

# 1 - Instalación

## 1 - Instalación

Vamos a ver más en detalle el proceso de instalación, para ser más conscientes de lo que realmente está haciendo.

### 1.1- Instalación del servidor.

Consistirá en descargar y descomprimir el fichero correspondiente a nuestro sistema operativo desde la página de PostgreSQL.

ATENCIÓN: Puedes descargar la última versión disponible, en estos momentos la versión 9.4. El tema está desarrollado con una versión anterior.

Para el entorno Windows :

<http://www.enterprisedb.com/products-services-training/pgdownload#windows>

### 1.2 Actualizaciones

Si no estamos instalando de nuevo, sino que estamos actualizando a una versión posterior de PostgreSQL, seguramente nos convendrá conservar las Bases de Datos. El proceso es muy sencillo: antes de empezar la instalación hacer una copia de seguridad de todas las Bases de Datos con `pg_dumpall` desde una terminal como `postgres`, y después de todo instalado y funcionando restaurar con `psql`, o con `pg_restore`, dependiendo del tipo de copia realizada. En el apartado 6 de este tema se ve este proceso de copia de seguridad y restauración.

### 1.3 - Instalación del cliente

El cliente incluye todo lo necesario para la conexión a un servidor remoto, a parte del `psql` y algunas herramientas administrativas (`createdb`, `pg_dump`, ...)

En Windows, en principio, sería suficiente con el programa `psql.exe`.

En cualquier caso tenemos la posibilidad de instalar PgAdmin para acceder y administrar una Base de Datos remota.

Una vez instalado el cliente debemos recordar que la conexión debe ser a un servidor remoto. Así con `psql` debemos utilizar la opción `-h`

`psql -h servidor`

Un ejemplo de conexión remota: `psql -h 84.124.27.36 -U miguel`

Una vez instalado podremos entrar desde la consola de administración de Postgres que se llama 'PgAdminIII'

Nos aparecerá un servidor de bases de datos en la máquina local (localhost:5432)

Al conectar con el servidor (contraseña la puesta en la instalación –qwerty- en mi caso) nos despliega las distintas bases de datos que tiene el servidor, en nuestro caso una por defecto llamada ‘postgres’.

Si nos situamos sobre el icono de la base de datos y pulsamos el botón secundario nos muestra el menú contextual en el cual podemos seleccionar la opción ‘RESTORE’.

Elegiremos el fichero GEO.backup, que puedes descargar desde la página del curso, y seleccionamos ‘Restore’.

Si todo ha ido bien tendrás dentro del esquema ‘public’ tres tablas.

Ya tienes lo necesario para comenzar a escribir comandos SQL sobre este esquema de base de datos:



Este proceso se puede seguir en el tutorial a continuación:

### Nota Previa

En este tema, probaremos hacer tareas de administración de Postgres tanto desde **PgAdmin** (entorno gráfico) como desde una consola (entrando en psql o no). En este segundo caso, muchas veces tendremos que hacer las cosas como un *usuario postgres del S.O.*

Para tener un terminal como usuari postgres de S.O. en Linux, podemos desde una consola normal cambiar al usuario root, y desde ahí cambiar a postgres. Concretamente en Edubuntu podríamos hacerlo así:

```
$ sudo su - postgres
Password:
$
```

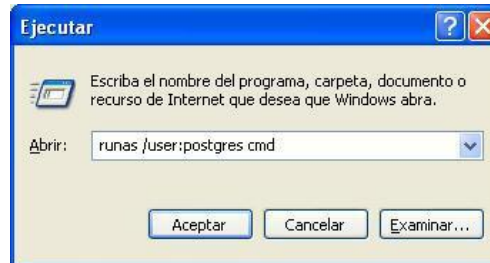
donde la contraseña es la del usuario, para poder hacer el sudo. En otros Linux seguramente tendríamos que cambiar a root, i después a postgres.

Desde Windows lo podemos hacer abriendo una consola (Símbolo de sistema), y desde ahí abrir una nueva consola, esta vez del usuario postgres, con el comando runas:

```
> runas /user:postgres cmd
```

recuerda que la contraseña que habíamos puesto (si no se te ha ocurrido otra cosa) era qwerty.

Nos podríamos ahorrar abrir la primera consola si ponemos el comando anterior en Inicio -> Ejecutar



Así se nos abrirá la consola como un usuario postgres, como podemos comprobar en la barra de título de la siguiente imagen:

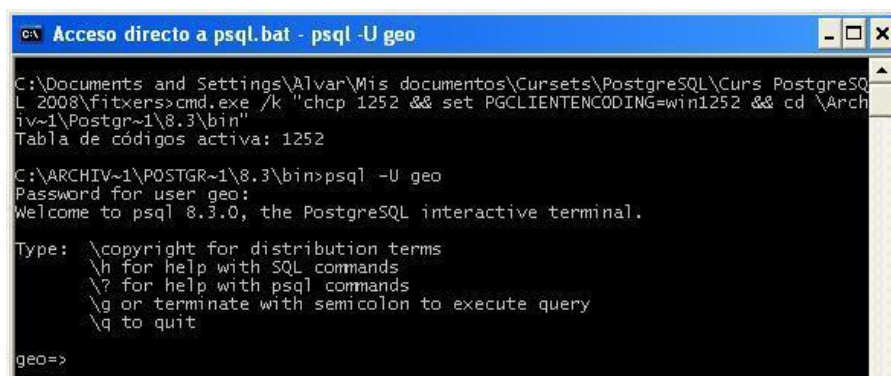


Haremos muchas pruebas de conexión de distintos usuarios sobre distintas Bases de Datos.

Para probar todas estas conexiones, seguramente lo más útil será utilizar psql, ya que en pgAdmin tendríamos que hacer una conexión para cada una. En Windows lo más cómodo será abrir la consola que tenemos en Inicio -> Todos los programas -> PostgreSQL 9.2 (o bien 9.3) -> Command Prompt, ya que el directorio por defecto será donde están los programas de PostgreSQL

O directamente ejecutar el fichero psql.bat que tenéis en este tema.

Recuerda que si ponemos psql.exe sin nada más intentará conectar como un usuario igual que el de S.O. y una B.D. llamada igual. Por tanto, lo más habitual será poner las opciones -U (para indicar el usuario) y -d (para indicar la Base de Datos; si no la ponemos intentará entrar en una BD con el mismo nombre que el usuario). La siguiente imagen nos muestra un ejemplo:



## 2. Arranque y parada del sistema

### 2 - Arranque y parada del sistema

Haremos la parada y arranque del sistema en modo línea desde una consola, y también lo intentaremos desde PgAdmin.

- En Windows:

Automático.

Por haber pedido en la instalación como un servicio, manualmente podemos elegir la opción siempre que tengamos una versión anterior a la 8.4

*Inicio->TodoslosProgramas->PostgreSQL8.3->StartService.*

Para pararlo

*Inicio->TodoslosProgramas->PostgreSQL8.3->StopService.*

Utilizando **pg\_ctl.exe**.

Es el programa equivalente al script de Linux. Se encuentra con los ficheros ejecutables

(*C:\ArchivosdePrograma\PostgreSQL\9.2\bin*).

Da problemas en las opciones de arranque si se ejecuta como administrador del sistema, por tanto lo deberíamos hacer desde una consola como el usuario de S.O. postgres (miráis la nota previa al inicio del tema por ver como abrir una consola de postgres en Windows).

Tendremos las mismas opciones que hemos visto en el caso de Linux.

En la siguiente imagen tenemos la orden que hace posible el re arranque por medio de **pg\_ctl.exe**, donde la Base de Datos está en *C:\ArchivosdePrograma\PostgreSQL\9.2\data* (observa que se trata de una consola ejecutada como *postgres*):

Ejecutando directamente el programa postgres.exe situado en el mismo lugar. Tendremos las opciones :

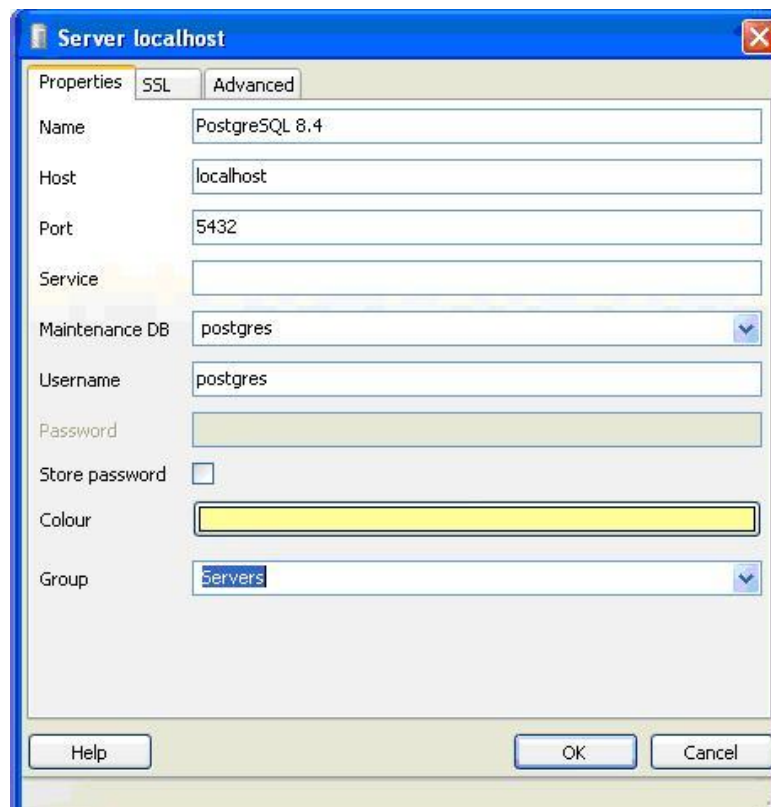
### postgresql-8.3 opción

donde la opción puede ser:

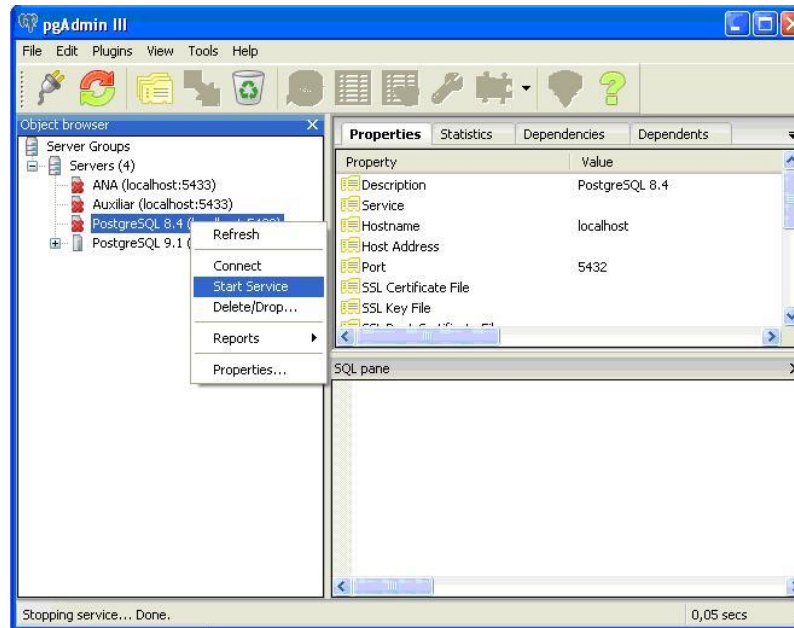
- **start** arranca el servidor
- **stop** lo para
- **restart** rearranca (lo para, y lo vuelve a poner en marcha)
- **reload** vuelve a leer los ficheros de configuración (sin rearmar)
- **status** ver el estado en que se encuentra actualmente (en marcha, parado, ...), y otros informaciones de interés

### Con PgAdmin

Si tenemos bien configurada la conexión al servidor como posgres podremos parar y poner en marcha el servidor. Para eso debemos tener donde dice Servicio (Service) bien el servicio de Windows (como en la figura) o bien la ruta de pg\_ctl con los parámetros necesarios para poder arrancar



Esta entrada era la única que había por defecto, y es para el usuario postgres. Si es así podremos parar o poner en marcha, bien en el menú de herramientas, bien en el menú emergente sobre el servidor



### Nota

Es muy posible que si hemos ido jugando con pg\_ctl.exe o postgres.exe para reiniciar el servidor PostgreSQL, que en el pgAdmin no se pueda iniciar o parar el servidor. De hecho solo nos propondrá Iniciar el servicio, cuando a lo mejor lo tenemos ya en marcha.

