

PostgreSQL

The world's most advanced  
open source database.

**PREVIRED**  
**05/2016**

**EXE**

**EXE**

[www.exe.cl](http://www.exe.cl)

# Administración



# Archivos de configuración



## •postgresql.conf

- Este archivo controla los parámetros generales como la memoria utilizada, el lugar de almacenamiento para las bases de datos, direcciones IP donde el servidor escucha, ubicación de los logs, etc.
- A partir de la versión 9.4 se agregó un archivo adicional postgresql.auto.conf el cual es creado y reescrito cada vez que se ejecuta el comando ALTER SYSTEM SQL.
- Estos parámetros tienen precedencia sobre los definidos en postgresql.conf.

## •pg\_hba.conf

- Controla la seguridad. Administra el acceso al servidor, que usuarios se pueden contactar a las bases de datos, que direcciones IP o grupos de ellas se pueden conectar y también que tipo de esquema de autenticación se espera.

## •pg\_ident.conf

- Si existe, hace un mapeo entre los usuarios autenticados en el SO con usuarios PostgreSQL. Algunos usuarios mapean la cuenta root del SO para que sea un superusuario PostgreSQL



# Archivos de configuración

•SELECT name, setting FROM pg\_settings WHERE category = 'File Locations';

	<b>name</b> <b>text</b>	<b>setting</b> <b>text</b>
1	<b>config_file</b>	/Users/utaladriz/Library/Application Support/Postgres/var-9.5/postgresql.conf
2	<b>data_directory</b>	/Users/utaladriz/Library/Application Support/Postgres/var-9.5
3	<b>external_pid_file</b>	
4	<b>hba_file</b>	/Users/utaladriz/Library/Application Support/Postgres/var-9.5/pg_hba.conf
5	<b>ident_file</b>	/Users/utaladriz/Library/Application Support/Postgres/var-9.5/pg_ident.conf



# postgresql.conf

- `postgresql.conf` controla los parámetros de inicio del servidor PostgreSQL, así como también los parámetros por omisión de las bases de datos nuevas.
- Muchos de estos parámetros pueden ser sobreescritos a nivel de base de datos, usuarios, sesión e incluso en las funciones.
- ```
SELECT name, context , unit ,setting, boot_val, reset_val FROM pg_settings WHERE name IN ( 'listen_addresses', 'max_connections', 'shared_buffers', 'effective_cache_size', 'work_mem', 'maintenance_work_mem') ORDER BY context, name;
```

|   | <code>name</code><br><code>text</code> | <code>context</code><br><code>text</code> | <code>unit</code><br><code>text</code> | <code>setting</code><br><code>text</code> | <code>boot_val</code><br><code>text</code> | <code>reset_val</code><br><code>text</code> |
|---|----------------------------------------|-------------------------------------------|----------------------------------------|-------------------------------------------|--------------------------------------------|---------------------------------------------|
| 1 | <code>listen_addresses</code>          | <code>postmaster</code>                   |                                        | <code>localhost</code>                    | <code>localhost</code>                     | <code>localhost</code>                      |
| 2 | <code>max_connections</code>           | <code>postmaster</code>                   |                                        | <code>100</code>                          | <code>100</code>                           | <code>100</code>                            |
| 3 | <code>shared_buffers</code>            | <code>postmaster</code>                   | <code>8kB</code>                       | <code>16384</code>                        | <code>1024</code>                          | <code>16384</code>                          |
| 4 | <code>maintenance_work_mem</code>      | <code>user</code>                         | <code>kB</code>                        | <code>65536</code>                        | <code>65536</code>                         | <code>65536</code>                          |
| 5 | <code>work_mem</code>                  | <code>user</code>                         | <code>kB</code>                        | <code>4096</code>                         | <code>4096</code>                          | <code>4096</code>                           |



# postgresql.conf



- Si el contexto es postmaster, el cambio del parámetro requiere un reinicio del servicio de PostgreSQL
- Si el contexto es user, basta con un reload (Ya lo veremos).
- El reinicio finaliza las conexiones activas y el reload no.
- unit:indica la unidad de medida utiliza. Esto pide ser confuso porque a veces las unidades son en KB o unidades de 8 KB.
- En el archivo postgresql.conf, la unidad a utilizar puede ser cualquiera; por ejemplo 128 MB es una buena elección y permite una lectura más facil.
- Para leer estos parámetros de una manera más amistosa se pueden usar los comandos Show
  - SHOW effective\_cache\_size;
  - SHOW maintenance\_work\_mem;
  - SHOW ALL.
- setting es el valor actual
- boot\_val es el valor de inicio
- reset\_val es el nuevo valor que toma el parámetro al reiniciar o recargar el servidor
- Desde la versión 9.4 o posterior, los parámetros con el mismo nombre en postgresql.auto.conf tienen precedencia por sobre los de postgresql.conf



# postgresql.conf



- **listen\_addresses.**

- Establece las direcciones IP donde el servidor escucha. El default es localhost o local, pero se puede cambiar a \*, para que este disponible en todas las direcciones IP.

