE carnet in (SELECT carnet from alumno);

```
INSERT INTO imagen
SELECT *
FROM dblink(
'host=adacad.academica.ues.edu.sv
dbname=ni_2007 user=publico password=publico',
'
select carnet,\'foto\',\'jpg\',fotografia,\'Fotografía de el/la alumno/a\',now() from aspirante_ingreso,solicita_primer_ingreso where numero_formulario=numero_formulario_aspirante_ing and estado_seleccion=\'S\' and fotografia is not null
'
)
AS resultados(carnet text, denominacion text, formato text, datos bytea, descripcion text, fecha date)
WHERE carnet in (SELECT carnet from alumno);
```

Se pueden dejar abiertas conexiones cuando se va a hacer más de una consulta por conexión:
dblink_connect('nombredelaconexion','stringdeconexion');

para desconectarlo
dblink_disconnect(nombredelaconexion);

Ejemplo:

```
select dblink_connect('dbname=postgres');
  dblink_connect
 ----------------
  OK
 (1 row)

 select * from dblink('select proname, prosrc from pg_proc')
  as t1(proname name, prosrc text) where proname like 'bytea%';
   proname   |   prosrc
 ------------+------------
  byteacat   | byteacat
  byteaeq    | byteaeq
  bytealt    | bytealt
  byteale    | byteale
  byteagt    | byteagt
  byteage    | byteage
  byteane    | byteane
  byteacmp   | byteacmp
  bytealike  | bytealike
  byteanlike | byteanlike
  byteain    | byteain
  byteaout   | byteaout
 (12 rows)

 select dblink_disconnect('myconn');
  dblink_disconnect
 ----------------
  OK
```

Para ejecutar comandos:

```
 select dblink_exec('insert into foo values(21,''z'',''{"a0","b0","c0"}'');');
```

Se pueden abrir varias conexiones simultáneamente, solamente se debe tener cuidado de indicarle a dblink y dblink_exec sobre cual conexión se está trabajando:

```
select dblink_connect('dbname=dblink_test_slave');
  dblink_connect
 ----------------
  OK
 (1 row)

 select dblink_exec('insert into foo values(21,''z'',''{"a0","b0","c0"}'');');
    dblink_exec
 -----------------
  INSERT 943366 1
 (1 row)

 select dblink_connect('myconn', 'dbname=regression');
  dblink_connect
 ----------------
  OK
 (1 row)

 select dblink_exec('myconn', 'insert into foo
values(21,''z'',''{"a0","b0","c0"}'');');
    dblink_exec
 ------------------
  INSERT 6432584 1
 (1 row)
```

Tablas temporales

Las tablas temporales se crean utilizando la opción temp:

Ejemplo:

```
CREATE TEMP TABLE prueba(ciud_id integer, ciud_codigo numeric, ciud_nombre
varchar);

O llenándola directamente con una consulta:
select name as id,'Ced' as codigo_tipoid,description as nombre,null as notas,1
as maxdocs,smodified as fecha_crea,smodified as fecha_modif into temp expediente
from folders where length(name)<=10;
```

Archivos CSV

Para llenar una tabla a partir de un archivo csv(valores separados por coma), utilizamos el comando \copy, por ejemplo:
\copy tcconsultas from '/home/usuario/ESS002-2.txt' DELIMITER AS ',' NULL as '';

El ejemplo anterior carga sobre la tabla tcconsultas los datos que estén guardados en el archivo ESS002.txt

Para enviar a un archivo csv los datos de una tabla:
\copy expediente to '/home/usuario/expediente.csv' DELIMITER AS ',' NULL as '';

CAPÍTULO 3

ADMINISTRACIÓN
Y
MANTENIMIENTO
DE LAS BASES DE DATOS

Herramientas gráficas para la administración de bases de datos en PostgreSQL

Para administrar las bases de datos contamos con la terminal interactiva, pero además contamos con dos valiosas herramientas gráficas que nos pueden llegar a facilitar el trabajo, ellas son:

- pgadmin3: es una herramienta muy completa.
- DbVisualizer: tiene una funcionalidad que pgadmin3 no tiene(genera el diagrama entidad relación de la base de datos)

PGADMIN3

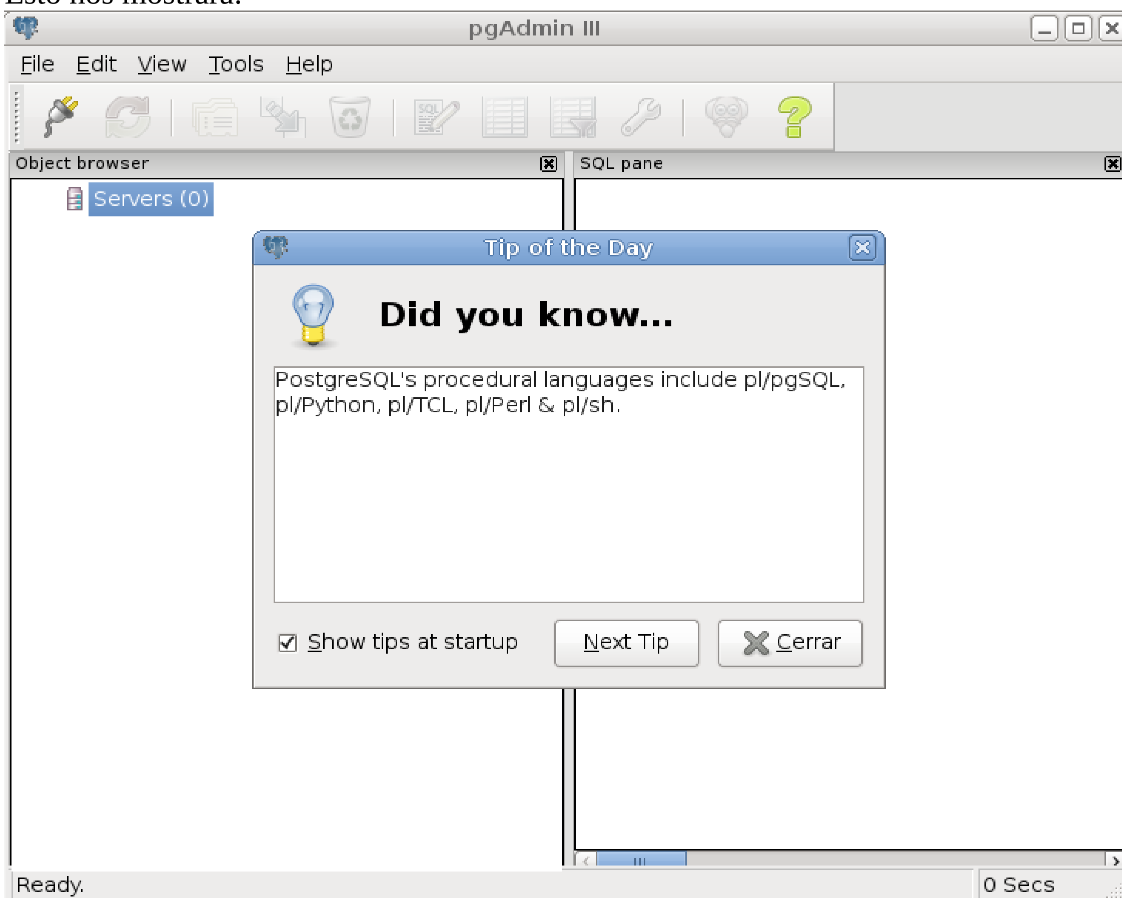
Para instalar pgadmin3 utilizaremos el siguiente comando como usuario root:

```
#apt-get install pgadmin3
```