Seguramente todo volverá a funcionar correctamente la siguiente vez que arranquemos Windows.

En Linux, aunque parezca mentira, es más complicado parar y poner en marcha desde pgAdmin. Como mínimo desde la versión que se está documentando en estos apuntes, que para el pgAdmin instalado por paquetes en Ubuntu es la 1.4.3. Tal y como dice la documentación deberemos colocar en el servicio la orden que pondría en marcha o pararía el servidor utilizando **pg\_ctl**, que sería esta:

```
sudo-uposgres/usr/lib/postgresql/9.2/bin/pg_ctl-D
/var/lib/postgresql/9.2/main-o'-c
config_file=/etc/postgresql/9.2/main/postgresql.conf'
```

Pero además se debería ejecutar el pgAdmin como root, ya que si no, no tendría efecto el sudo.

Por tanto, como veis, un poco "rollo"...

### 3 - Ficheros de configuración

Básicamente son 3 los ficheros de configuración:

- **postgresql.conf** donde están la mayor parte de parámetros
- **pg\_hba.conf** para controlar las conexiones
- **pg\_ident.conf** se combina con el anterior para las conexiones

En Windows se encuentran normalmente en el directorio de datos, que es *C:\Archivos de programa\PostgreSQL\9.2\data*

En Lliurex, y en general en Linux, si hemos hecho la instalación por paquetes los ficheros de configuración estarán en */etc/postgresql/9.2/main*. Si la instalación la hemos hecho a mano, seguramente estarán en el directorio de datos que es normalmente

*/var/lib/postgresql/9.2/main*.

Vamos a comentar algunos de los parámetros de **postgresql.conf**. Los otros dos ficheros de configuración los veremos en la pregunta 4.4, cuando veamos la autenticación de los usuarios. Si llevan **# delante**, están comentados y entonces se coge el valor por defecto. Cuando cambiamos algún parámetro, deberemos hacer como mínimo un *reload* (*/etc/init.d/postgresql-9.2 reload*), aunque con algunos parámetros eso no es suficiente y se deberá hacer un *restart*.

- **hba\_file** lugar donde está el fichero **pg\_hba.conf**
- **ident\_file** lugar donde está el fichero **pg\_ident.conf**
- **listen-addresses** lista de IP donde escuchar; **localhost** solo local (por defecto)\* todas
- **port** puerto donde escuchar las conexiones TCP; por defecto 5432
- **max\_connections** número máximo de conexiones (excepto DBA que puede hacer las que quiera)
- **superuser\_reserved\_connections** número de conexiones reservadas para los DBA;

así el máximo de conexiones para el resto de usuarios son **max\_connections** –

- **superuser\_reserved\_connections authentication\_timeout** tiempo máximo (en segundos) para autenticarse un usuario.
- **ssl** habilita las conexiones ssl (por defecto off)
- **password\_encryption** determina si por defecto la contraseña será encriptada (evidentemente el valor por defecto es donde)



## Nota

Si después de modificar algún fichero de configuración, este tiene algo mal (es sintácticamente incorrecto), PostgreSQL no podrá arrancar de nuevo. Por eso es altamente recomendable hacer una copia de los ficheros de configuración antes de hacer alguna modificación.

## 4 - Gestión de usuarios y privilegios

Para poder proteger las datos entre las múltiples personas que se pueden conectar a una Base de Datos, que los datos no puedan ser vistos por otros (a no ser que lo queramos expresamente) y mucho menos que puedan ser manipuladas consciente o inconscientemente, los SGBD utilizan la autenticación por **usuarios**. En principio cada usuario nada más puede acceder a sus tablas, las que crea él (es el propietario). Pero también podremos dar permisos. Así un usuario podría permitir la utilización de una tabla a otro, dando diferentes grados de acceso: nada más consultar, o también añadir, o borrar, o actualizar, ..., o todas juntas, e incluso darle permiso para que a su vez diera permisos a otros.

Si muchos usuarios deben tener permisos similares, sería muy conveniente que el SGBD pudiera gestionar grupos de **usuarios**. Los usuarios podrían así pertenecer a determinados grupos. Si después damos permiso de acceso a una determinada tabla a este grupo de usuarios, es como si hubiéramos dado el permiso uno a uno a todos los usuarios que pertenecen al grupo. Por tanto facilitarán mucho el trabajo del administrador.

Hasta la versión 8.0, *PostgreSQL* gestionaba los usuarios y grupos como tales (*user* y *group*), pero a partir de la versión 8.1 generaliza estos dos conceptos en un *rol* (*role*).

Antes eran dos entidades diferentes. Ahora es una sola, y un rol puede actuar como un usuario, un grupo o ambos.

La única diferenciación será que algunos roles se pueden conectar (login) y se llaman **roles de entrada**, que sería el equivalente de usuario; y otros no (que sería el equivalente de grupo) y se llaman justamente roles **de grupo**.

Por otra parte, PostgreSQL es muy potente en cuanto a la limitación de la conexión de los usuarios, pudiendo dar más de un sistema de autenticación, y limitar mucho el acceso desde máquinas remotas, controlando tanto el usuario como la IP de la máquina desde donde se quiere conectar.

Por último, la seguridad de los ficheros queda garantizada por propietario, *postgres*.

### 4.1 - Gestión de roles: utilización como usuarios

Un **rol** es una entidad capaz de recoger permisos y privilegios. Uno de estos permisos es el de conexión (login). En este sentido sería como un usuario, eso sí, completamente independiente de los usuarios de S.O.



Este **rol** o usuario (PostgreSQL mantiene en cierta medida la nomenclatura de usuarios) será un nombre con una posible contraseña (que dependiendo del método de autenticación, servirá para controlar el acceso) que tendrá distintos permisos para crear tablas y otros objetos en una determinada Base de Datos, utilizarlos para hacer consultas o actualizaciones,...

Aparte de esto, los permisos de un rol pueden asignarse a otro. Entonces es como si el segundo rol pertenecerá al primero, y el primero funciona como un grupo.

Los roles no están incluidos en ningún base de datos particular. Por tanto son globales a toda la instalación de PostgreSQL (la gran Base de Datos, el cluster).

Los roles, tanto si pueden hacer login como si no, se guardarán en la tabla *pg\_authid* (en caso de usuarios con la contraseña encriptada o no). Por comodidad (y compatibilidad con versiones anteriores) hay unas vistas a que pueden hacer más cómoda la consulta.

- **pg\_roles** contiene todos los roles.
- **pg\_user** contiene los usuarios, es decir, los roles a que pueden hacer un login.
- **pg\_shadow** contiene también las contraseñas.
- **pg\_group** contiene los grupos, es decir, los roles que no pueden hacer login.

Por tanto la primera manera de gestionar los roles sería manipular directamente las tablas o vistas, aunque parece demasiado laborioso.

Vamos a ver las maneras normales de crear roles.

## CREATE ROL

Deberemos ejecutar esta sentencia SQL desde un usuario con permiso para crear roles, y conectado a cualquiera B.D.

La sintaxis es la siguiente:

```
CREATE ROLE nombre [ [ WITH ] opción [ ... ] ]
```

donde la opción puede ser (las subrayadas son las opciones por defecto):

**SUPERUSER** | **NOSUPERUSER** permiso de superusuario (por defecto no)

**CREATEDB** | **NOCREATEDB** permiso para crear B.D (por defecto no)

**CREATEROLE** | **NOCREATEROLE** permiso para crear usuarios (por defecto no)

**CREATEUSER** | **NOCREATEUSER** similar a anterior (obsoleta)

**INHERIT** | **NOINHERIT** determina si el rol hereda las propiedades de los grupos (roles) a los que pertenece

**LOGIN** | **NOLOGIN** permiso para conectarse (será un usuario)

**CONNECTION LIMIT** **connlimit** si un usuario puede conectarse esto especifica cuantas conexiones

concurrentes puede haber (por defecto -1, que quiere decir ilimitadas)

**[ENCRYPTED | UNENCRYPTED] PASSWORD 'password'** contraseña del usuario que puede ir encriptada o no

**VÁLIDO UNTIL 'fecha'** fecha de caducidad del usuario

**IN ROL rolname** [, ...] roles (grupos) a los que pertenece

**IN GROUP rolname** [, ...] similar al anterior (obsoleta)

**ROLE rolname** [, ...] roles que pertenecerán a este rol (grupo)

**USER rolname** [, ...] similar al anterior (obsoleta)

**ADMIN rolname** [, ...] similar al anterior pero además tendrá permiso para administrarlo (**WITH ADMIN OPTION**)

Para mantener la compatibilidad con versiones anteriores tenemos la sentencia equivalente:

**CREATE USER nombre [ [ WITH ] opción [ ... ] ]**

donde por defecto sí que tendrá el privilegio LOGIN.

Así, por ejemplo, desde una conexión por *PSQL como* posgres a xarxa *podemos* hacer:

```
CREATE ROLE xar1 LOGIN;
--puede conectarse, no tiene password y no puede crear ni usuarios
ni B.D.
CREATE USER xar2;
--puede conectarse, no tiene password y no puede crear ni usuarios
ni B.D.
CREATE ROLE xar3 LOGIN PASSWORD 'xar3';
--puede conectarse, tiene password, no puede crear ni usuarios
ni B.D.
CREATE ROLE xar4 LOGIN PASSWORD 'xar4' CREATEDB;
--puede conectarse, tiene password, no puede crear usuarios,
pero sí B.D.
CREATE ROLE xar5 LOGIN PASSWORD 'xar5' CREATEDB CREATEROLE;
--puede conectarse, tiene password, y puede crear usuarios y
```

B.D.

```
CREATE ROLE xar6 LOGIN PASSWORD 'xar6' VALID UNTIL '31-05-2010';  
--puede conectarse hasta el 31 de mayo a las 0:00, tiene password y  
no puede crear ni usuarios ni B.D.
```

Si queremos modificar algún aspecto del usuario, lo haremos con la sentencia

```
ALTER ROLE nombre [ [ WITH ] opción [ ... ] ]
```

donde las opciones son las mismas que en CREATE ROLE, por ejemplo:

```
ALTER ROLE xar2 PASSWORD 'xar2';
```

aunque esta sentencia tiene más utilidades:

```
ALTER ROLE nombre RENAME TO nombre_nuevo
```

O

```
ALTER ROLE nombre SET parámetro TO valor
```

```
ALTER ROLE nombre RESET parámetro
```

que servirán para inicializar determinados parámetros a un determinado valor, para el usuario.

Si queremos eliminar un rol, sencillamente

```
DROP ROLE nombre
```

Evidentemente existen las sentencias sinónimas **ALTER USER** y **DROP USER**

**\*create user**

Otra manera de crear roles, esta vez desde el sistema (sin entrar en PostgreSQL). Es un fichero ejecutable que proporciona PostgreSQL, y que nada más podrán ejecutar los superusuarios PostgreSQL. Si no ponemos la opción **-U** se intentará ejecutar como un

usuario PostgreSQL con el mismo nombre que el usuario de S.O. que está ejecutando este ejecutable. En principio, como de momento nada más tenemos un superusuario (más concretamente un usuario con permiso para crear usuarios) si queremos tener éxito deberemos ejecutarlo como usuario de S.O. *postgres*, pero si estamos en una consola como un usuario distinto podemos esquivarlo con la opción **-U**. Más adelante pondremos un ejemplo.

La sintaxis es:

```
create user [opciones] nombre
```

Y en las opciones, entre otros, podremos poner:

- s el nuevo usuario será superusuario (si no lo ponemos nos lo pedirá)
- S el nuevo usuario no será superusuario
- d el nuevo usuario podrá crear bases de datos (si no lo ponemos nos lo pedirá)
- D el nuevo usuario no podrá crear bases de datos
- r el nuevo usuario podrá crear roles (si no lo ponemos nos lo pedirá)
- R el nuevo usuario no podrá crear roles
- l el nuevo usuario podrá conectarse (por defecto)
- L el nuevo usuario no podrá conectarse
- P **pedirá** la contraseña para el nuevo usuario

Aparte de estas opciones podemos poner otras, similares a las opciones de *psql*

- h nombre nombre del servidor a que nos conectamos (por defecto local)
- p port puerto del servidor a través del que nos conectamos (por defecto 5432)
- U nombre nombre del usuario que hace la operación (no el que se crea)
- W para que nos pida obligatoriamente la contraseña del usuario que hace la operación (no el que se crea).