- **port.**

- Por omisión el puerto es el 5432. En Red Hat o CentOS, se debe cambiar el valor de la variable PGPORT en el archivo /etc/sysconfig/pgsql/nombre\_del\_servicio, para cambiar el puerto.

- **max\_connections.**

- Número máximo de conexiones permitido.

- **shared\_buffers.**

- Define la cantidad de memoria compartida por las conexiones para almacenar las páginas a las cuales se ha accedido recientemente. Este parámetro afecta mucho la velocidad de las consultas y su valor debe de ser alto, idealmente un 25% de la memoria disponible. Sin embargo, las mejoras se ven disminuidas con valores mayores a 8 GB. Su cambio requiere de reiniciar el servidor.



# postgresql.conf



## •effective\_cache\_size

- Una estimación de cuanta memoria se espera que esté disponible en el sistema operativo para ser utilizada por PostgreSQL para buffer caches. No tiene un efecto inmediato en el uso de memoria, pero el query planner utiliza este parámetro en los pasos intermedios para chequear si la salida de un query se puede almacenar en memoria. Si el valor es mucho más bajo que la RAM disponible, el planificador hace un uso ineficiente de los índices.
- En un servidor dedicado el valor effective\_cache\_size debe de ser de al menos el 50% de la memoria. Su cambio requiere de un reload del servidor.

## •work\_mem

- Controla la cantidad de memoria máxima utilizada para operaciones como sorting, hash join y table scans. El valor óptimo depende de cómo se este usando la base de datos, cuanto es la memoria disponible y si el servidor es dedicado a PostgreSQL o no.
- Si existen muchos usuarios ejecutando queries simples, el valor puede ser relativamente bajo. Su cambio requiere de un reload

## •maintenance\_work\_mem

- La memoria total dedicada a actividades de mantenimiento del servidor, como vacuum (Eliminación de los registros marcados como eliminados).
- No debería ser mayor a 1GB. Su cambio requiere de un reload



# postgresql.conf



- Estos valores se pueden cambiar a nivel de base de datos, usuarios o en funciones.
  - Por ejemplo se puede cambiar el valor de work\_mem higher al momento de ejecutar queries sofisticados
  - Si existe una función que es intensiva en sort, se puede subir el valor de work\_mem
- Lo nuevo en PostgreSQL 9.4 es que estos parámetros se pueden cambiar con el comando SQL  
**ALTER SYSTEM**
- **EJEMPO:**
  - `ALTER SYSTEM set work_mem = 8192;`
- **Reload del servidor**
  - `SELECT pg_reload_conf();`
- PostgreSQL registra los cambios hechos por el comando SQL en el archivo `postgresql.auto.conf`, y no directamente en el archivo `postgresql.conf`.



# postgresql.conf



- “Metí la pata. Mi servidor no parte”
- Examinar el archivo de log que se encuentra en el directorio raíz de data, o en el subdirectorio pg\_log.
  - Abrir el último archivo y leer la última línea
  - Habitualmente el error se explica por si solo
- Un error común es setear el valor de shared\_buffers demasiado alto.
- Otro error común es que el archivo postmaster.pid exista previamente por algún proceso de shutdown que falló
  - Para solucionar esto basta con borrar el archivo que se encuentra en el directorio data y reiniciar el servidor.



# **pg\_hba.conf**

- El archivo pg\_hba.conf file controla como los usuarios se conectan a las bases de datos PostgreSQL
- Cambios en el archivo requieren un reload o un reinicio
- Configuración típica:

```
# TYPE DATABASE USER ADDRESS METHOD
# IPv4 local connections:
host all all 127.0.0.1/32 ident
# IPv6 local connections:
host all all ::1/128 trust
host all all 192.168.54.0/24 md5
hostssl all all 0.0.0.0/0 md5
# Allow replication connections from localhost, by a user with the
# replication privilege.
#host replication postgres 127.0.0.1/32 trust
#host replication postgres ::1/128 trust
```



# pg\_hba.conf



- Métodos de autenticación.