Una vez que se ha instalado pgadmin3 lo podemos utilizar con el comando:

```
$pgadmin3 &
```

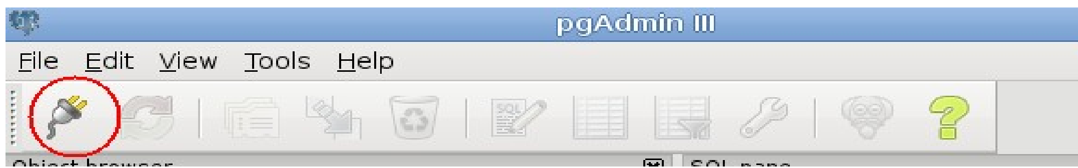
Esto nos mostrará:



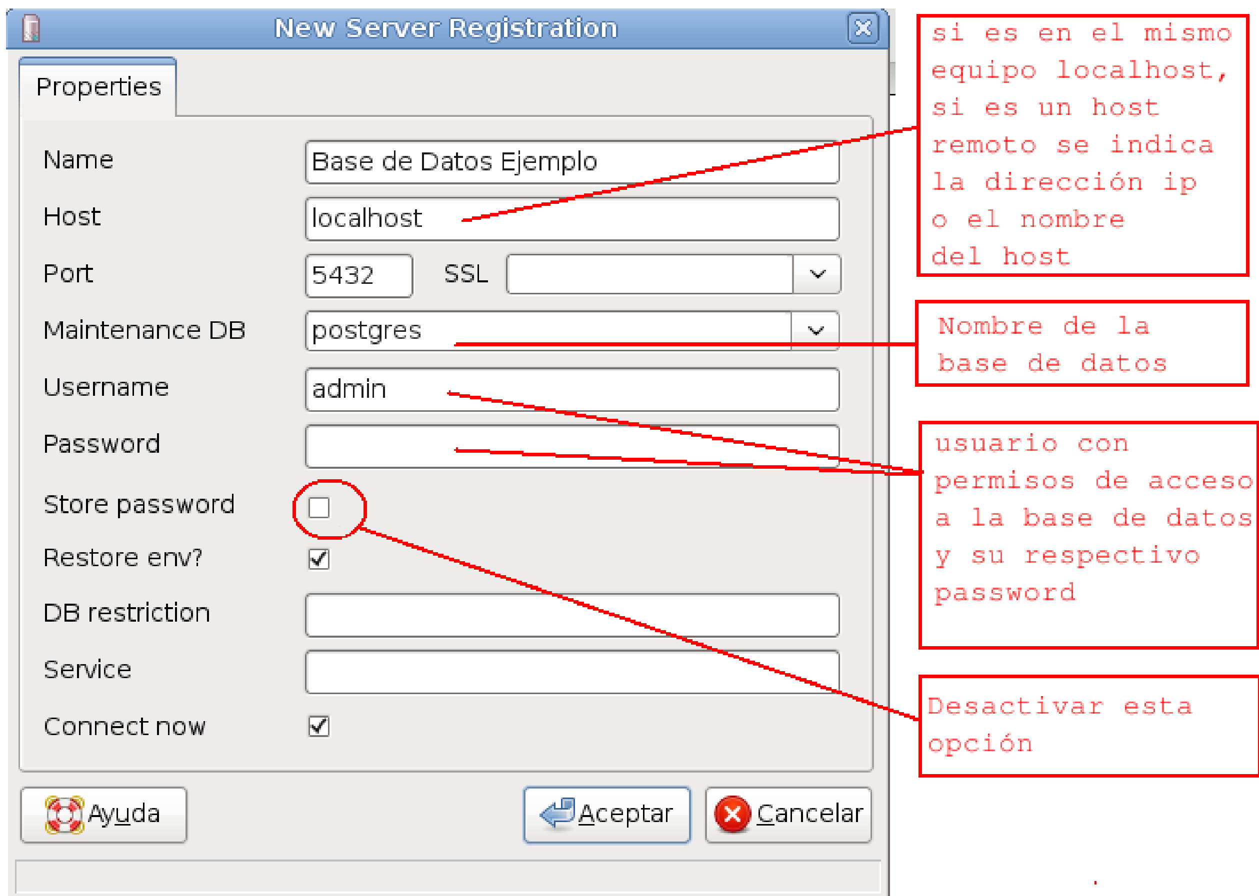
Hacemos clic en cerrar, podemos desactivar que muestre tips al iniciar si así lo deseamos.

A continuación vamos a crear la conexión a la base de datos. Podemos configurar cuantas conexiones deseemos:

Hacemos clic en el ícono que se muestra a continuación:

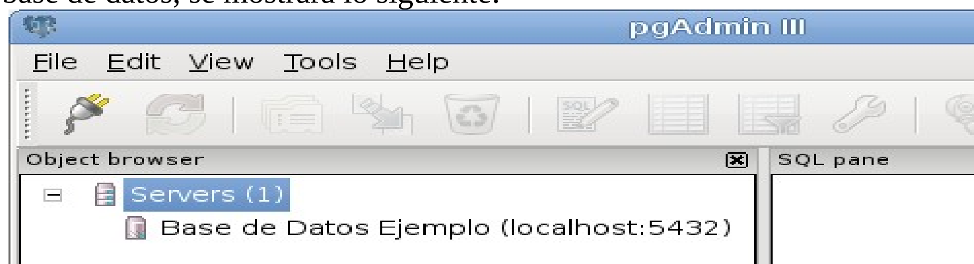


Se mostrará la siguiente caja de diálogo:

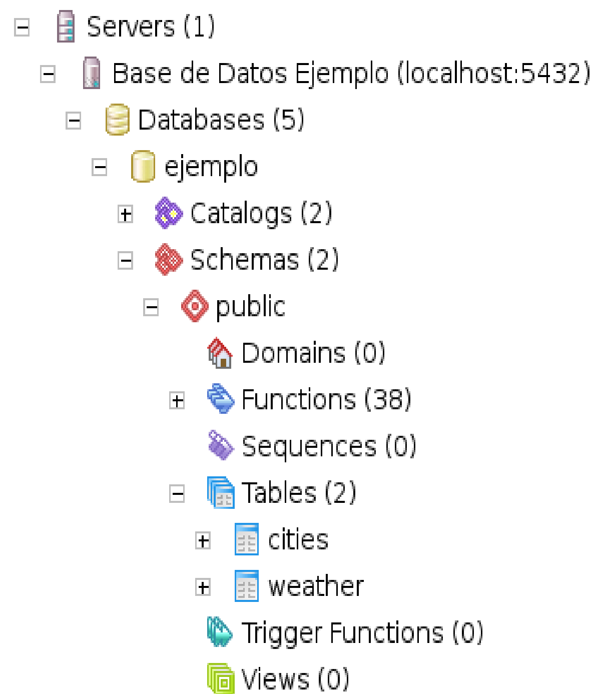


Es muy importante desactivar la opción de guardar passwords(Store password), ya que si se deja activada en la carpeta /home/usuario se creará un archivo llamado **.pgpass** oculto, el cual contiene en texto plano, el nombre de la base de datos, el usuario y el **password!!!**

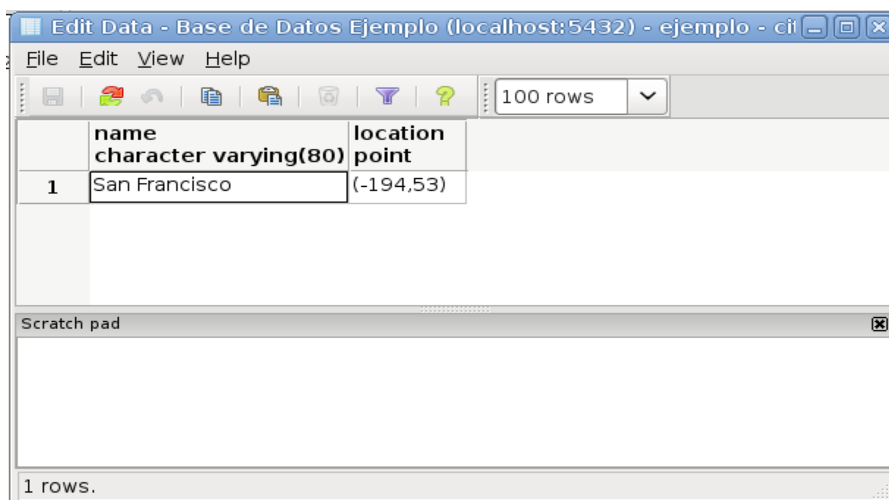
Hacemos clic en Aceptar, si todos los datos ingresados fueron correctos y tenemos permisos de acceso a la base de datos, se mostrará lo siguiente:



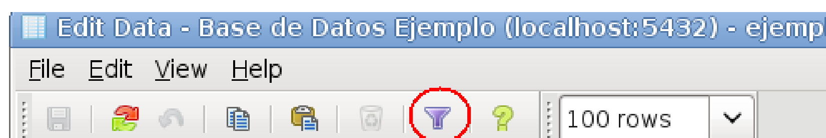
Haciendo clic sobre la conexión que acabamos de crear vamos a ver las bases de datos de esa conexión, pero solo vamos a tener acceso a las bases de datos a las que tenga permiso el usuario:



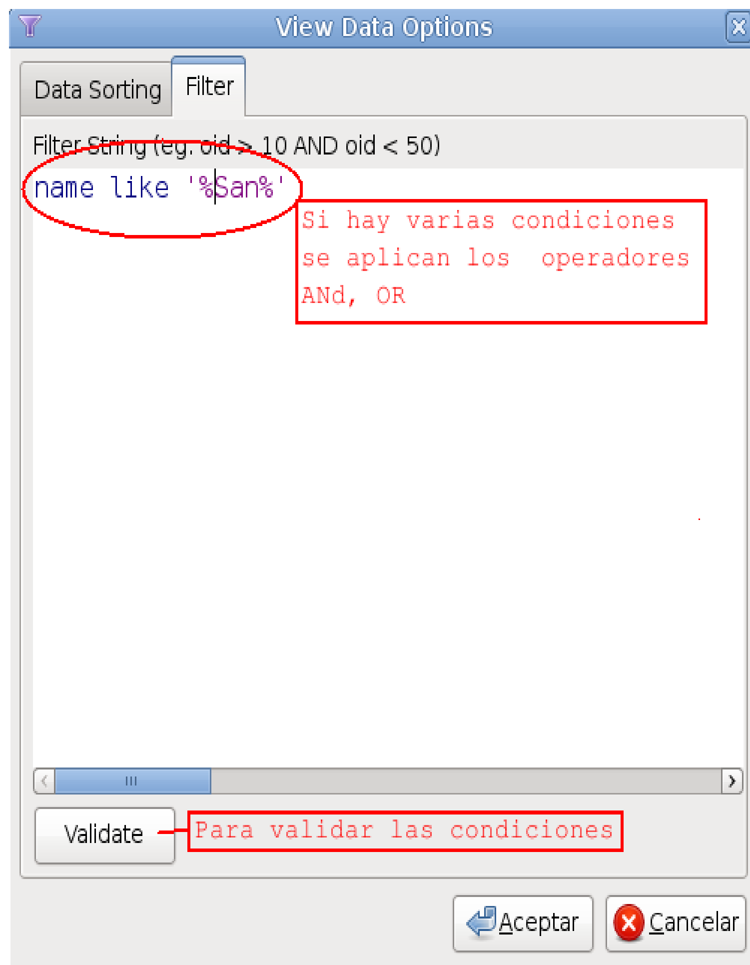
Operaciones sobre tablas:
Haciendo clic derecho sobre una tabla se mostrará un menú, entre sus opciones tenemos: View Data-> View top 100 rows, la cual nos muestra el contenido de la tabla, y nos permite hacer inserciones, borrados y modificaciones de datos.



Se pueden aplicar filtros de búsqueda sobre los datos de la tabla con el ícono:



Se mostrará la siguiente caja de diálogo:



Una vez que validamos las condiciones hacemos clic en Aceptar y se mostrarán los resultados.

La herramienta provee muchísimas funcionalidades, se insta al estudiante a estudiarlas!