Así, si en la consola no estamos como usuario posgres, podemos cambiar a este usuario, o sencillamente hacer:

```
create user -U posgres
```

Aquí tenemos algunos ejemplos. Los dos primeros hacen exactamente el mismo. Hemos puesto tanto las sentencias de creación como los comentarios que salen, como las contestaciones. Observa que la contraseña que pide es la de quien ejecuta el orden. Por tanto deberemos ejecutarlo como **posgres** y poner su contraseña. En el último también pide la contraseña que se quiere poner al usuario que se va a crear.

```
create user -h 127.0.0.1 xar7
```

```
¿Será el nuevo rol un superusuario? (s/n) n
```

```
¿Debe permitírsele al rol la creación de bases de datos? (s/n) n
```

```
¿Debe permitírsele al rol la creación de otros rolas? (s/n) n
```

```
Contraseña:
```

```
CREATE ROLE
```

```
create user -h 127.0.0.1 -S -D -R xar7
```

Contraseña:

```
CREATE ROLE
```

```
create user -h 127.0.0.1 -S -P xar8
```

Ingrese la contraseña para el nuevo rol:

Ingrésela nuevamente:

¿Debe permitírsele al rol la creación de bases de datos? (s/n) n

¿Debe permitírsele al rol la creación de otros roles? (s/n) n

Contraseña:

```
CREATE ROLE
```

**Nota:**

Los ejercicios anteriores están para que funcionen igual tanto en Windows como en Linux. Si en Windows no ponemos el host (**-h 127.0.0.1**) funcionaría exactamente igual.

En cambio, en Linux si no ponemos el host no nos pedirá la contraseña, pero funcionará bien (siempre que lo ejecutamos como **postgres**). La explicación de todo esto está en el punto **4.5 Autenticación de usuarios**.

No existe el programa **alteruser**, para modificar un rol, pero sí **dropuser** por eliminarlo

```
dropuser [opciones] nombre
```

donde las opciones son algunas del createuser, con el mismo significado: **-h -P -U -W**

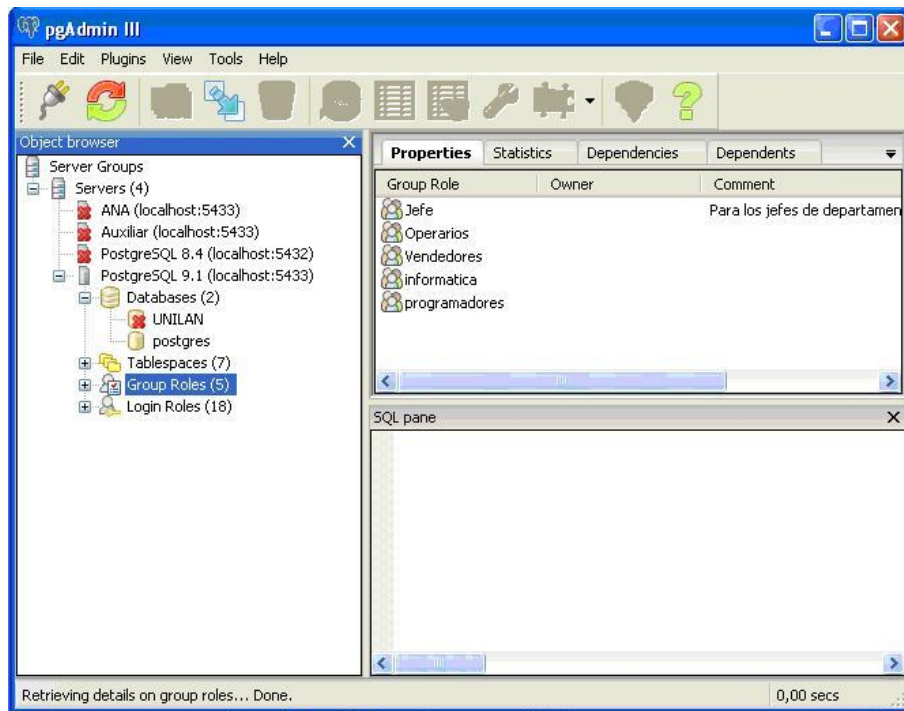
```
dropuser xar7
```

Contraseña:

```
DROP ROLE
```

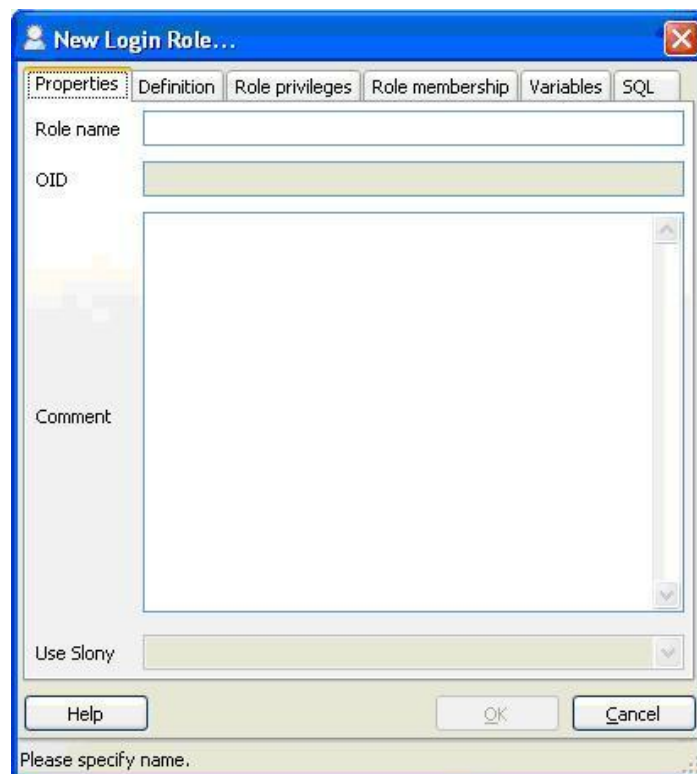
## Herramientas gráficas

Para dar de alta un rol utilizado como un usuario con PgAdmin deberemos ir a la opción **Roles de entrada (Login Roles)** habiéndonos autenticado como un usuario (un rol) que puede crear usuarios (roles), que en principio puede ser **postgres**.



Podemos ver que ya tenemos 5 roles de entrada creados (si ya has ejecutado las sentencias del apartado anterior, también te aparecerán **xar1** ... **xar8**), y que son independientes de las Bases de Datos, mejor dicho, no están dentro de ellas.

Si intentamos crear un usuario nuevo, veremos que tenemos todas las posibilidades que teníamos en la creación por SQL. Y además no nos deja tocar la opción **Puede entrar (Login)**, ya que es un rol utilizado como un usuario.



De la misma manera podremos modificar un usuario (yendo a sus propiedades) o eliminarlo (a la papelera!).

Estos roles de entrada o usuarios son absolutamente independientes de los usuarios del S.O. (que ya se encarga él de que se autentiquen). Puede haber usuarios de ambos tipos con el mismo nombre, cosa que puede facilitar la autenticación, pero no tiene porque ser así. Perfectamente puede haber usuarios de la B.D. que no existen en el S.O. (y al revés, evidentemente). Pero si no hacemos nada más, estos usuarios no podrán conectarse. En el punto 4.5 trataremos el problema de la autenticación.

## 4.2 - Gestión de roles: utilización como grupos

Como veremos en el siguiente punto, a los usuarios les podremos dar diferentes permisos de acceso a las tablas. Pero si son muchos los usuarios que deben tener los mismos permisos, se hace incómodo tener que darlos uno a uno.

Aquí entra en juego el concepto de Roles *de grupo*, que son roles pero que se utilizan no para autenticarse, sino para que otros roles (en principio de usuario) puedan heredar sus permisos. Como ya hemos comentado anteriormente, antes de la versión 8.1 de PostgreSQL existían los usuarios y los grupos como entidades diferentes. A partir de esta versión se generaliza el concepto, englobando las características de las dos entidades en una, el *rol*.

La tabla donde se guardarán los roles de grupos es, evidentemente, la misma *pg\_authid*, pero existirá una vista para ver únicamente los roles de grupo, es decir, aquellos que no pueden entrar, que será *pg\_group*.

La manera de crear un rol de grupo es la misma, pero ahora sin dar la posibilidad de que se conecte. Tenemos también la sentencia similar (y obsoleta) **CREATE GROUP**. Este sería un ejemplo utilizando la sentencia **CREATE ROLE**

```
CREATE ROLE grup0 NOLOGIN;
```

Y ahora creamos un grupo e incorporamos al usuario a este grupo

```
CREATE ROLE grup1 NOLOGIN ROLE xar1;
```

### **CREATE GROUP**

La sintaxis es la misma. No vamos a repetirla entera, tan solos señalar el aspecto de *grupo* que tiene:

```
CREATE GROUP nombre [ [ WITH ] opción [ ... ] ]
```

donde daremos nombre al grupo y opcionalmente pondremos el nombres de los usuarios que pertenecen.



Por ejemplo:

```
CREATE GROUP grup2;
```

```
CREATE GROUP g_xar WITH USER xarxa, xar1, xar4;
```

Podremos modificar un grupo para añadir o quitar usuarios, o para cambiar el nombre del grupo.

```
ALTER GROUP nombre_grupo ADD USER nombre_usuario [, ...] ;
```

```
ALTER GROUP nombre_grupo DROP USER nombre_usuario [, ...] ;
```

```
ALTER GROUP nombre_grupo RENAME nombre_nuevo
```

Y para borrar el grupo:

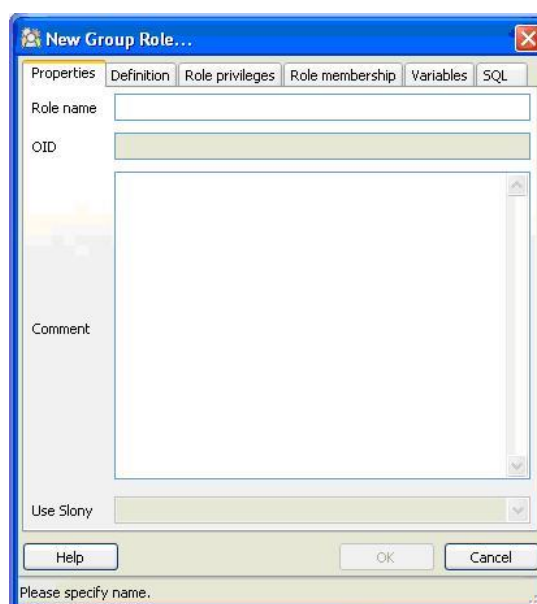
```
DROP GROUP nombre_grupo;
```

Ahora no tendremos el programa *creategroup*, pero tampoco es necesario, ya que para hacer un rol considerado como grupo, nada más debemos procurar que no se pueda conectar. Nos preguntará el de siempre (si es superusuario, si puede crear BD, si puede crear roles; podemos contestar a todo que no). Recordáis que se debe ejecutar como **postgres**, o como cualquier otro usuario que tenga privilegio de crear roles (en los ejemplos de la pregunta anterior el único que lo podría hacer sería **xar5**)

```
createuser -h 127.0.0.1 -L g_geo
```

### Herramientas gráficas.

EEEn PgAdmin tenemos el apartado de Roles *de Grupo*, que no da la posibilidad de conectarse salvo que introduzcamos una password en la pestaña 'Definición':



## 4.3 - Permisos

En principio solo quien crea una tabla (o cualquier objeto) puede acceder a ella (salvo el superusuario). Pero este propietario puede dar permisos a los otros roles (individuales o grupos) para leerla, insertar, borrar, ... También lo podrá hacer el superusuario. Estos permisos no incluirán el de borrar la tabla o modificarla, que solo lo podrá hacer el suyo propietario.