- Ident
  - Trus
  - Md5
  - pass

- En la versión 9.1 se introdujo el método peer. Los métodos ident y

- peer están disponibles solo en Linux, Unix, y Mac, pero no en Windows.

- Existen otras opciones como gss, radius, ldap, y pam, que no vienen preinstaladas.

- La sintaxis IPv4 permite definir el rango de IPs. Ejemplo 192.168.54.0/24

- La primera parte, en este caso es 192.168.54.0, es la dirección de la red, seguida de una máscara de bits /24 as the bit mask. Esto permite que cualquiera en la subred 192.168.54.0 se conecte en la medida que provea una password correcta en formato md5 hash.

- Para la máscara 32, un solo host (172.20.143.89/32), 24 para una red pequeña (172.20.143.0/24) o 16 para una grande (10.6.0.0/16)

- Regla para SSL.

- hostssl all all 0.0.0.0/0 md5

- Se permiten todas las conexiones en la medida que la password md5 sea válida.

- A partir de la versión 9.0 se puede definir el rango de IPs a las cuales se les permite replicar este servidor.



# pg\_hba.conf



- Para cada petición de conexión, el servicio postgres chequea el archivo pg\_hba.conf, ordenadamente desde arriba hacia abajo.
- Tan pronto como alguna regla de acceso calza, el proceso termina y se permite la conexión.
- Tan pronto como alguna regla de rechazo calza, el proceso termina y se rechaza la conexión.
- Si se llega al final del archivo y no ha calzado ninguna regla la conexión es rechazada.
- El error típico es no colocar las reglas en el orden apropiado.
  - Ejemplo:
  - 0.0.0.0/0 reject antes de 127.0.0.1/32 trust
  - Impide que los usuarios locales se conecten, a pesar de que existe una regla que lo permite.



# **pg\_hba.conf**



- NUEVAMENTE METÍ LA PATA MI SERVIDOR NO FUNCIONA
- “Examinar el archivo de log que se encuentra en el directorio raíz de data, o en el subdirectorio pg\_log.
- Abrir el último archivo y leer la última línea
- El error más común es un error de tipoe o utilizar un esquema de autenticación que no esté disponible .
- Cuando Postgres no puede parsear el archivo pg\_hba.conf, bloquea todos los accesos o sencillamente no parte.



# **pg\_hba.conf**



- PostgreSQL provee muchas opciones para autenticar (Quizás más que cualquier otro servidor)
- Los más populares
  - trust
  - peer
  - ident
  - md5
  - password.
  - También está reject, para rechazar conexiones
- El método de autenticación en pg\_hba.conf es el portero del servidor PostgreSQL
- Los roles, restricciones de acceso, se aplican después de conectarse



# pg\_hba.conf



- **trust**

- El menos seguro de todos, permite que el usuario se auto identifique, sin solicitar password. En la medida que cumpla con las restricciones de IP, y las restricciones de usuarios de la base de datos, se permite la conexión.
- Habitualmente solo se usa para conexiones locales o de usuario único
- El nombre del usuario habitualmente es el mismo usuario que está conectado al SO

- **md5**

- Muy usado envía una clave encriptada con md5

- **password**

- Usa password sin encriptación alguna

- **ident**

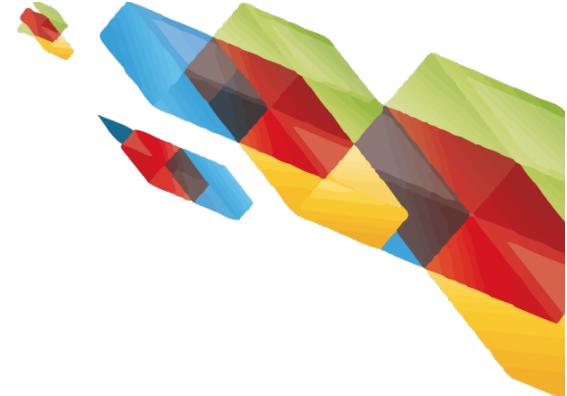
- Usa el archivo pg\_ident.conf para hace un calce con una cuenta del SO

- **peer**

- Usa el nombre del usuario del OS. Disponible para Linux, BSD, Mac OS X, y Solaris
- Solo se puede utilizar para conexiones locales.