DBVISUALIZER

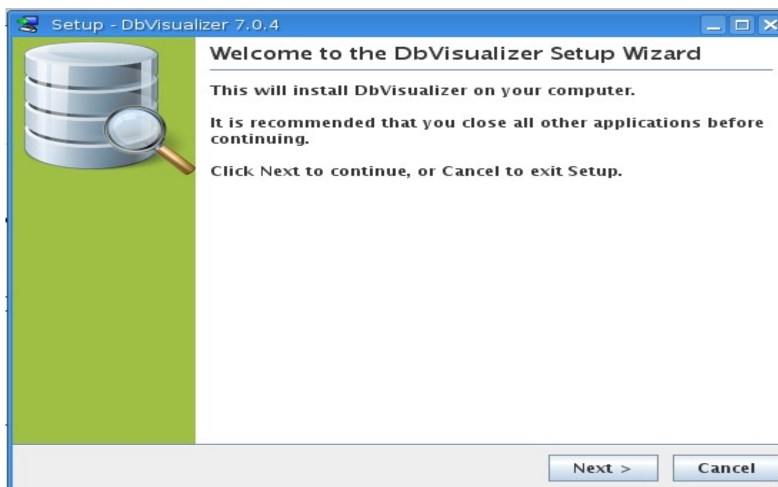
Verificar que esté Java instalado.

Descargarlo de:
http://www.dbvis.com/products/dbvis/download/

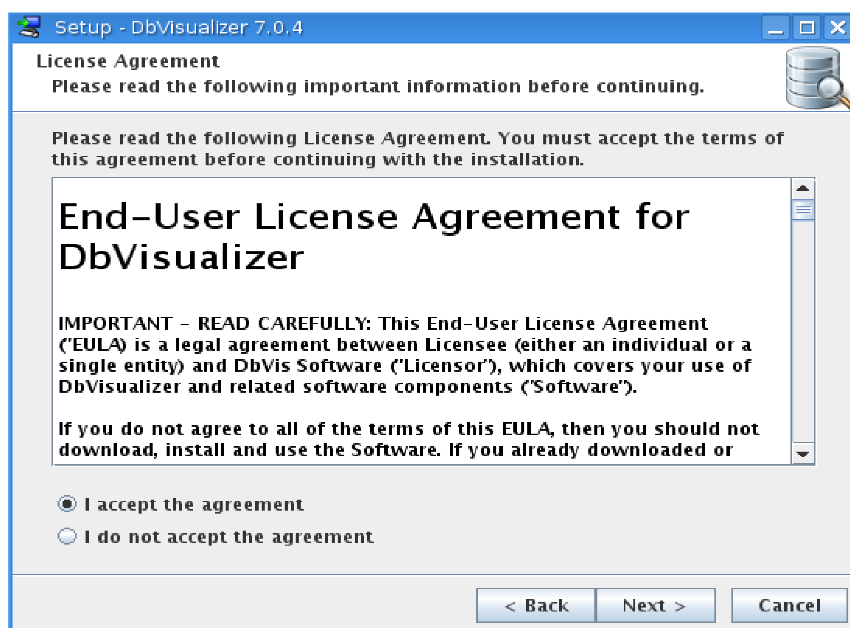
En consola, como usuario root:

```
#sh dbvis_linux_7_0_4.sh
```

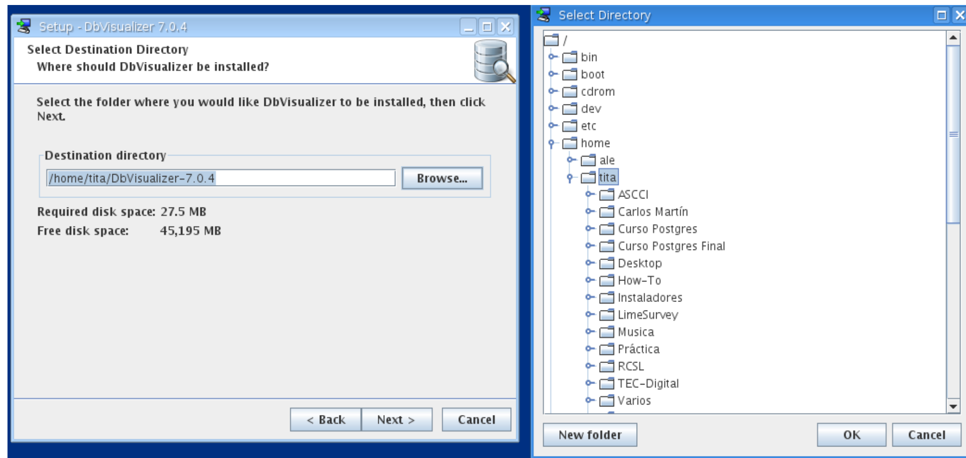
Se iniciará el instalador, hacer clic en next.



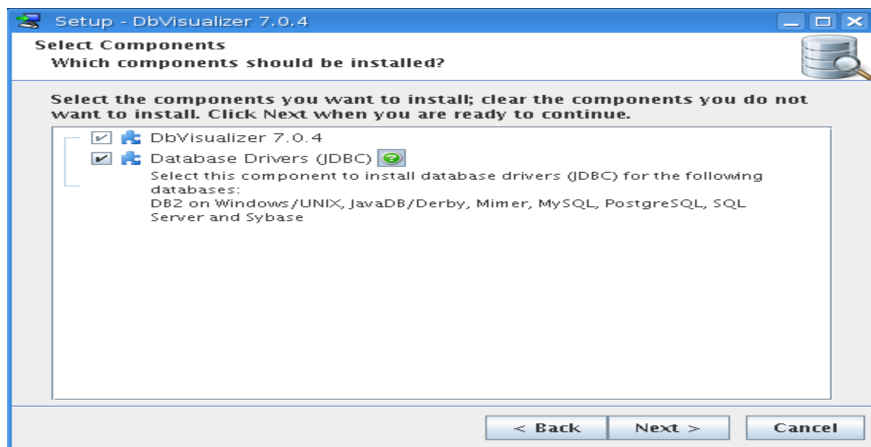
Hacer clic en “I accept ...” y seguidamente en next.



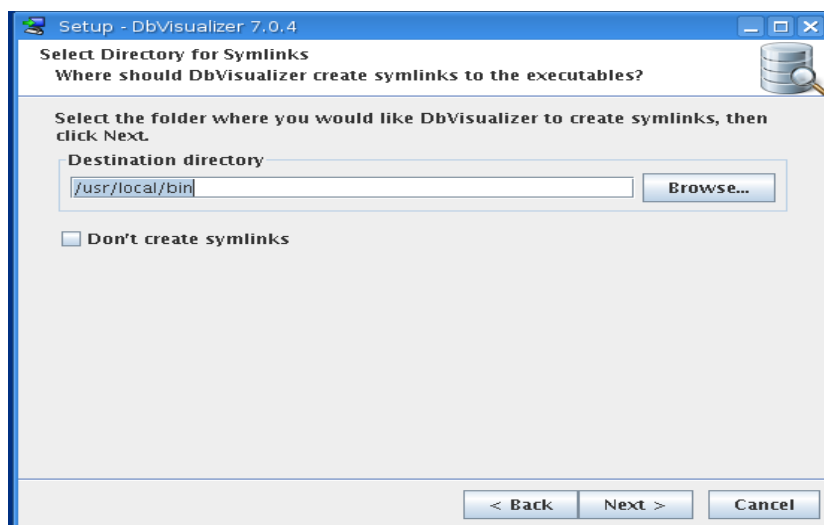
Escoger en “Browse” donde desea instalar dbvis y click en next.