### **GRANT (y REVOKE)**

Para dar permisos utilizamos **GRANT**

Su sintaxis es

```
GRANT { {SELECT|INSERT|UPDATE|DELETE|REFERENCES|TRIGGER}
[,...] | ALL [ PRIVILEGES ] }
ON [ TABLE ] nombre_tabla [, ...]
TO { nombre_usuario | GROUP nombre_grupo | PUBLIC } [, ...]
WITH GRANT OPTION ]
```

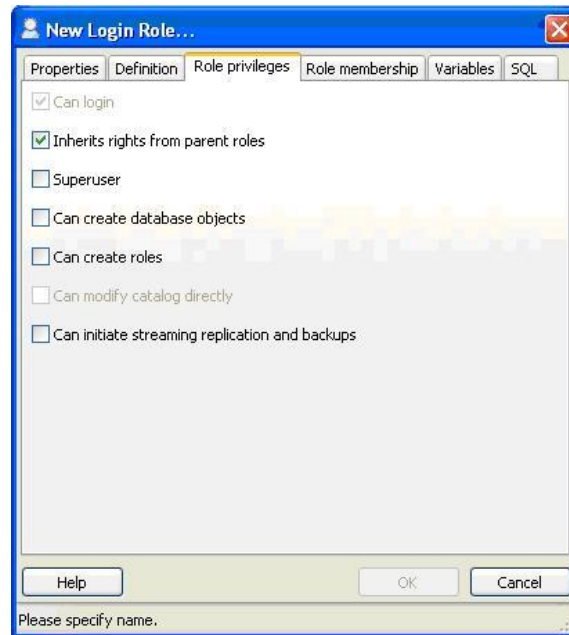
Es decir, que sobre una tabla se puede dar permiso solo para seleccionar, o insertar, o modificar, o borrar, o crear una clave externa sobre esta tabla, o crear un trigger sobre esta tabla. O muchos de ellos. O todos (*ALL*).

Se puede dar permiso a un usuario (o más) o a un grupo, o a *PUBLIC*, es decir a todo el mundo. En realidad la distinción entre usuario y grupo es para mantener compatibilidades, ya que ya sabemos que ahora todo son roles. De hecho no habrá problema si ponemos el nombre de uno rol de grupo sin la cláusula **GROUP**, o incluso si ponemos el nombre de un rol de usuario en la cláusula **GROUP**.

Si además ponemos la opción **WITH GRANT OPTION**, los usuarios a los que hemos dado permiso pueden dar estos permisos a otros.

Cuando se da un permiso a un rol al que pertenecen una serie de miembros, estos heredarán los permisos solo si tienen el privilegio **INHERIT**.

Este privilegio **INHERIT** se ve muy gráficamente en PgAdmin



Por ejemplo, en el usuario y Base de datos **geo**:

```
GRANT SELECT ON provincias TO PUBLIC; -- provincias es una tabla
creada en el Tema 3, pregunta 5.6
```

```
GRANT SELECT,UPDATE ON comarcas,poblaciones TO GROUP g_xar;
```

Ahora podríamos conectarnos como usuario **XARXA** (que pertenece al grupo **g\_xar**) a la BD **geo** y podríamos consultar la tabla **provincias** (igual que todo el mundo), y también podríamos consultar y actualizar las tablas **comarcas** y **poblaciones** (como todos los miembros de **g\_xar**).

Para quitar los permisos utilizaremos **REVOKE**, que tiene una sintaxis paralela al **GRANT**

```
REVOKE [ GRANT OPTION FOR ]
{ SELECT | INSERT | UPDATE | DELETE | REFERENCES | TRIGGER }
[, ...] | ALL [ PRIVILEGES ] }
ON [ TABLE ] nombre_tabla [, ...]
FROM { nombre_usu | GROUP nombre_grupo | PUBLIC } [, ...]
CASCADE | RESTRICT ]
```

Donde deberemos tener en cuenta que solo el usuario que ha dado un permiso (o el superusuario) puede quitarlo, excepto en el caso de la opción **CASCADE**, que reemplaza también el permiso a todos los que se lo han pasado por tener el permiso **WITH GRANT OPTION**, en ese caso.

**GRANT** y **REVOKE** también sirven para dar permisos o quitarlos sobre la Base de Datos (para poder crear nuevos esquemas), o sobre esquemas (para poder crear nuevos objetos en ellos), o sobre funciones (para poder ejecutarlas), o sobre lenguajes (para poder utilizarlos). Se puede consultar la documentación de PostgreSQL para ver la sintaxis. Nada más vamos a poner la sintaxis de funciones:

```
GRANT { EXECUTE | ALL [ PRIVILEGES ] }  
  
ON FUNCTION nombre_func ([type, ...]) [, ...]  
  
TO { nombre_usu | GROUP nombre_grupo | PUBLIC } [, ...]  
  
WITH GRANT OPTION ]
```

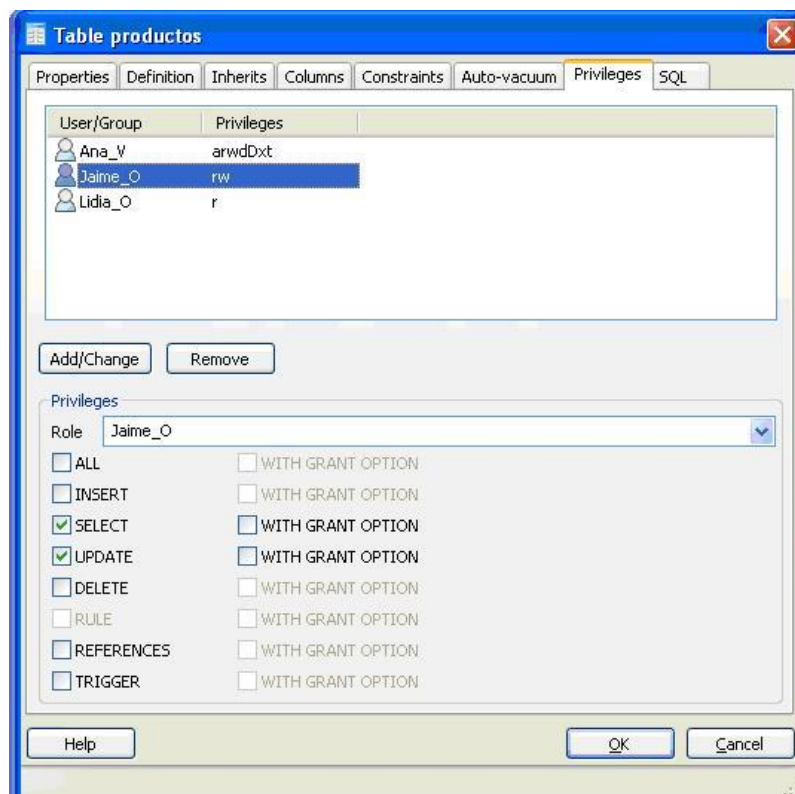
Y el **REVOKE** es similar

**GRANT** y **REVOKE** también sirven para hacer miembro de un rol de grupo a otro rol.

Lo veremos mejor en el siguiente apartado.

### \* Herramientas gráficas

Todo lo anterior, en las herramientas gráficas, se hace sobre el objeto del que se quieren dar permisos, en la pestaña *Privilegios*, de una manera muy clara y sencilla:



donde las letras que representan los permisos en la parte de arriba son:

- **a** (append): INSERT
- **r** (read): SELECT
- **w** (write): UPDATE
- **d** (delete): DELETE
- **x** (execute): REFERENCES
- **t** (trigger): TRIGGER

## 4.4 - Roles y miembros

Vamos a ver con más detalle qué supone el pertenecer a un grupo (en realidad a un rol).

Pondremos algunos ejemplos y sacaremos conclusiones prácticas.

Vamos a crear un rol de grupo (en principio sin poder conectarse, aunque podría hacerlo).

```
CREATE ROLE g_xarxa;
```

Creamos unos roles de usuario, y les haremos miembros de este rol de grupo.

```
CREATE ROLE xarxa1 LOGIN INHERIT PASSWORD 'xarxa1';
```

```
GRANT g_xarxa TO xarxa1;
```

```
CREATE ROLE xarxa2 LOGIN INHERIT IN ROLE g_xarxa
```

```
PASSWORD 'xarxa2';
```

```
CREATE ROLE xarxa3 LOGIN NOINHERIT PASSWORD 'xarxa3';
```

```
GRANT g_xarxa TO xarxa3;
```

Todas estas maneras son válidas, y además hemos hecho una cosa rara, y es que **xarxa3** no hereda de sus “padres”.

Si ahora, por ejemplo el usuario **geo** que es el propietario de la Base de Datos **geo** y por tanto de todos sus objetos hace esto:

```
GRANT SELECT ON institutos TO g_xarxa;
```

```
GRANT SELECT ON poblaciones TO xarxa3;
```

Entonces, tanto **xarxa1** como **xarxa2** tendrán acceso a la tabla **institutos**. En cambio **xarxa3** no podrá acceder a esta tabla, porque por defecto no hereda (**NOINHERIT**) los permisos de los roles a los que pertenece. En cambio sí que puede acceder a la tabla **poblaciones**, ya que se le han personalmente, y por tanto no podrán hacerlo **xarxa1** y **xarxa2**.

¿De que sirve hacer miembro de un grupo si no hereda los permisos?

Pues la gracia está en que los permisos no los tiene por defecto, pero sí si lo decimos expresamente en una sesión de **xarxa3** con la sentencia

```
SET ROLE g_xarxa;
```

Entonces en esta sesión de **xarxa3** tendrá únicamente los permisos de **g\_xarxa**. Por lo tanto ahora podrá acceder en la tabla **institutos**, pero no a la de poblaciones !!!

Para dejarlo como estaba podemos hacer una de las 3 siguientes cosas:

```
SET ROLE xarxa3;
```

```
SET ROLE NONE;
```

```
RESET ROLE ;
```

Si borramos el grupo, dejarán de tener permiso sobre él sus miembros, obviamente, pero por lo demás no se ven afectados.

Eso sí, para borrar un grupo, no puede tener permisos sobre ningún objeto, y por tanto los deberemos quitar primero.

```
REVOKE ALL ON institutos FROM g_xarxa;
```

```
DROP ROL g_xarxa;
```

De esta manera, con un rol de grupo (que no se puede conectar) podemos controlar los permisos que se dan a un conjunto de usuarios. Pero hasta ahora no podían “retocar” o borrar las tablas, cosa reservada en principio al propietario del objeto. Pero ¿y si el propietario es un rol de grupo?

***Esto que tenemos a continuación lo dejamos como ejercicio opcional para quien quiera practicar un poco más con Postgres.***

Vamos a ver un último ejemplo en el que un rol será de usuario (podrá conectarse) y también de grupo (habrá otros roles que serán miembros suyos). Para no tener de crear nuevos perfiles en PgAdmin lo hacemos todo en *psql*, entrando inicialmente como *postgres*:

```
psql -U postgres -h 127.0.0.1
```

Creamos un rol que pueda crear bases de datos y roles:

```
CREATE ROL empresa LOGIN PASSWORD 'empresa'
```

```
CREATEDB CREATEROLE;
```

Y ahora nos conectamos como este usuario **empresa** a cualquier base de datos, por ejemplo a la Base de Datos **postgres**. No importa a qué Base de Datos nos conectemos, porque inmediatamente crearemos una nueva Base de Datos, que es

lo que queríamos hacer. La creación de Bases de Datos la veremos mejor en la pregunta 5.

```
#\c posgres empresa
```

Desde aquí creamos una base de datos nueva (el propietario será por tanto **empresa**):

```
#CREATE DATABASE empresa;
```

Nos conectamos a esta base de datos nueva

```
#\c empresa
```

Y ahora creamos dos tablas; el propietario será **empresa**

```
#CREATE TABLE t1 (c1 integer);
```

```
#CREATE TABLE t2 (c1 integer);
```

Ahora creamos dos roles nuevos, ya que el rol **empresa** puede crear. Haremos a que sean de usuario, con el password, que heredan privilegios y que pertenecen al rol **empresa**. Acabamos de utilizar el rol de usuario **empresa** como un rol de grupo!!!

```
CREATE ROLE emp1 INHERIT LOGIN PASSWORD 'emp1'
```

```
IN ROLE empresa;
```

```
CREATE ROLE emp2 INHERIT LOGIN PASSWORD 'emp2'
```

```
IN ROLE empresa;
```

Estos usuarios ya pueden acceder a la base de datos **empresa**, ver las tablas, insertar, borrar e incluso modificar o eliminar las tablas.

Eso sí, si crean alguna tabla, en principio serán ellos los propietarios, y por tanto los otros no las podrán ver.

## 4.5 - Autenticación de usuarios

Cuando un cliente se quiere conectar a la Base de Datos, bien sea por uno de los programas ya vistos (*PSQL*, *PgAdmin*, *PhpPgAdmin* ...) o bien desde otro entorno (como veremos en el último tema), siempre se dice a qué Base de Datos quiere conectarse y como qué usuario PostgreSQL, aunque no lo especifiquemos expresamente.

Por ejemplo, si entramos en una consola y hacemos:

```
psql -U xarxa
```



parece que nada más estamos especificando el usuario, pero también se especifica la Base de Datos (que cogerá por defecto el mismo nombre que el usuario, *xarxa*) y el lugar del servidor (por defecto es el *local* en Linux, y *localhost* en Windows).