# Reload



- Muchos de los cambios mencionados requiere un reinicio del servicio.
- Otros cambios requieren solo de un reload
  - El reload no bota las conexiones activas
  - Para hacer reload por línea de comando
    - `pg_ctl reload -D directorio_de_datos`
- Si PostgreSQL está instalado como servicio en RedHat Enterprise Linux, CentOS, o Ubuntu, se puede dar el siguiente comando:
  - `service postgresql-9.3 reload`
- `postgresql-9.3` es el nombre del servicio, a veces en versiones antiguas el nombre es `postgresql` a secas.
- También es factible hacerlo vía comando SQL si se está conectado como super usuario:
  - `SELECT pg_reload_conf();`



# Conexiones



- Siempre es necesario monitorear las conexiones
- Incluso a veces es necesario terminar una conexión o cancelar lo que estaba haciendo por exceso de consumo de recursos.
- Para operaciones como full backup , restore, o incluso hacer un restore de una tabla en particular se necesita terminar las conexiones activas
- Obviamente terminar una conexión puede traer consecuencias.
- Secuencia
  - 1.- Listado de conexiones y procesos
    - SELECT \* FROM pg\_stat\_activity;
    - Este comando entrega el detalle del query que está corriendo esa conexión, el usuario (username), la base de datos (datname) y fechas asociadas. Lo más importante es el id del proceso.
  - 2.- Para cancelar todas las actividades de esa conexión:
    - SELECT pg\_cancel\_backend(procid)
    - Esto no termina la conexión
  - 3. Terminar la conexión:
    - SELECT pg\_terminate\_backend(procid)
    - Esto igual cancela la actividad de manera abrupta.



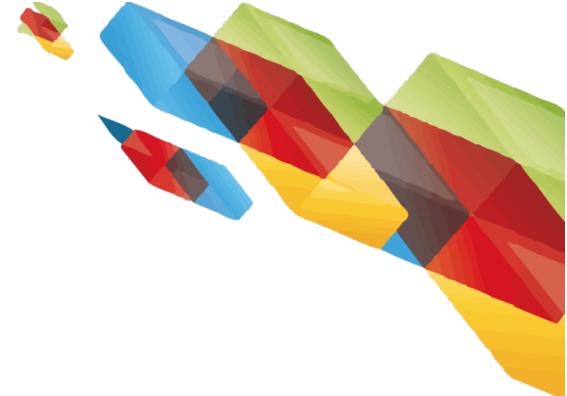
# Conexiones



- PostgreSQL permite el uso de funciones en un SELECT
- pg\_terminate\_backend y pg\_cancel\_backend actúan sobre una conexión
- Sin embargo es posible cancelar o terminar múltiples conexiones
- Versión 9.2 en adelante
  - ```
SELECT pg_terminate_backend(pid) FROM pg_stat_activity WHERE
username ='algun_usuario';
```
- Antes de la versión 9.2
  - ```
SELECT pg_terminate_backend(procpid) FROM pg_stat_activity WHERE
username =algun_usuario';
```
- La vista pg\_stat\_activity ha cambiado bastante después de la versión 9.1, agregando nuevas columnas y cambiando el nombre de otras. Por ejemplo, procpid es ahora pid.



# Roles



- PostgreSQL representa las cuentas como roles
- Los roles que se pueden conectar se denominan login roles.
- Los roles pueden ser miembros de otros roles. Los roles que pueden contener otros roles se denominan group roles. Esta relación puede ser jerárquica
- Los roles que son grupos y se pueden conectar se denominan group login roles.
  - Sin embargo, por razones de mantenimiento y seguridad los DBA's no conceden derechos de login a los roles de grupo
- Un rol puede ser un superusuario
  - Un super usuario tiene acceso completo al servicio PostgreSQL
- Las versiones recientes de PostgreSQL no usan más los términos usuarios y grupos, pero alguna vez se utilizaron por lo que es factible encontrarlos en documentación antigua o en discusiones de foros..
- Por compatibilidad los comandos CREATE USER y CREATE todavía existen, pero se debe usar en vez de ellos el comando CREATE ROLE.