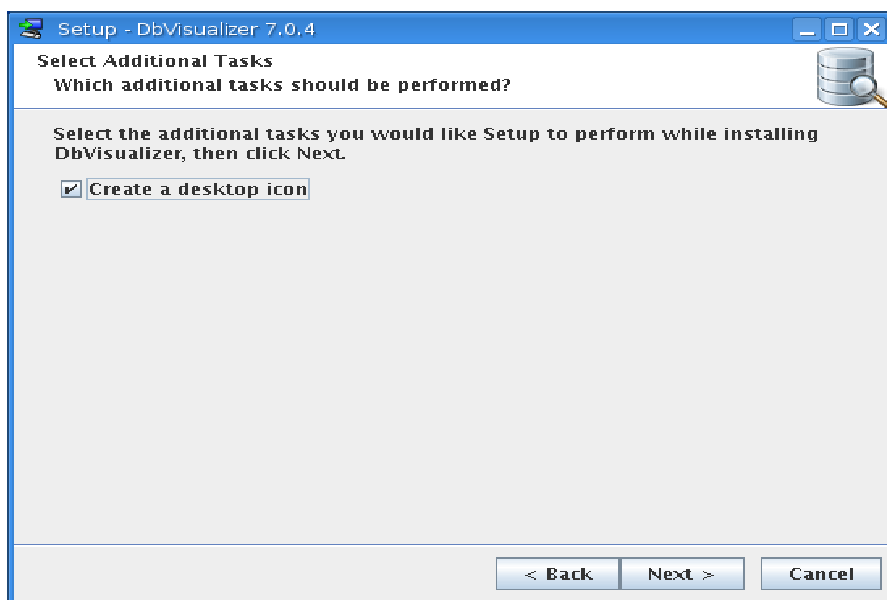
Clic en next.



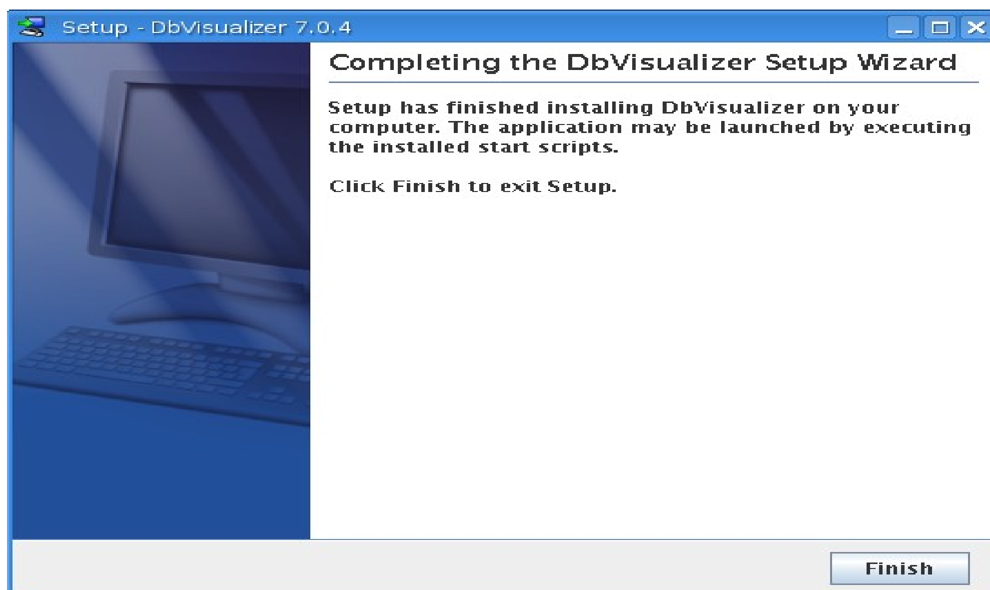
Clic en next.



Si lo desea marque la opción de crear ícono en el Escritorio y haga clic en next.



Clic en “Finish” y listo, dbvis está instalado.



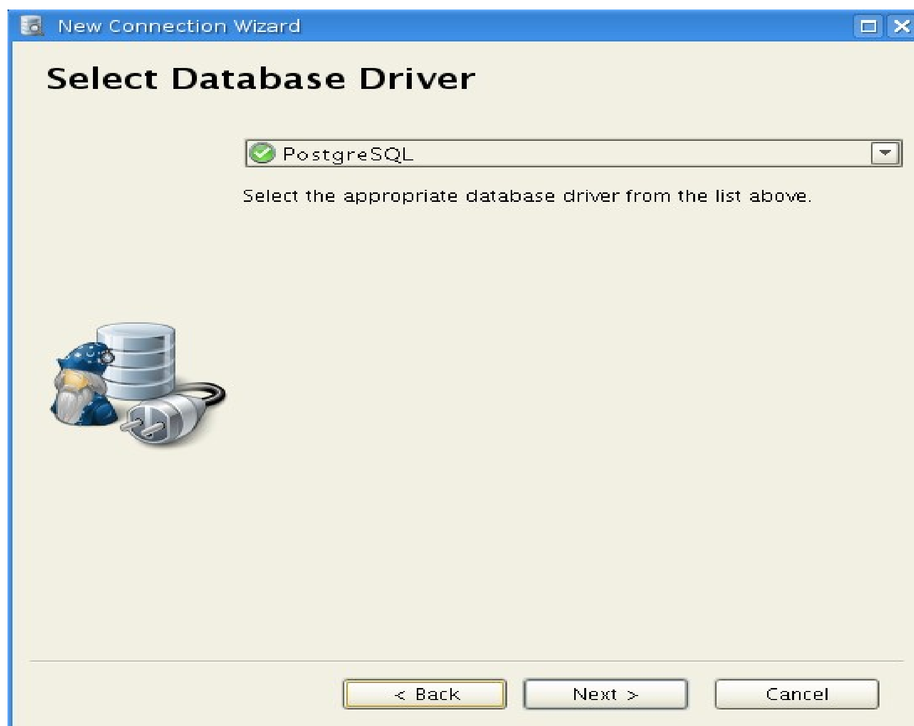
Utilizando DbVisualizer:

Para iniciarlo:

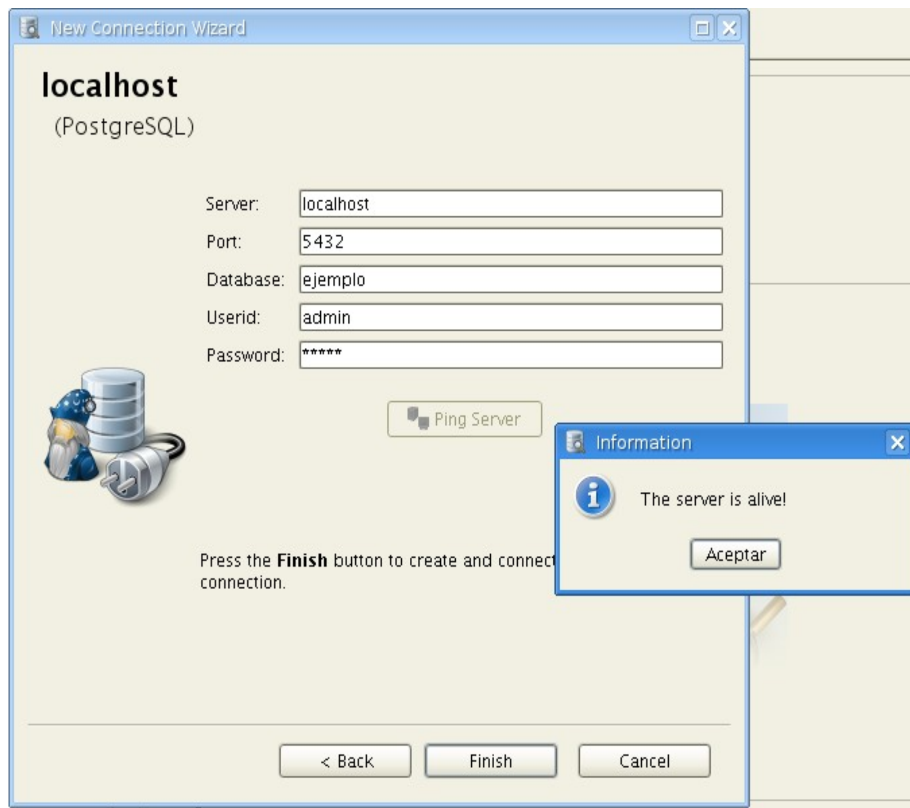
```
$/var/opt/DbVisualizer-7.0.4/dbvis &
```

Escribimos el nombre que deseamos para la conexión.

Escogemos el driver para PostgreSQL y hacemos clic en next.

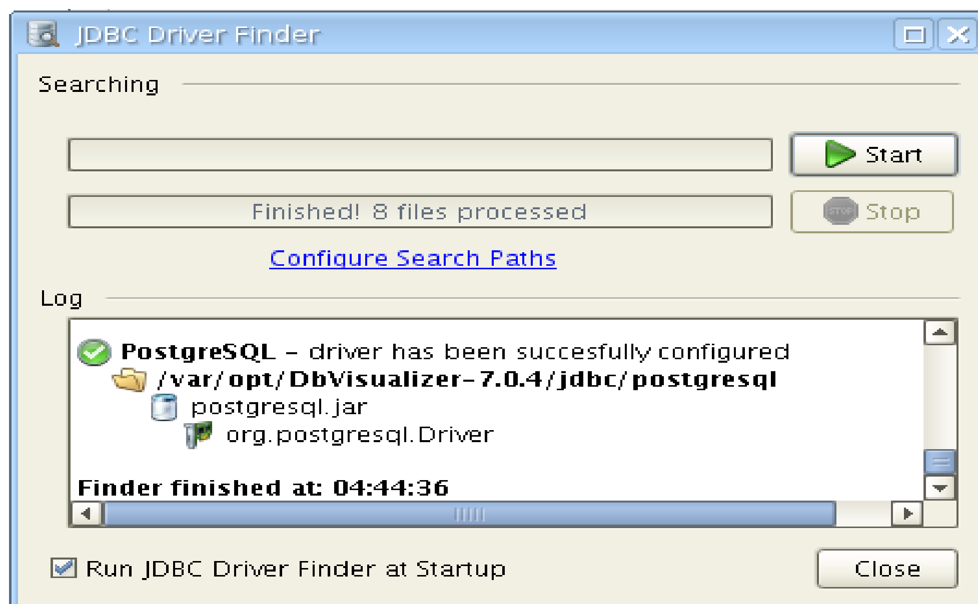


Escribir la información de conexión. Hacer clic en “Ping Server” para verificar que la conexión esté bien.

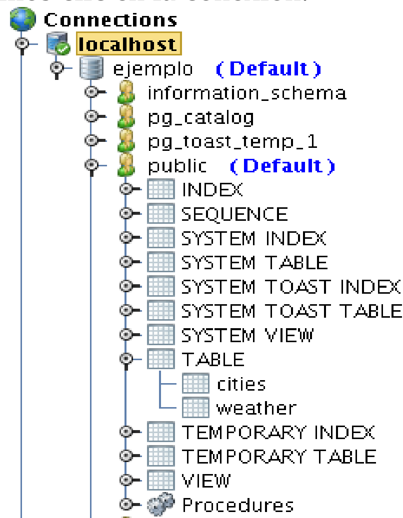


Hacer clic en “Finish”.

Hacer clic en “Close”.