Una vez conectado, el usuario tendrá determinados permisos para ver las tablas y otros objetos, como hemos visto en el punto anterior.

**La autenticación** es el proceso por el que el servidor comprueba la identidad del cliente, es decir, que el cliente es realmente quien dice ser, y así poder darle acceso a lo que le corresponda.

PostgreSQL ofrece diversos métodos de autenticación, y se podrán definir diferentes métodos según la Base de Datos, el usuario (o rol de entrada), y la dirección IP de la máquina desde donde se conecta.

Esta configuración se guarda en el fichero *pg\_hba.conf* (HBA: *host-based authentication*), situado normalmente en el mismo directorio que los datos, aunque no es preciso (por ejemplo en la instalación por paquetes de **Lliurex** estará en */etc/postgresql/9.2/main*). En el punto 3 del presente tema ya se comentó esta cuestión.

Este fichero, de texto y con propietario *postgres*, guardará en diferentes líneas las diferentes especificaciones (las líneas en blanco y las comentadas con # no cuentan). En cada línea habrá una serie de campos:

**Tipo\_conexión Base\_datos Usuario Dirección\_ip Máscara\_ip Método \_autenticación**

donde algunos campos pueden estar vacíos según el tipo de conexión (por ejemplo, para la conexión local no tiene sentido la dirección IP).

Cuando un cliente hace una solicitud de conexión, se comprueba este fichero hasta que encuentra la primera línea en la que coincide el tipo de conexión, Base de Datos, usuario, ..., y entonces se aplica el método de conexión especificado.

### Nota

Es muy conveniente realizar una copia de seguridad del fichero *pg\_hba.conf* antes de hacer modificaciones, ya que si la sintaxis del fichero es incorrecta, PostgreSQL no arrancará.

### Tipo de conexión

El tipo de conexión puede ser uno de los siguientes:

<b>LOCAL</b>	La conexión es local, es decir, desde la misma máquina donde está el servidor. Esta conexión se hace, en <b>Linux</b> , utilizando <i>sockets Unix</i> . A parte del usuario que ponemos para conectar, le llega el usuario de S.O. que hace la solicitud de conexión, ya que según qué método de conexión utilizamos le hará falta. No existe este tipo de conexión en Windows
<b>HOST</b>	<b>Indicará</b> conexiones externas por TCP/IP. Por defecto, las conexiones vía TCP/IP están deshabilitadas. Para habilitarlas se debe cambiar un parámetro del fichero de configuración <i>postgresql.conf</i> :  <pre>listen_addresses = '*' # esto indica que escuche de todos los lugares</pre> (y seguramente volver a arrancar el servidor)
<b>HOSTSSL</b>	Para conexiones que utilizan <b>SSL</b> sobre <b>TCP/IP</b>
<b>HOSTNOSSL</b>	Para conexiones que no utilizan SSL sobre TCP/IP

### **Nota muy importante**

La conexión **LOCAL** utiliza sockets Unix. Por tanto en Windows **no estará disponible**. Hasta ahora la hemos hecho por *localhost*, que es una conexión **HOST** en el ordenador local (con IP 127.0.0.1).

Por lo anterior, debemos tener en consideración que **pgAdmin**, en Windows siempre utilizará **localhost**. En cambio en **Linux**, podemos utilizar cualquiera de las dos (poniendo en la dirección del servidor **local** o **localhost**, respectivamente)

Por otra parte, para utilizar la conexión al HOST **127.0.0.1** no hace falta a que esté `listen_addresses` a `'*'`

### **Base de Datos**

La **Base de Datos**; que además de un nombre de una B.D. existente, puede ser algunos de los siguientes:

<b>ALL</b>	Se refiere a todas las bases de datos.
<b>SAMEUSER</b>	Se refiere a la Base de Datos con el mismo nombre que el usuario (mejor dedo, rol de usuario).
<b>SAMEROLE</b>	Se refiere a la Base de Datos con el mismo nombre que un rol de grupo, al que debe pertenecer el usuario. Se puede utilizar también el sinónimo SAMEGROUP.

Si no es ningún de los anteriores, entonces se interpretará como el nombre de una Base de Datos. O más de una, separadas por comas

**Usuario** Podemos utilizar una palabra "comodín":

<b>ALL</b>	Se refiere a todos los usuarios.
------------	----------------------------------

Si no es ALL, entonces se interpretará como el nombre de un usuario. O más de uno, separados por comas. O un grupo, si va precedido por + (entonces se refiere a todos los miembros del grupo). Observamos que a partir de la versión 8.1 todo son roles. Lo que especifica realmente el signo + será los roles que directa o indirectamente dependen de este rol. Si no lleva el signo + se refiere únicamente al rol mencionado, y no sus posibles miembros.

**La dirección IP y la máscara IP:**

Entre las dos especifican una dirección o un rango de direcciones. Evidentemente no tienen sentido cuando el tipo de acceso es local.

Pueden coger dos formatos. Los dos ejemplos siguientes son completamente equivalentes y especifican solo una máquina:

`192.168.3.17 255.255.255.255`

`192.168.3.17/32 # (no deben haber espacios en blanco en medio)`

Mientras que estos dos, también equivalente, especifican todo un rango:

`192.168.3.0 255.255.255.0`

`192.168.3.0/24`

Ya hemos comentado antes el caso especial de localhost. Se especificará la dirección local de cualquiera de estas maneras:

`127.0.0.1 255.255.255.255`

`127.0.0.1/32`

## Método de autenticación

Indicará la manera de autenticarse el usuario especificado, a la base de datos especificada, desde la máquina especificada. Algunos de los métodos son:

<b>TRUST</b>	<p>La conexión siempre se acepta, sea cual sea el usuario, sin comprobar el password.</p> <p>Por ejemplo:</p> <pre>local xarxa ALL trust</pre> <p>hace que todos los usuarios de PostgreSQL puedan entrar sin problemas a la BD <b>xarxa</b></p>
<b>REJECT</b>	<p>La conexión siempre se rechaza.</p> <p>Por ejemplo:</p> <pre>HOST ALL ALL 192.168.4.0 255.255.255.0 reject</pre> <p>rechazará todas las conexiones que se intentan hacer desde este rango de máquinas</p>
<b>PASSWORD</b>	<p>Obliga al cliente a proporcionar la contraseña, que se comprobará con la del usuario. Debemos estar seguros que el usuario tiene contraseña, ya que si no le hemos proporcionado la tendrá a null y por tanto no se podrá conectar. Lo podemos saber consultando la vista <i>pg_shadow</i>, que aunque la clave esté encriptada, nos dice si tiene o no.</p> <p>Si nos hace falta cambiar la contraseña del usuario lo podemos hacer con:</p> <pre>ALTER ROLE nombre PASSWORD 'contraseña'</pre> <p>Cuando la conexión se hace desde un otra máquina, esta contraseña será visible desde la red, estará en texto claro. Según el grado de seguridad de que vulguem aplicar, este método puede ser insuficiente.</p>
<b>MD5</b>	<p>Igual que el anterior, pero la contraseña deberá estar encriptada por MD5, con lo cual ya no es inteligible por la red.</p>
<b>IDENT mapa</b>	<p>Obtiene el nombre de usuario del S.O. (bien el local, por conexiones locales o bien el remoto, para conexiones vía TCP/IP) y comprueba si el usuario tiene permiso por a conectarse dependiendo de lo que va a continuación:</p> <ul style="list-style-type: none"> <li>• Si a continuación va <b>SAMEUSER</b> el usuario de S.O. y de PostgreSQL deben ser idénticos. Entonces, PostgreSQL se fía del S.O. y deja conectarse.</li> <li>• Si no, lo que va a continuación se interpretará como un <i>mapa</i> de correspondencias entre usuarios de S.O. y de PostgreSQL. Este</li> </ul>

	<p>mapa estará en el fichero <i>pg_ident.conf</i>, donde cada línea tendrá la siguiente estructura:</p> <pre>nombre_mapa usuario_sonido usuario_posgres</pre> <p>donde <i>nombre_mapa</i> es lo que aparecía a continuación de <i>ident</i> en el fichero <i>pg_hba.conf</i>.</p> <p>Por ejemplo, vamos a suponer que a la Base de Datos <i>xarxa</i> (el propietario de la que es <i>xarxa</i>) dejaremos conectarse a todos los alumnos del grupo de segundo del ASI, que tienen un usuario en Linux, en Windows, o aún mejor, en el dominio solo los dejaremos conectarse si lo intentan desde una máquina de los Talleres de Informática (192.168.3.0/24). Lo podríamos hacer así:</p> <p>En <i>pg_hba.conf</i> ponemos:</p> <pre>HOST xarxa ALL 192.168.3.0/24 ident tablaasi</pre> <p>Y en <i>pg_ident.conf</i> ponemos:</p> <pre>tablaasi al_1111 xarxa tablaasi al_2222 xarxa tablaasi al_3333 xarxa tablaasi xarxa xarxa</pre> <p>(la última es por si nos habíamos olvidado del propietario, que si no, no se podrá conectar).</p>
<b>LDAP</b>	<b>Utiliza</b> la autenticación de LDAP

### NOTA

No debemos restringir el acceso de postgres a la Base de Datos *template1*, ya unas cuantas utilidades necesitan acceder a esta BD.

Aquí tenemos algunos ejemplos, que supondremos absolutamente independientes, con la explicación inmediatamente después:

```
local ALL ALL reject
```

```
HOST ALL ALL 0.0.0.0 0.0.0.0 reject
```

no se puede conectar nadie, ni de dentro ni de fuera.

```
local ALL ALL trust
```

```
HOST ALL ALL 0.0.0.0 0.0.0.0 reject
```

se pueden conectar todos los de dentro, sin contraseña ni nada, y nadie de fuera

(en Windows, el resultado sería que nadie se puede conectar)

```
local ALL ALL trust
```

```
HOST ALL ALL 127.0.0.1/32 trust
```

```
HOST ALL ALL 0.0.0.0 0.0.0.0 reject
```

se pueden conectar todos los de dentro, sin contraseña ni nada, tanto utilizando sockets Unix como sockets TCP/IP, y ninguno de fuera (ahora en Windows, el resultado sería que los de dentro se pueden conectar y los de fuera no)

```
local ALL postgres password
```

```
local sameuser ALL password
```

postgres se puede conectar a cualquier Base de Datos (con contraseña), y los otros usuarios solo a la suya (con contraseña), y siempre en local

```
local ALL postgres password
```

```
local xarxa xarxa ident
```

```
sameuser
```

```
local empresa ALL trust
```

```
local ALL ALL reject
```

```
HOST ALL postgres 127.0.0.1/32 password
```

```
HOST xarxa xar3,xar4 127.0.0.1/32 password
```

```
HOST xarxa +g_xarxa 127.0.0.1/32 password
```

```
HOST sameuser ALL 127.0.0.1/32 password
```

```
HOST xarxa ALL 192.168.3.0/24 ident
```

```
tablaasi
```

```
HOST ALL ALL 0.0.0.0 0.0.0.0 reject
```

*postgres* se puede conectar a todas (poniendo la contraseña) desde dentro;

*xarxa* a ***xarxa*** (desde una sesión del usuario *xarxa* del S.O.) todos los usuarios se pueden conectar a la BD empresa sin contraseña ni nada (excepto postgres, para que prevalece la primera línea) todas las otras locales son rechazadas

*postgres* se puede conectar a todas (poniendo la contraseña) desde localhost (TCP/IP en la misma máquina)

*xar3* y *xar4* se pueden conectar a la BD *xarxa* con contraseña desde localhost todos los miembros del grupo *g\_xarxa* se pueden conectar a la BD *xarxa* **con** contraseña desde localhost

todos los usuarios del mapa *tablaasi* de *pg\_ident.conf* se podrán conectar en red desde las direcciones 192.169.3.x todas las otras remotas son rechazadas (también se rechazarán las localhost que no estén especificadas anteriormente)

## 5 - Gestión de Bases de Datos

Una Base de Datos es un lugar donde puede haber uno o más de un esquema, y en cada esquema se pueden crear muchos objetos.

La jerarquía será la siguiente: *Servidor -> Base de Datos -> Esquema -> Tabla*.