# Roles



- Al momento de iniciar PostgreSQL crea un rol llamado `postgres`. (PostgreSQL también crea una base de datos llamada `postgres`.)
- Se puede establecer un mapeo entre usuarios del SO con el nuevo rol
- Después de instalar PostgreSQL, un debe ingresar con el usuario `postgres` utilizando `psql` o `pgAdmin` y crear roles.
- `CREATE ROLE leo LOGIN PASSWORD 'king' CREATEDB VALID UNTIL 'infinity';`
  - La opción `VALID` es opcional y estable el periodo de expiración para el rol (y la pérdida de privilegios)
  - Por omisión el valor es `infinity`, es decir, no expira
  - La opción `CREATEDB` otorga los derechos de creación de base de datos al nuevo rol.
- Para crear un super usuario se necesita estar conectado como super usuario
  - `CREATE ROLE reina LOGIN PASSWORD 'queen' SUPERUSER VALID UNTIL '2020-1-1 00:00';`



# Roles



- Los roles de grupo generalmente no tienen derechos de login y son contenedores de privilegios para ser usados por otros roles. Esto es una buena práctica o sugerencia.
- Para crear un rol de grupo se utiliza el siguiente comando SQL:
  - CREATE ROLE realeza INHERIT;
  - INHERIT. Esto implica que cualquier miembro del role realeza, herederá sus privilegios
- Para agregar roles al grupo se utiliza el siguiente comando:
  - GRANT realeza TO leo;
  - GRANT realeza TO reina;
- Los roles **LOGIN**, **SUPERUSER**, **CREATEDB**, y **CREATEROLE**, no se heredan como el resto de los roles.
  - Requiere hacer uso explícito de SET ROLE.



# Roles



- En PostgreSQL es factible que un rol grupo no herede sus derechos a sus miembros
  - INHERIT
  - NOINHERIT
- Algunos derechos no pueden ser heredados como los de super usuario.
  - Sin embargo, se puede ejecutar un SET ROLE para obtener dichos privilegios durante la sesión.
- Otra forma de obtener roles más poderosa que SET ROLE es SET SESSION AUTHORIZATION
  - Solo super usuarios pueden ejecutar SET SESSION AUTHORIZATION
  - SET SESSION AUTHORIZATION cambia los valores de current\_user y las variables asociadas a la sesión de usuario session\_user
  - SET ROLE solo cambia la variable current\_user



# Creación de base de datos

- Lo más simple
  - CREATE DATABASE mibd;
- Esto crea una base de datos cuyo dueño es el rol que ejecutó el comando y que es una copia de la base de datos template1
- Cualquier rol con derechos de CREATEDB puede crear nuevas bases de datos.
- Un template, o plantilla, tal como dice su nombre es una base de datos que actúa como modelo.
- Cuando se crea una base de datos, PostgreSQL copia todo los parámetros de la base de datos y la data de la plantilla.
- PostgreSQL viene con dos base de datos plantillas : template0 y
- template1.
- Si no se especifica una planilla se usa por omisión la plantilla template1
- Nunca se debe alterar el template0 porque el modelo base sin nada y que se va a necesitar si es que algo se arruina al modificar los templates. Habitualmente las personalizaciones se hacen sobre el template1.
- No es posible cambiar el encoding y el collation del template1 al momento de crear la base de datos. Si se requiere un encoding o un collation distinto, la base de datos se debe crear desde el template0
- Para crear una base de datos a partir de un template:
  - CREATE DATABASE my\_db TEMPLATE my\_template\_db;



# Creación de base de datos

- Cualquier base de datos puede ser usada como plantilla.
- Cualquier base de datos puede ser marcada como plantilla.
  - Al marcarla como plantilla la base de datos no puede ser editada ni borrada
  - UPDATE pg\_database SET datistemplate = TRUE WHERE datname = mibd';
- Para relaizar cambios o eliminar una base de datos plantilla asta con cambiar datistemplate a FALSE.



# Creación de base de datos



- **Schemas**

- Los esquemas permiten organizar lógicamente una base de datos.
  - Una buena idea es considerar que si se tienen docenas de base de datos es factible agruparlas en una base de datos, pero con diferentes esquemas.

- **Los objetos deben tener un identificador único dentro de un esquema (Nombre)**

- Si todo va en el esquema por omisión, public, tarde o temprano habrá una colisión de nombres.

- Las agrupaciones, esquemas, deben ser hechos de acuerdo al negocio y su contexto.

- Otra manera de organizar la base de datos es creando esquemas por rol.

- Esto es particularmente útil cuando se tienen múltiples clientes cuyos datos deben mantenerse separados.

- La idea es crear el esquema con el mismo nombre que el rol

- La técnica de asignar al esquema el mismo nombre que el rol tiene beneficios adicionales.