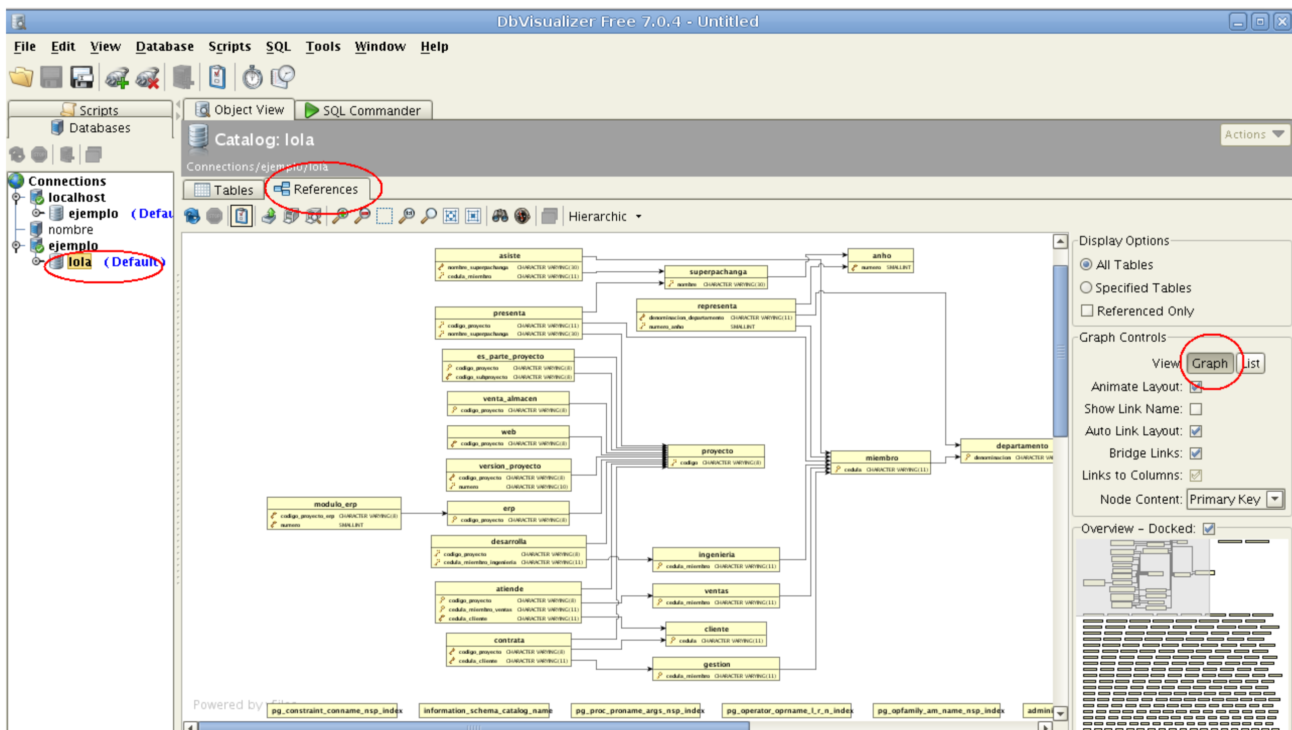
Al igual que en pgadmin3 hacemos clic en la conexión.



Podemos llevar a cabo solamente funciones de consulta, por ejemplo, no podemos insertar, modificar o borrar datos de las tablas.

Pero la funcionalidad que nos da DbVisualizer es que podemos ver el diagrama Entidad-Relación de la base de datos:

Hacemos clic sobre la base de datos.
Al lado derecho de la ventana hacemos clic en la pestaña references, verificar que esté marcada la opción “Graph”:



Rutinas de Mantenimiento:

Routine Vacuuming[6]

En términos de bases de datos Postgresql, se ocupa realizar un mantenimiento preventivo cada cierto tiempo, normalmente basta con ejecutar Autovacuum.

El mantenimiento se debe realizar por:

- Reutilización o recuperación de espacio en el disco duro, debido a las actualizaciones o eliminación de filas.
- Evitar pérdidas de datos.
- Actualizar estadísticas de datos.

El Vacuum se puede realizar como actividad paralela, en conjunto con las demás transacciones de la base de datos (selects, updates, inserts, deletes).

Hay dos formas de Vacuum, el Lazy y el total. El Lazy Vacuum necesita más espacio en disco, ya que se ejecuta más lento, así las transacciones en la base de datos continuarían funcionando normalmente, sólo no se modifica durante el proceso la transacción ALTER TABLE.

El Vacuum Total bloque la tabla con la que trabaja, no permite transacciones en paralelo, y aunque reduce el tamaño de la tabla, no hace lo mismo con el índice (puede ocurrir lo contrario, que se haga más grande) . Se aconseja utilizar el Vacuum estándar.

Actualización de Planificación de Estadística

La información estadística sirve para generar organizados planes para consultas, esto por medio del comando Analyze, que ejecuta Autovacuum cuando lo considere necesario. Aplica también para tablas y columnas, pero es más rápida si se ejecuta sobre la base de datos directamente.

La prevención de errores en las transacciones de ID Wraparound

En ocasiones puede resultar que algunos datos se pierdan por la transacción semántica de PostgreSQL, aunque estos existan aún. Por ellos, en promedio cada dos millones de transacciones se debe realizar un vacuum; se debe considerar los autovacuum_freeze_min_age y autovacuum_freeze_max_age, y sus diferencias de uso, como el aumento de tamaño del directorio pg_clog al utilizar autovacumm_freeze_max_age o lo irrelevante q puede llegar a ser un autovacuum_freeze_min_age en algún momento determinado.

Con las tablas pg_class y pg_databases del sistema es posible realizar seguimientos a los XIDs antiguos, gracias a la columna relfrozenxid pg_class.

Para realizar manualmente un vacuum completo de la base de datos:

```
$>vacuumdb -z -v -f dotlrn
```

Para ver las diferentes opciones:

```
$man vacuumdb.
```

6 Resumen de la información de www.postgresql.org realizado por Andrea Torres.

Autovacuum Daemon

Las funciones principales son:

- Autovacuum_naptime: consiste en distribuir los trabajos de cada base de datos a través del tiempo.
- Autovacuum_max_workers: lo que hace es indicar los procesos que se utilizaran en cada base de datos al mismo tiempo; si hay más de una base de datos en el sistema, se ejecutará una después de finalizar otra.

Rutina de Indexación

El comando REINDEX sirve para generar índice de forma periódica, prevenir el aumento del tamaño del índice (esto por problemas en la recuperación del espacio interno de índice de B-tree) y evitar que lo que se tenga en índices, sea mayor a la información contenida; al mismo tiempo, la indexación ayuda a mejorar la velocidad de acceso de los datos.

Periódicamente la base de datos debe ser reindexada para que no diminuya el rendimiento. Especialmente cuando se han realizado muchas inserciones.
Reindexar la Base de Datos:

Abrir una terminal

```
$>psql -U usuario template1
dotlrn=>reindex database basedatos;
dotlrn=>\q
```

RESPALDOS

Hay dos formas de hacer respaldos de las bases de datos, como una fotografía de la base de datos en un momento dado(pg_dump) o como un respaldo a un momento del tiempo dado(PITR). A continuación estudiaremos ambos métodos:

PG_DUMP

Es como tomar una fotografía de la base de datos en ese momento. No se puede recuperar la base de datos a un momento dado.