Un usuario que se conecte a una Base de Datos podrá ver (si tiene permiso) los objetos de los diferentes esquemas de esta Base de Datos. Ya hemos visto que los permisos le podemos dar a 2 niveles: dependiendo de como tengamos configurado *pg\_hba.conf*, se podrá conectar o no; y dependiendo de los permisos dados sobre cada uno de los objetos podrá hacer unas operaciones u otros (SELECT, INSERT, ..., o nada)

Por otra parte llamamos **CLUSTER DE BASE DE DATOS** (*DataBase Cluster*) al lugar donde se guardarán todas las Bases de Datos (a veces abusamos del lenguaje y nos hemos referido a él como Base de Datos; sería la súper Base de Datos donde están todas las mini Base de Datos). Normalmente solo habrá un cluster en un servidor. Nosotros lo tenemos en */var/lib/postgresql/8.3/main* o en *C:\Archivos de Programa\PostgreSQL\8.3\data*.

Pero se puede crear otro lugar, otro cluster. Incluso pueden estar los dos en marcha, eso sí escuchando puertos distintos (sino el primero que se pone en marcha, escuchará todas las peticiones). Lo intentaremos, teniendo en cuenta que ya es administración avanzada, pero quizá no lo consigamos...

Dentro del **cluster**, PostgreSQL guarda todos los objetos de todas las mini - Bases de Datos, pero si miramos el contenido, nos percataremos que lo hace a su modo. Primero observaremos que hay algunos directorios, y quizá los ficheros de configuración (ya los hemos comentado). Las datos están en el subdirectorio *base*, y dentro de él hay más subdirectorios, el nombre de todos ellos son números. Si miramos el contenido de alguno de ellos, veremos que hay ficheros, y el nombre de todos ellos son números.

¿Qué números son estos? Tiene que ver con los **OID** (*Object Identifier*).

Este es el sistema que tiene PostgreSQL para organizarse internamente. Todo objeto creado en un cluster (tablas, vistas, funciones, operadores, triggers, Bases de Datos, Esquemas,...) tiene un número que le identifica unívocamente. Se puede comprobar fácilmente desde PgAdmin cuando vemos las propiedades de cualquier objeto, que siempre tendrá un OID. Incluso como administradores nos puede servir



para seguir la pista a un objeto, consultando determinadas vistas. PostgreSQL tiene un contador de 32 bites encargado de llevar este control y poder ir asignando valores.

Los OID nos pueden servir incluso para identificar una fila de una tabla. Es decir, se aprovecha la estructura de OID para asignar una una fila. Si es así tendremos un campo en la tabla que será de tipo **OID**, y que le asignará automáticamente, a partir del contador OID, un número cada vez que se introduce una fila en la tabla. E iría bien como clave principal.

Sin embargo el uso de OID's en las tablas está en desuso, y PostgreSQL cada vez lo aconseja menos, aunque debe mantener la posibilidad por compatibilidad con versiones anteriores. No olvidemos que aunque el contador OID es grande (32 bits), utilizarlo indiscriminadamente en tablas que puedan llegar a ser muy grandes puede ser peligroso, ya que es vital para su funcionamiento

La manera de crear una tabla con OID's es poniendo la cláusula **WITH OIDS** en la creación

```
CREATE TABLE ....

WITH OIDS | WITHOUT OIDS]

...
```

Si no se pone nada, se coge el valor que tenga el parámetro del fichero de configuración *default\_with\_oids*, que por defecto, en las nuevas versiones, está a *off*

## 5.1 - Bases de Datos: Creación, Modificación y Eliminación

### CREACIÓN

La creación la efectuaremos con la sentencia SQL:

```
CREATE DATABASE nombre
WITH ] [ OWNER [=] nombre_propietario ]
TEMPLATE [=] template ]
ENCODING [=] encoding ]
TABLESPACE [=] tablespace ]
CONNECTION LIMIT [=] núm_conn ] ]
```

En principio el propietario de la Base de Datos será quien la crea, que evidentemente habrá de tener permiso para crear (permiso de **CREATEDB**).

Si somos súper usuarios podremos hacer que el propietario sea otro usuario.

Cuando se crea una Base de Datos en realidad, por ir más rápido, lo que hace es copiarla de otra, una especie de plantilla. Podemos elegir esta plantilla o Base de Datos original (*template1* o *template0*; por defecto *template1*).

### Nota

Aunque podríamos estar tentados de utilizar la cláusula *template* para copiar Base de Datos nuestras (es decir, utilizar las nuestras como *plantilla* y así esperar que se copie bien), no es aconsejable ya que puede producirnos muchos problemas (con transacciones pendientes, ...). Para copiar las tablas y contenidos, ya veremos los comandos oportunos en el apartado *Copia de seguridad y restauración*.

La cláusula *encoding* servirá para especificar un conjunto de caracteres. La lista es larga, aunque parece que la cosa va decantándose por UTF-8. por lo menos así lo aconseja PostgreSQL.

Podemos especificar en cuál **Tablespace** se guardará la Base de Datos y sus objetos (a no ser a que digamos lo contrario). Veremos los **Tablespaces** a continuación.

Por último podemos decir cuántas conexiones concurrentes se pueden hacer a la presente B.D.

Por defecto (-1) no hay límite.

También lo podemos hacer con las utilidades proporcionadas por PostgreSQL, y con una sesión de un usuario de S.O. que coincida con un usuario de PostgreSQL y que tenga permiso para crear B.D.

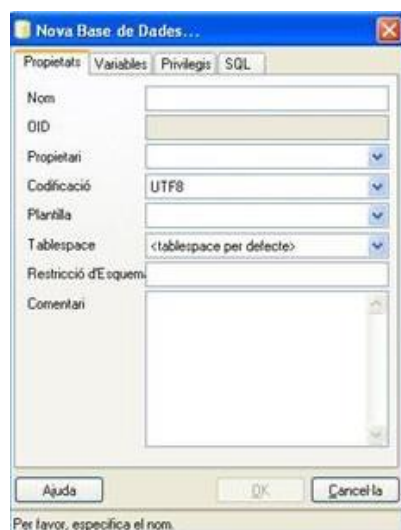
**createdb [ opciones ] nombre**

Las opciones correspondientes a las de la sentencia SQL son

**-O** (*owner*) **-T** (*template*) **-E** (*encoding*) **-D** (*tablespace*)

Aparte tendrá otros (como viéramos con *createuser*) para especificar el HOST, o pedir contraseña, ...

Desde *PgAdmin*



Como se puede comprobar tenemos las mismas opciones.

## MODIFICACIÓN

La sentencia **ALTER DATABASE** permite reconfigurar la Base de Datos, variante los parámetros, o renombrarla.

## ELIMINACIÓN

Para borrar una Base de Datos utilizaremos la sentencia SQL **DROP DATABASE**, o la utilidad **dropdb**.

## 5.2 - Esquemas

En una Base de Datos tendremos como mínimo un *Esquema (Schema)*, aunque podemos tener más de uno con diferentes permisos de acceso. El esquema será, por tanto, una separación lógica de la Base de Datos. En cada esquema podremos crear diferentes objetos (tablas, vistas, funciones, ...).

Siempre que creamos una Base de Datos se crea el primer esquema que será *público*. Eso quiere decir que cualquiera usuario que se conecte a la Base de Datos podrá listar las tablas de este esquema, e incluso *crear sus tablas*, si no hemos controlado el acceso a la Base de Datos (con **pg\_hba.conf**). Así llegaríamos a la contradicción que el propietario de la Base de Datos no podría ver el contenido de estas tablas creadas por otro usuario en su Base de Datos (en el esquema público). Por ejemplo, el usuario **geo** es propietario de la Base de Datos **geo**, donde solo hay un esquema, **público**. Si no impedimos el acceso de otros usuarios (ved pregunta 4.5 Autenticación de usuarios), se podría conectar el usuario **xarxa**, por ejemplo. Este usuario podría ver qué tablas hay creadas, aunque no podría ver su contenido (si no tiene permiso por medio de **Grant**). Pero perfectamente podría crear una tabla en este esquema, **tl\_xarxa**. Entonces el usuario **geo**, no podrá ver el contenido de esta tabla porque no es suya. En la imagen tenemos todo este proceso.

```

lliurex@lliurex-virtual: ~
Fitxer Editar Visualitza Terminal Pestanyes Ajuda
lliurex@lliurex-virtual:~$ psql -h 127.0.0.1 -U xarxa geo
Contraseña para usuario xarxa:
Bienvenido a psql 8.3.0, 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

conexión SSL (cifrado: DHE-RSA-AES256-SHA, bits: 256)
geo=> \d
          Listado de relaciones
Schema | Nombre | Tipo | Dueño
-----+-----+-----+-----
public | comarques | tabla | geo
public | instituts | tabla | geo
public | poblacions | tabla | geo
(3 filas)

geo=> select * from comarques;
ERROR: permission denied for relation comarques
geo=> create table tl_xarxa(c1 varchar(15));
CREATE TABLE
geo=> insert into tl_xarxa values ('Hola');
INSERT 0 1
geo=>
geo=> \c geo geo
Contraseña para usuario geo:
Ahora está conectado a la base de datos «geo» como el usuario «geo».
geo=> \d
          Listado de relaciones
Schema | Nombre | Tipo | Dueño
-----+-----+-----+-----
public | comarques | tabla | geo
public | instituts | tabla | geo
public | poblacions | tabla | geo
public | tl_xarxa | tabla | xarxa
(4 filas)

geo=> select * from tl_xarxa;
ERROR: permission denied for relation tl_xarxa
geo=>

```

Para evitar a que se cuelan, podemos por una banda limitar el acceso por medio del *pg\_hba.conf*. Y por otra limitar el acceso al esquema público o crear otros esquemas.

La sintaxis de creación de un esquema es uno de los dos siguientes:

```
CREATE SCHEMA nombre_esquema [ AUTHORIZATION nombre_usuario ]
```

```
CREATE SCHEMA AUTHORIZATION nombre_usuario
```

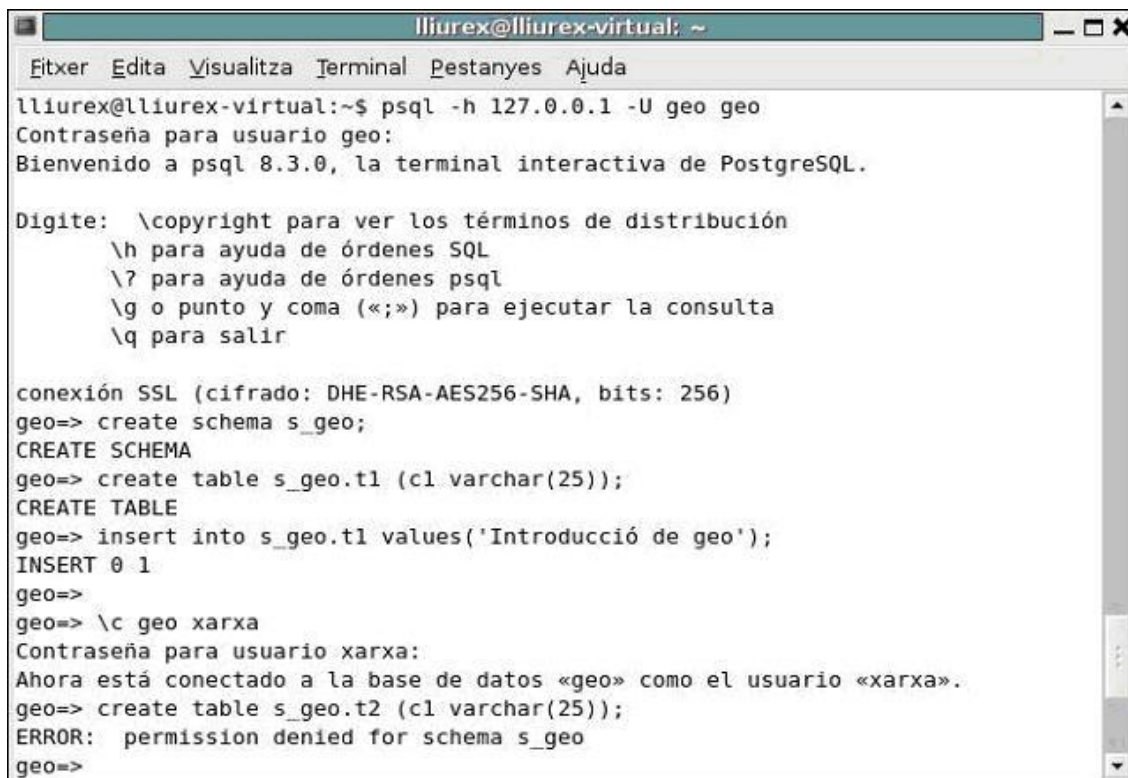
Si no se especifica el nombre del esquema se debe especificar quien tiene la autorización (quien es el propietario), y entonces el nombre del esquema será el mismo del usuario.

Si no se especifica el usuario, el propietario será quien hace la creación.