- Ya revisaremos la variable de base de datos search\_path.



# Creación de base de datos



- Los nombres de objetos son únicos dentro de un esquema.
- Si tenemos la misma tabla, por ejemplo, clientes, en varios esquemas, ¿Cómo PostgreSQL sabe a cual tabla clientes me refiero?
  - Una respuesta es siempre agregar como prefijo al nombre de la tabla el nombre del esquema
  - Otra alternativa es utilizar la variable search\_path que contiene los nombres de esquemas, para la búsqueda de objetos.
    - Ejemplo si la variable search\_path contiene los valores public, facturacion y crm, al realizar un SELECT \* from clientes, busca la tabla clientes en los esquemas public, facturacion y crm, en ese orden.
- PostgreSQL tiene una variable que contiene el nombre del usuario conectado.
- Puede ser utilizada en search\_path
  - Por ejemplo se puede establecer el valor de search\_path en el archivo postgresql.conf:  
–search\_path = "\$user", public;



# Creación de base de datos



- Otra práctica recomendable es crear esquemas para almacenar las extensiones
- Al momento de instalar una extensión, se crean tablas, funciones, tipos de datos, entre otras cosas y si no se tiene un esquema propio para ello todo queda en el esquema public.
- A modo de ejemplo la extensión PostGIS instala mas de mil funciones. Esto hace difícil la navegación en el esquema public si no se instala en un esquema propio.
- Una recomendación es crear un esquema como:
  - CREATE SCHEMA mis\_extensiones;
  - Luego agregar ese esquema a search\_path:
    - ALTER DATABASE mydb SET search\_path=' "\$user", public, mis\_extensiones ';
- Al momento de instalar las extensiones se debe indicar el esquema de instalación



# Privilegios



- **Privileges**

- Los privilegios, también llamados permisos, permiten un control fino de la seguridad y el acceso.
- Los privilegios operan a nivel de objetos
- Se puede llegar a otorgar permisos sobre columnas, si así se requiere
- La administración de privilegios no es fácil, pero la idea es cubrir los aspectos básicos.

- **Algunos de los privilegios de PostgreSQL a nivel de objetos son:**

- SELECT, INSERT, UPDATE, ALTER, EXECUTE, TRUNCATE y calificadores que operan con GRANT WITH

- Obviamente muchas cosas tienen sentido sobre el objeto que se aplican.



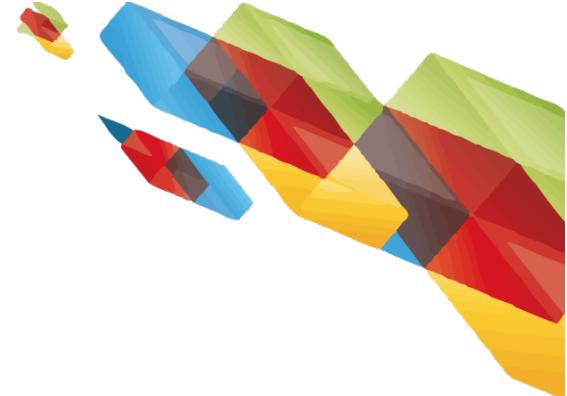
# Privilegios



- 1. PostgreSQL crea un super usuario y una base de datos al momento de instalar, ambos llamados **postgres**.
- 2. Conectarse con el usuario **postgres**
- 3. Antes de crear la primera base de datos, se debe crear un rol que actúe como dueño de la base de datos. Por ejemplo
  - CREATE ROLE administrador LOGIN PASSWORD 'algunapwd' ;
- 3.Crear la base de datos y asignar el dueño:
  - CREATE DATABASE mibd WITH owner = administrador;
- 4. Luego conectarse como administrador y comenzar creando los esquemas



# Privilegios



- El comando GRANT permite asignar privilegios. El uso básico es:

- GRANT privilegio TO rol;

- Algunas cosas a considerar:

- Para otorgar un privilegio el usuario que lo está otorgando lo debe tener y además debe tener el privilegio de hacer grant.