Para realizar el respaldo:

```
$pg_dump dbname > archivo.dmp
```

Para recuperar el respaldo:
Antes de recuperar el respaldo se debe crear la base de datos a partir de template0, para que la base de datos conserve las características del template1 en el que fue creado.

```
$createdb -T template0 dbname
$psql dbname < archivo.dmp
```

Para respaldar una base de datos de un servidor a otro

```
$pg_dump -h host1 dbname | psql -h host2 dbname
```

Para respaldar todas las bases de datos de un cluster:

```
$pg_dumpall > archivo.dmp
```

Para restaurar todas las bases de datos de un cluster:

```
$psql postgres < archivo.dmp
```

o lo que es loo mismo:

```
$psql  -f  archivo.dmp postgres
```

Para respaldar bases de datos grandes:

```
pg_dump dbname | gzip > archivo.gz
```

Restaurar con:

```
gunzip -c archivo.gz | psql dbname
```

Usando split:

```
pg_dump dbname | split -b 1m - archivo
```

Restaurar con:

```
cat archivo* | psql dbname
```

POINT IN TIME RECOVERY (PITR)[7]

Deben establecerse en el postgresql.conf (/etc/postgresql/8.3/main/postgresql.conf)
archive_mode = on
archive_command = 'test ! -f /opt/bitacoras/%f && cp %p /opt/bitacoras/%f'

Solamente si es necesario forzar la creación de un backup incremental cada XXX segundos
archive_timeout = 300

Lo importante es decidir adonde van a enviarse los archivos de respaldo (ojalá un servidor o dispositivo de almacenamiento remoto).

Si falla el almacenamiento, los archivos incrementales se quedan guardados en el pg_xlog hasta que tal circunstancia se resuelva.

El pg_xlog se encuentra en: **/var/lib/postgresql/8.3/main/pg_xlog.**

Obviamente la partición donde esté el pg_xlog/ podría llenarse. En caso de que eso suceda, PostgreSQL hará un PANIC shutdown, lo que implica que no se pierde ninguna transacción, pero hasta que no haya algo de espacio, la BD no podrá levantarse nuevamente. Si todo va bien, en el pg_xlog/ sólo se mantienen los 9 últimos archivos.

El directorio donde vayan a almacenarse (p.e. /opt/bitacoras) debe pertenecerle a postgres (o éste usuario tener permiso de escritura). Si no, no podrá trasladar los archivos y pasará lo mencionado anteriormente.

Cuando algo sucede con el comando de traslado, queda registrado en el log de postgresql, así que es recomendable consultar de vez en cuando el log.

Cuando todo está preparado, se puede hacer un restart de postgresql .

Hacer un backup:

Conectarse con el usuario postgres.

```
$psql -U postgres template1
template1#SELECT pg_start_backup('/opt');
```

El nombre que devuelve, es el id del segmento del primer archivo de respaldo incremental que debemos copiar, los anteriores pueden ignorarse. Para ver el nombre del archivo:

```
template1#select * from pg_xlogfile_name_offset('0/1E000020');
```

También es posible hacer las dos cosas directamente con:

```
template1#select * from pg_xlogfile_name_offset(pg_start_backup('/opt'));
```

Salir y hacer una copia completa del directorio donde está el cluster (/var/lib/postgres/8.3/main)

```
$cd /var/lib/postgres/8.3/
$tar czf /opt/data.tar.gz main
```

7 Guía elaborada por Carlos Juan Martín Pérez

Volver a entrar como postgres y ejecutar:
```
template1#SELECT pg_stop_backup();
```

La salida dice cual es el último segmento de transacciones durante la realización del backup. Para saber el archivo:
```
select * from pg_xlogfile_name_offset('0/1E01AC1C');
```
//Cambiar la cadena por la que corresponda que fue la salida de pg_stop_backup();

O bien haber ejecutado de una sola vez:
```
select * from pg_xlogfile_name_offset(pg_stop_backup());
```

Mover el backup completo a un lugar seguro.

Mover los archivos incrementales de /opt/bitacoras a partir del que dijo el start_backup hasta el que dijo el stop_backup al dispositivo donde se vaya a realizar el backup

Recuperar un backup :

Para poder restaurar necesitaremos el backup completo así como los archivos incrementales que se generaron durante el respaldo (entre el start y el stop) y los posteriores.

Se puede restaurar hasta un punto dado en el tiempo, por lo que no hay problema si lo que necesito es recuperar datos borrados antes de un momento en concreto y tengo archivos incrementales posteriores al desastre.

Detener el servicio de postgres.

Eliminar todo lo que haya dentro del directorio del cluster (p.e. **/var/lib/postgres/8.3/main**) .

Descomprimir el archivo de backup completo (.tar.gz).

Copiar los archivos incrementales respaldados(los que estan en **/opt/bitacoras**) en el pg_xlog (p.e. **/var/lib/postgres/8.3/main/pg_xlog**) .

Crear un archivo de instrucciones de restauración denominado "**recovery.conf**" en el directorio del cluster (p.e. vi /**var/lib/postgres/8.3/main/recovery.conf**).

Indicar qué se debe hacer para restaurar. Lo más básico es indicar donde están los archivos de restauración y hasta qué momento queremos restaurar:
restore_command = 'cp /var/lib/postgres/8.3/main/pg_xlog/%f "%p"'
recovery_target_time='13/05/2009 12:55:47.548867 CST'

Como usuario root :
Antes de arrancar la BD, puede ser importante impedir que se conecte nadie al SGBD para asegurar previamente que todo está en orden: para ello únicamente es necesario modificar el **pg_hba.conf**.

Para que no haya problemas, antes de reiniciar es conveniente limpiar el directorio donde se estaban haciendo la transferencia de archivos incrementales para que no se encuentre con uno que ya exista y quede detenido el traslado.

Entonces podemos ya arrancar el servidor.

Pueden verse en la bitácora **/var/log/postgresql/postgresql-8.3-main.log** los efectos de la restauración: el servidor .

PostgreSQL procede a leer los archivos incrementales hasta el punto en tiempo dado.

Una vez concluída la restauración, el archivo **recovery.conf** es renombrado a **recovery.done**

Comprobar que todo se restauró como debía .

Rehabilitar el acceso de los usuarios en el **pg_hba.conf** y recargar la configuración.

CAPÍTULO 4

REPLICACIÓN

Replicación

Para el estudio de éste tema utilizaremos el artículo: Replicación y alta disponibilidad de PostgreSQL con pgpool-II[8] de Jaume Sabater, el cual se encuentra en los materiales adicionales del curso con el nombre postgres-pgpool.odt. Este artículo ha sido modificado y corregido. Asimismo en los materiales adicionales se pueden encontrar varios archivos con los scripts sugeridos en el artículo.

8 http://linuxsilo.net/articles/postgresql-pgpool.html

FUENTES CONSULTADAS:

- www.postgresql.org
- http://www.linux-es.org/node/660
- http://www.ibiblio.org/pub/linux/docs/LuCaS/Tutoriales/NOTAS-CURSO-BBDD/notas-curso-BD/node134.html
- http://linuxsilo.net/articles/postgresql-pgpool.html#sobre-pgpool-ii
- http://lca2007.linux.org.au/att_data/Miniconfs(2f)PostgreSQL/attachments/getting_started.pdf
- http://wiki.postgresql.org/wiki/
- http://www.postgresql-es.org/node/301,
- http://linuxsilo.net/articles/postgresql-pgpool.html