Para poder crear el esquema en la Base de Datos se debe tener permiso (solo el propietario y superusuarios). A un esquema creado en principio nada más tendrá acceso el propietario, aunque se puede dar permiso por medio de **GRANT**.

Para crear un objeto dentro de un esquema que no sea público (que es el esquema por defecto), se deberá calificar el objeto con el nombre del esquema, es decir *nombre\_esq.nom\_obj*.

Vamos a ver un ejemplo basado en el ejemplo anterior, por ver como limitamos el acceso.



```
lliurex@lliurex-virtual: ~  
Fitxer  Edita  Visualitza  Terminal  Pestanyes  Ajuda  
lliurex@lliurex-virtual:~$ psql -h 127.0.0.1 -U geo geo  
Contraseña para usuario geo:  
Bienvenido a psql 8.3.0, 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  
  
conexión SSL (cifrado: DHE-RSA-AES256-SHA, bits: 256)  
geo=> create schema s_geo;  
CREATE SCHEMA  
geo=> create table s_geo.t1 (c1 varchar(25));  
CREATE TABLE  
geo=> insert into s_geo.t1 values('Introducció de geo');  
INSERT 0 1  
geo=>  
geo=> \c geo xarxa  
Contraseña para usuario xarxa:  
Ahora está conectado a la base de datos «geo» como el usuario «xarxa».  
geo=> create table s_geo.t2 (c1 varchar(25));  
ERROR:  permission denied for schema s_geo  
geo=>
```

Hay manera, por hacer las cosas más cómodas, de cambiar el orden de los esquemas para un usuario (por defecto el primero es el *público*), pero con lo que hemos visto ya hemos cumplido las expectativas del curso.

## 5.3 - Tablespaces

De forma lógica ya hemos visto que todos los objetos de todos los usuarios se organizan en Bases de Datos, y dentro de estas en esquemas.

Pero de forma física, todas las datos se guardan el mismo lugar, y de forma un poco críptica. Es en `/var/lib/postgresql/8.3/main` o `C:\Archivos dePrograma\PostgreSQL\8.3\data`.

Para guardar físicamente de forma más ordenada disponemos de los **TABLESPACES**. Literalmente son espacios de tablas, lugares donde se pueden guardar las tablas y otros objetos. De esta manera se puede poner más de un lugar y destinar todos los objetos de algunos usuarios a un Tablespace distinto al por defecto. Eso podría permitir copias de seguridad independientes, mejor seguridad, ...

Pero todo eso es a nivel físico. Un usuario no notará diferencia entre tener físicamente guardado un objeto en uno o en otro Tablespace.

La sintaxis es la siguiente:

```
CREATE TABLESPACE nombre[ OWNER usuario] LOCATION  
'directorio'
```

donde el nombre del tablespace no puede empezar por *pg\_*ya que está reservado a los tablespaces del sistema; si no se pone el nombre del usuario propietario lo será quien ejecuta la sentencia, pero esta sentencia solo la pueden ejecutar los superusuarios; y el directorio donde hará referencia el tablespace debe existir, estar vacío y ser propietario el superusuario.

Por ejemplo, vamos a crear un Tablespace para guardar las cosas de la Base de Datos empresa (previamente crearemos el directorio como usuario de S.O. *postgres*). En Linux lo podríamos hacer:

```
CREATE TABLESPACE ts_empresa  
  
OWNER empresa  
  
LOCATION '/var/lib/postgresql/8.3/main/empresa';
```

Y en Windows:

```
CREATE TABLESPACE ts_empresa  
  
OWNER empresa  
  
LOCATION 'c:/Archivos de  
Programa/PostgreSQL/8.3/fecha/empresa';
```

Una vez creado el Tablespace podemos hacer que las tablas se guarden en él, siempre que tengamos permiso, está claro. Tal y como está nada más podrían **empresa** y el superusuario. Incluso cuando creamos una nueva Base de Datos, podemos hacer que se guarde en el nuevo tablespace. Entonces, por defecto, cuando se crean tablas y objetos en esta Base de Datos, se guardarán físicamente en el nuevo tablespace, a no ser a que especifiquemos explícitamente otro.

## 5.4 Lugares alternativos

De momento nada más tenemos creada una Base de Datos (de las grandes, la súper Base de Datos, o con palabras técnicas el Database Cluster). Se creó en el momento de la instalación de PostgreSQL, y no especificamos donde se debía guardar. El lugar por defecto, si instalamos por paquetes Ubuntu, es */var/lib/postgresql/9.1/main*, aunque otro lugar habitual es */var/lib/posgres/data* (hay una variable de entorno, PGDATA, que contiene la ruta ).En Windows es **C:\Archivos de Programa\PostgreSQL\9.1\data**

Pero ya hemos visto, en el momento de instalar PostgreSQL a mano, que en la sintaxis de la sentencia de creación podemos especificar otro lugar para una Base de Datos. Vamos a ver este proceso. Intentaremos hacerlo en Linux y en Windows (para que cada uno elija el que quiera), aunque la explicación más larga estará en Linux. Al final tendremos dos clusteres en marcha, es decir dos grandes Bases de Datos, como si fueran dos servidores en marcha.

*Abrimos una consola, y nos conectamos como posgres con la opción – para a que coja el entorno de este usuario (quizá nos toque hacerlo desde root):*

Linux

Windows

```
sudo su - posgres      > runas /user:postgres cmd
```

Inicializamos el nuevo lugar. Podemos colocarlo donde vulguem, nada más debemos controlar que el directorio padre de esta ruta (*/var/lib/postgresql/8.3* en el ejemplo siguiente) debe existir y debe poder escribir el usuario posgres. El subdirectorio (*main2*) se creará durante el proceso, así como toda la estructura interna.

Linux

```
cd /usr/lib/postgresql/8.3/bin
$ ./initdb
/var/lib/postgresql/8.3/main2
```

Windows

```
cd "\Archivos de
Programa\PostgreSQL\8.3\bin"
initdb ../data2
```

podría darse el caso de que el usuario no tuviera permiso de escritura sobre el directorio que estamos indicando. Si es así, podemos dar permiso al de S.O postgres sobre el directorio raíz de PostgreSQL. O si no queremos complicarnos la vida creamos el nuevo lugar en un directorio donde seguro tendremos permiso, por ejemplo C: emp\data2

```
initdb /temp/data2
```

Si todo ha ido bien nos avisará que ya está creado y como ponerlo en marcha. Nada más tendremos en cuenta una cosa: si el anterior cluster (lo que estamos utilizando durante todo el curso) está en marcha, y queremos a que lo esté, deberemos cuidar de poner el nuevo en otro puerto. Normalmente el ya existente estará en el puerto 5432. Por ejemplo podríamos utilizar para el nuevo el 5433. Podemos utilizar para ponerlo en marcha las maneras que viéramos en el punto 2 donde se especifique cuál es el lugar (y también el puerto). Es decir postmaster o pg\_ctl.

Linux

```
postgres -D  
/var/lib/postgresql/8.3/main2 -p  
5433
```

Windows

Dependiendo donde la habíamos habíamos haremos:

```
postgres -D ../data2  
-p 5433
```

O bien

```
postgres -D  
/temp/data2 -p 5433
```

De esta manera está en marcha en esta terminal. Cuando cerramos la terminal o hacemos ctrl-c, pararemos esta Base de Datos. Podríamos hacerlo en segundo plano, ponerlo en marcha nada más arrancar, ...

Aquí tenemos todo lo que hemos hecho en Linux (en Windows debería salir de forma similar), resaltando en **negrita** los comandos escritos. Fijaos que si al poner en marcha la Base de Datos no especificamos el puerto, entonces se percatará que ya está ocupado

```
lliurex@lliurex-virtual:~$ sudo su - postgres  
Password:  
postgres@lliurex-virtual:~$ cd /usr/lib/postgresql/9.1/bin/  
postgres@lliurex-virtual:/usr/lib/postgresql/8.3/bin$ ./initdb  
/var/lib/postgresql/9.1/main2
```



Los archivos de este cluster serán de propiedad del usuario «postgres».

Este usuario también debe ser quien ejecute el proceso servidor.

El cluster será inicializado cono configuración local qcv\_ES.UTF-8.

La codificación mieda omisión ha sido mieda lo tanto definida a UTF8.

initdb: no se pudo encontrar una configuración para búsqueda en texto apropiada  
para la configuración local qcv\_ES.UTF-8

La configuración de búsqueda en texto ha sido definida a «simple».

creando el directorio /var/lib/postgresql/8.3/main2 ... hecho

creando subdirectorios ... hecho

seleccionando el valor para max\_connections ... 100

seleccionando el valor para shared\_buffers/max\_fsm\_camposino ... 24MB/153600

creando archivos de configuración ... hecho

creando base de datos templat1 en /var/lib/postgresql/8.3/main2/base/1 ... hecho

inicializando pg\_authid ... hecho

inicializando dependencias ... hecho

creando laso vistas de sistema ... hecho

cargando laso descripciones de los objetos del sistema ... hecho

creando conversiones ... hecho

creando directorios ... hecho

estableciendo privilegios en objetos predefinidos ... hecho

creando el esquema de información ... hecho

haciendo vacuum a la base de datos templat1 ... hecho

copiando templat1 a template0 ... hecho

copiando templat1 a postgres ... hecho

ATENCIÓN: activando autenticación «trust» para conexiones locales.

Puede cambiar esto editando pg\_hba.conf o usando el parámetro -A

la próxima vez que ejecute initdb.

Completado. Puede iniciar el servidor de bases de datos usando:

```
./postgres -D /var/lib/postgresql/8.3/main2
```

```
./pg_ctl -D /var/lib/postgresql/8.3/main2 -l archivo_de_registro start
```

```
postgres@lliurex-virtual:/usr/lib/postgresql/8.3/bin$ ./postgres -D
```

```
/var/lib/postgresql/8.3/main2 -p 5432
```

LOG: no se pudo enlazar al socket IPv4: La dirección ya está en uso

HINT: ¿Hay otro postmaster corriendo en el puerto 5432? Si no, aguarde unos segundos y

reintento.

WARNING: no se pudo crear el socket de escucha para «localhost»

FATAL: no se pudo crear ningún socket TCP/IP

```
postgres@lliurex-virtual:/usr/lib/postgresql/8.3/bin$ ./postgres -D
```

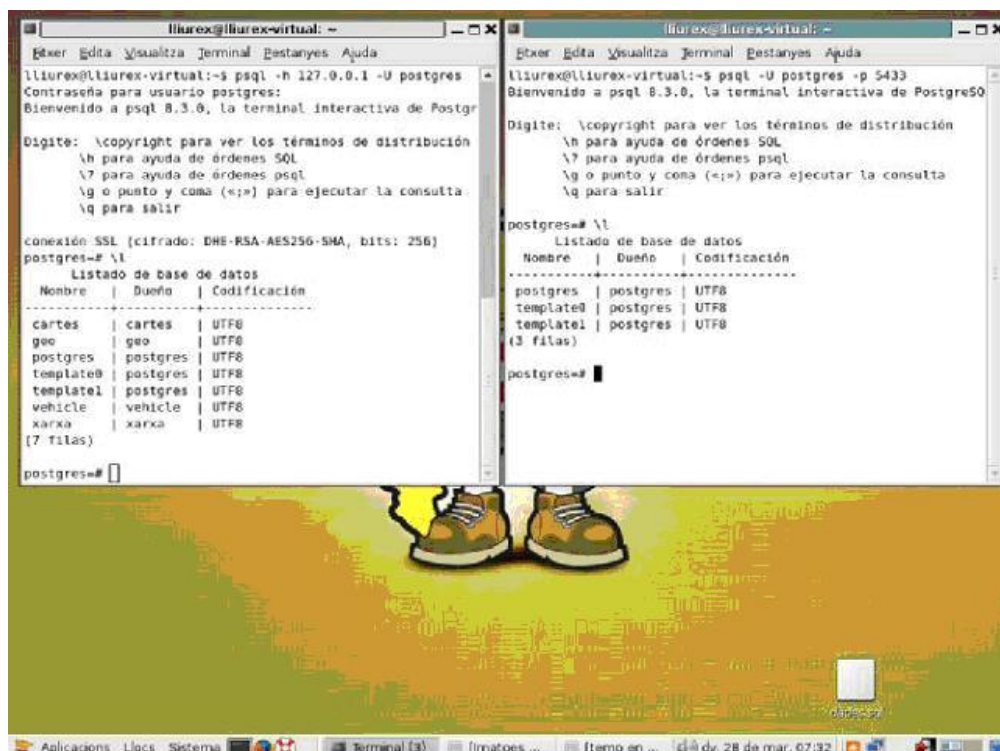
```
/var/lib/postgresql/8.3/main2 -p 5433
```