- Algunos privilegios siempre permanecen con el dueño del objeto y no se puede eliminar
    - DROP y ALTER
    - Hacer un grant de esos privilegios a un owner es innecesario

- Para entregar el privilegio de otorgar privilegios se debe agregar

- WITH GRANT OPTION

- Ejemplo:

- GRANT ALL ON ALL TABLES IN SCHEMA public TO administrador WITH GRANT OPTION;

- Para otorgar todos los privilegios se debe usar ALL:

- GRANT SELECT, REFERENCES, TRIGGER ON ALL TABLES IN SCHEMA mi\_esquema TO PUBLIC;



# Privilegios



- El alias **ALL** permite dar permisos sobre todos los objetos en una base de datos o esquema:

–GRANT SELECT, UPDATE ON ALL SEQUENCES IN SCHEMA mi\_esquema TO PUBLIC;

- Para otorgar privilegios a todos los roles se puede usar el alias **PUBLIC**:

–GRANT USAGE ON SCHEMA mi\_esquema TO PUBLIC;

- Algunos privilegios se otorgan por omisión a **PUBLIC**. Por ejemplo:

–CONNECT, CREATE TEMP TABLE, EXECUTE para funciones, y USAGE para lenguajes.

- Para revocar los privilegios se puede utilizar el comando **REVOKE**:

–REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA mi\_esquema FROM PUBLIC;



# Privilegios



- En PostgreSQL 9.0 aparecieron los privilegios por omisión. Esto permite agregar permisos a todos los roles. Esto facilita la administración.

- Ejemplos:

```
-GRANT USAGE ON SCHEMA my_schema TO PUBLIC;  
-ALTER DEFAULT PRIVILEGES IN SCHEMA my_schema  
-GRANT SELECT, REFERENCES ON TABLES TO PUBLIC;  
-ALTER DEFAULT PRIVILEGES IN SCHEMA my_schema  
-GRANT ALL ON TABLES TO mydb_admin WITH GRANT OPTION;  
-ALTER DEFAULT PRIVILEGES IN SCHEMA my_schema  
-GRANT SELECT, UPDATE ON SEQUENCES TO public;  
-ALTER DEFAULT PRIVILEGES IN SCHEMA my_schema  
-GRANT ALL ON FUNCTIONS TO mydb_admin WITH GRANT OPTION;  
-ALTER DEFAULT PRIVILEGES IN SCHEMA my_schema  
-GRANT USAGE ON TYPES TO PUBLIC;
```



# Privilegios



- A diferencia de otros servidores, el dueño de una base de datos PostgreSQL no brinda acceso a todos los objetos de la base de datos, pero permite otorgar los privilegios.
- Otro rol puede crear objetos al cual el dueño no tiene acceso.
- Eso no impide que el dueño pueda eliminar la base de datos completa.
- A menudo se olvida otorgar el derecho de uso con: GRANT USAGE ON SCHEMA or GRANT ALL ON SCHEMA. Aún cuando las tablas y funciones pueden tener derechos asignados a un rol, estas tablas y funciones no son accesibles por el rol si no tiene el privilegio de USAGE.



# Extensiones



- Extensiones, originalmente llamadas contribs (Contribuciones), son add-ons que se instalan en PostgreSQL para extender su funcionalidad.
- Son el mejor ejemplo del modelo colaborativo open source
- A partir de la versión 9.1 se creó el modelo de extensiones.
- Para ver cuales son las extensiones instaladas

```
-SELECT name, default_version, installed_version, left(comment,30)
AS comment FROM pg_available_extensions WHERE installed_version IS
NOT NULL ORDER BY name;
```



# Extensiones

- Para obtener los detalles de una extensión que se encuentre instalada se puede dar el siguiente comando psql:

```
-\dx+ plpgsql
```

- De manera alternativa se puede ejecutar el siguiente comando SQL:

```
SELECT pg_catalog.pg_describe_object(d.classid, d.objid, 0) AS
description FROM pg_catalog.pg_depend AS D INNER JOIN
pg_catalog.pg_extension AS E ON D.refobjid = E.oid WHERE
D.refclassid = 'pg_catalog.pg_extension'::pg_catalog.regclass AND
deptype = 'e' AND E.extname = 'plpgsql';
```

- Esto muestra que es lo está empaquetado en la extensión:

| description | text                             |
|-------------|----------------------------------|
| language    | plpgsql                          |
| function    | plpgsql_validator(oid)           |
| function    | plpgsql_inline_handler(internal) |
| function    | plpgsql_call_handler()           |