LOG: el sistema de bases de datos fue apagado en 2008-03-28 07:09:45 CET

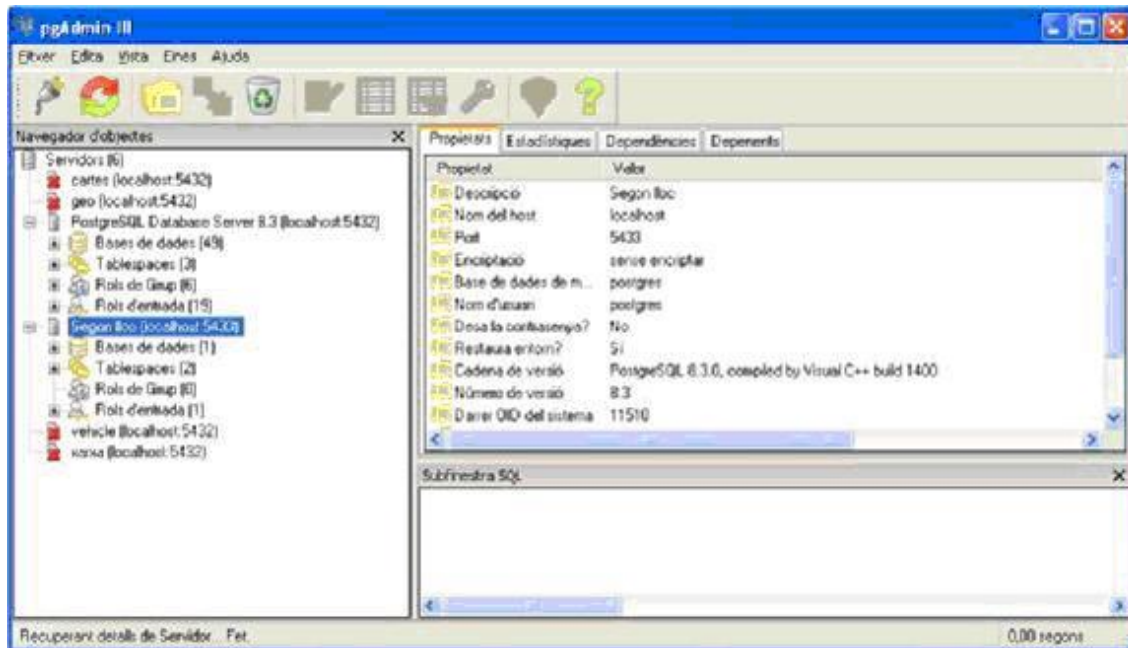
LOG: lanzador de autovacuum iniciado

LOG: el sistema de bases de datos está listo para aceptar conexiones

A continuación hay una imagen donde se ven dos terminales con dos conexiones a los dos clusteres en marcha. Nos conectamos como postgres en los dos, pero observa como las Bases de Datos no son las mismas.



Y ahora otra imagen del PgAdmin (en esta ocasión en Windows) donde se ha creado una conexión al segundo lugar (al puerto 5433). Observáis que el número de objetos (Base de Datos, Roles, Tablespaces, ...) son diferentes.



## 6- Copia de seguridad y restauración

Esta tarea tan fundamental del Administrador de la Base de Datos es muy sencilla de hacer en PostgreSQL, y se puede planificar muy fácilmente, con un sencillo script que se ejecute periódicamente.

Hay básicamente dos maneras de hacer el backup:

Con *pg\_dump*

Copiando los ficheros

Siempre tendremos, sin embargo, la posibilidad de hacerla cómodamente desde **PgAdmin**. Hay otra manera de hacer copia en caliente, que asegura la recuperación de datos prácticamente total. Esta administración *avanzada* la dejaremos para otra ocasión.

### 6.1 - PG\_DUMP

Básicamente el método consiste en generar un fichero de texto con los comandos SQL necesarios para rehacer la Base de Datos tal y como estaba en el momento de hacer el *dump*.

Utilizaremos el programa *pg\_dump*, que tiene la siguiente sintaxis:

```
pg_dump [ opciones ] nombre_bd
```

El resultado irá por la salida estándar, la pantalla. Por tanto seguramente siempre lo gastaremos así:

```
pg_dump nombre_bd > fichero_salida
```

Si queremos que no se guarde en el directorio activo deberemos poner la ruta también. Como cuestión de estilo, podríamos acostumbrarnos a poner siempre la extensión `.sql`. Ahora tendremos en este fichero todas las sentencias SQL para dejar la Base de Datos como estaba.

Lo puede ejecutar cualquiera usuario, y sobre cualquier Base de Datos, pero evidentemente si el usuario no tiene permiso de acceso sobre la B.D. fallará.

Como *pg\_dump* es una aplicación cliente de PostgreSQL se podrá ejecutar desde cualquier lugar, en local o remoto. Y la autenticación será igual que lo que hemos visto hasta ahora.

Estas son algunas de las opciones:

- a** solo guarda los datos
- s** solo guarda la estructura
- d** incluye sentencias INSERT para los datos (sino pondrá sentencias COPY)
- D** lo mismo pero explicitando los nombres de las columnas
- f fichero** envía el resultado en un fichero; si no se especifica, lo envía en la salida estándar.
- F format** indica el formato con que se guardará el fichero (**p** fichero de texto; **t** de tipo tar; **c** de tipo custom, que es la más flexible; con el primero deberemos restaurar utilizando *psql*; con los dos últimos utilizaremos *pg\_restore*)
- n nombre** incluye solo el esquema especificado
- t nombre** incluye solo la tabla especificada

Aparte estarán las habituales opciones de conexión: **-h** (*HOST*), **-p** (*puerto*), **-U** (*usuario*) y **-W**

Esta sería la manera de hacer copia de seguridad de la Base de Datos *xarxa*, guardándola en un fichero del directorio activo renombrado *xarxa.sql*.

```
pg_dump -U xarxa > xarxa.sql
```

Para hacer un *dump* no hace falta parar el servidor.

Una variante es el *pg\_dumpall* que hace el dump sobre todas las bases de datos (seguramente este es el candidato para copias de seguridad periódicas).

Evidentemente lo deberemos hacer como *postgres*, para poder tener acceso a todas las B.D. Tiene el ventaja adicional que también guarda los objetos que no pertenecen a una determinada Base de Datos, como son los usuarios. El inconveniente es que deberemos poner la contraseña de *postgres* *tantas* veces como Bases de Datos tengamos, porque se debe conectar a cada una de ellas.

## 6.2 - Restauración del DUMP

Si el fichero de salida del *dump* (que ahora llamaremos de entrada) no tenía ningún formato (no hemos utilizado la opción *-F*) le ejecutaremos en el *psql*, ya que son comandos SQL. Lo podemos hacer de las dos siguientes maneras:

```
psql nombre_bd < fichero_entrada
```

```
psql -f fichero_entrada nombre_bd
```

Como siempre, si el fichero no está en el directorio activo deberemos poner la ruta. Incluso una vez hemos entrado en la base de datos, podremos hacer:

```
\i fichero_entrada
```

habrá una restricción lógica, pero importante: tanto la Base de Datos como el usuario deben existir (no los crea esta restauración)

Para restaurar todas las B.D., ejecutaremos desde *postgres*, y mejor accediendo a *template1*.

Puestos a hacer virguerías, incluso podemos transvasar la información de una B.D. a otra:

```
pg_dump bd1 | psql bd2
```

que incluso puede estar en otro lugar

```
pg_dump bd1 | psql -h HOST bd1
```

En cambio, si el fichero de salida tenía algún formato, porque hemos utilizado la opción *-Ft* o *-Fc*, habremos de utilizar el programa *pg\_rerestore*, con el que se pueden especificar más cosas:

```
pg_rerestore [ opciones ] nombre_fichero
```

Y en las opciones, entre otros, podremos poner:

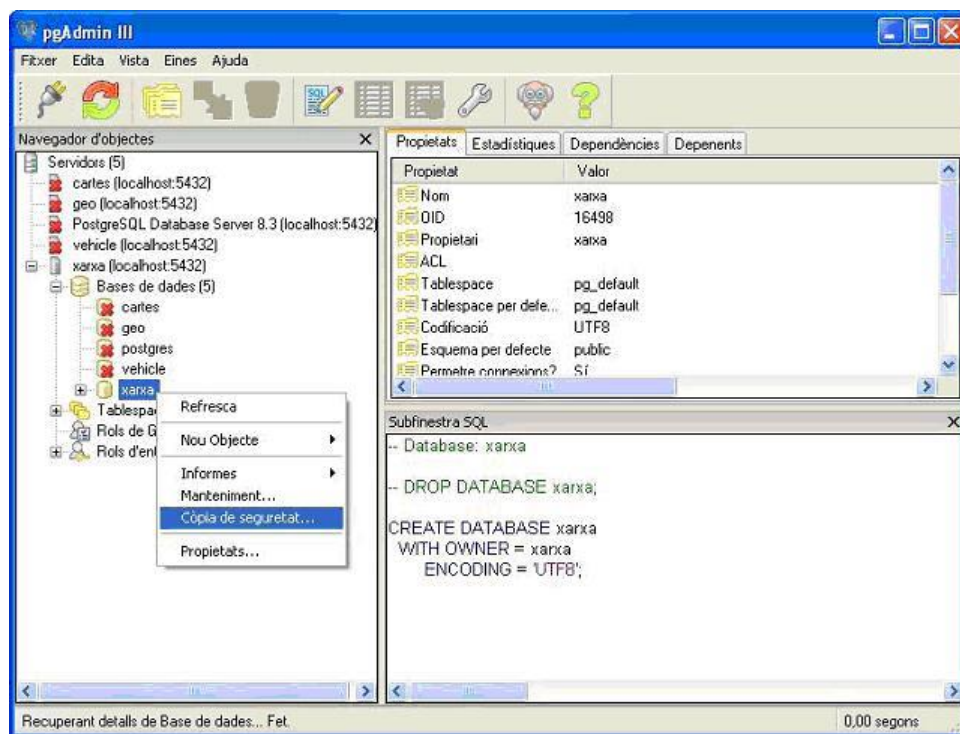
- |           |  |
|-----------|--|
| <b>-a</b> | nada más restaura los datos            |
| <b>-s</b> | nada más restaura la estructura        |
| <b>-c</b> | borra los objetos antes de rellenarlos |

- C** crea la B.D. antes de restaurar en ella
- d nombre\_bd** se conecta a la B.D. y restaura directamente en ella
- F format** indica el formato del fichero de entrada, aunque no hace falta para que lo detecta automáticamente
- t nombre** restaura solo la tabla especificada

Aparte estarán las habituales opciones de conexión: **-h** (*HOST*), **-p** (*puerto*), **-U** (*usuario*) y **-W**

## 6.3 - Backup desde PgAdmin

Desde **PgAdmin** es muy sencillo hacer una copia de seguridad de una Base de Datos determinada. Sencillamente nos situamos encima de ella desde un usuario que tenga permisos, y apretando el botón de la derecha elegimos **Copia de seguridad**.



En la pantalla siguiente, aparte de decir el fichero donde se guardará la copia, tenemos aproximadamente las mismas opciones que en **pg\_dump** (de hecho invocará este programa).





Si elegimos la opción **PLAIN**, el fichero será plano (visible) con las sentencias SQL para reconstruir la Base de Datos. Será la opción más útil.

La operación de restauración, en principio, solo la puede realizar el usuario **postgres**. Para restaurar sobre la Base de Datos elegimos esta opción del menú emergente:



## 6.4 - Backup por copia de ficheros

Habíamos comentado que hay dos maneras de hacer un backup, una utilizando el *dump*, que puede hacer una copia de seguridad en caliente.

La segunda manera consistirá sencillamente en copiar todos los ficheros, "a la brava".

Evidentemente deberemos tener la precaución de hacerlo en frío, sino podemos tener problemas de todo tipo. Por tanto para hacer la copia:

- Parar el servidor:

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

- Copiar todos los ficheros, con la estructura de carpetas. En el entorno Linux lo más conveniente es utilizar el *tar*

```
tar -cf nombre_fichero_tar /var/lib/postgresql/8.3/main
```

que incluso después podríamos comprimir, ...

- Volver a arrancar el servidor:

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

La restauración sería el proceso inverso:

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

```
tar -xf nombre_fichero_tar -C /
```

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