# Extensiones



- Las extensiones incluyen objetos de todos los tipos: funciones, tablas, tipos de datos, conversores, lenguajes, operadores, aunque habitualmente su principal contenido son funciones.
- Para que una extensión este disponible en una base de datos se requiere de dos pasos:
  - Primero descargar la instalación e instalarla en el servidor
  - Segundo instalar la extensión en la base de datos
- La instalación de extensiones varía de acuerdo al SO.
  - Se descargan los archivos binarios y se copian al directorio bin y lib, y los scripts al directorio share/extension (Versiones 9.1 y posteriores) o share/contrib (Versiones anteriores a la 9.1).
  - Existen extensiones que vienen preempaquetadas con PostgreSQL
- Para obtener todas las extensiones instaladas
  - `SELECT * FROM pg_available_extensions;`



# Extensiones



• Antes de la versión 9.1, las extensiones se instalaban manualmente ejecutando scripts SQL en la base de datos. Habitualmente el instalador de la extensión descarga los scripts en un directorio de PostgreSQL:

– A modo de ejemplo, en CentOS con la versión 9.0, para ejecutar los scripts SQL para las extensiones de pgAdmin se debe dar el siguiente comando en psql:

- `psql -p 5432 -d postgres -f /usr/pgsql-9.0/share/contrib/adminpack.sql`



# Extensiones



- Para instalar en versiones posteriores a la 9.1
- Se debe usar el comando **CREATE EXTENSION**
- Los beneficios son que no es necesario conocer donde se encuentran los archivos de la extensión, es fácil desinstalar una extensión utilizando el comando **DROP EXTENSION**
- La instalación de PostgreSQL incluye las extensiones más populares, por lo que instalar una extensión se remite a ejecutar el comando **CREATE EXTENSION**.
- Por ejemplo para crear la extensión **fuzzystrmatch** basta con el siguiente comando SQL:  
    –CREATE EXTENSION fuzzystrmatch;
- Vía psql:  
    –psql -p 5432 -d mydb -c "CREATE EXTENSION fuzzystrmatch;"
- Las extensiones basadas en C deben ser instaladas como super usuario. La mayoría de las extensiones son de este tipo
- Siempre se sugiere que las extensiones sean creadas en un esquema aparte:  
    –CREATE EXTENSION fuzzystrmatch SCHEMA my\_extensions;



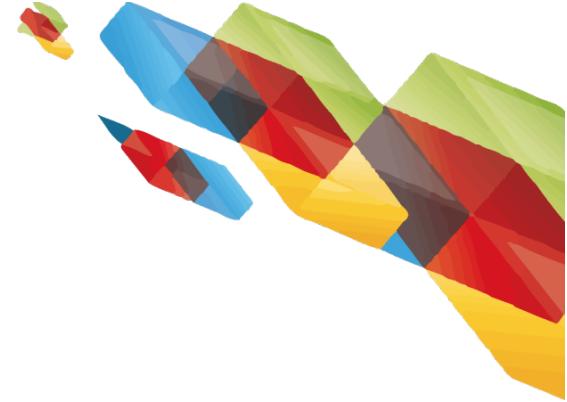
# Extensiones



- Al usar una versión de Postgres anterior a la 9.1 y restaurar esa base de datos en una versión 9.1 o posterior, todas las extensiones continúan funcionando, sin necesidad de ajustar o cambiar algo..



# Extensiones



- **Extensiones populares**

- **btree\_gist**

- Provee operadores de indices GiST basados en B-Trees

- **btree\_gin**

- Provee operadores de GIN basados en B-Trees

- **Postgis**

- Proporciona funcionalidades GIS a PostgreSQL transformandola en una base de datos geoespacial. PostGIS es una extensión mayor, y agrega más de 800 funciones, tipos, e índices espaciales.



# Extensiones



- **fuzzystrmatch**

- Una extensión liviana para búsquedas fuzzy tipo soundex, levenshtein, y meta phone para strings

- **Hstore**

- Transformar a PostgreSQL en una base de datos NOSQL de tipo key value.

- **pg\_trgm (trigram)**

- Otra biblioteca de búsqueda difusa de strings que agrega el operador ILIKE pero con capacidad de usar índices

- **dblink**

- Permite consultar a PostgreSQL desde otro servidor. Es el mecanismo que antecede a los foreign data wrappers que aparecieron en la versión 9.3.

- **pgcrypto**

- Provee herramientas de encriptación basadas en PGP



# Extensiones



- Estas extensiones ya están incorporadas en PostgreSQL

- **tsearch**

- Un conjunto de índices, operadores, diccionarios y funciones que permiten hacer búsquedas de tipo full text

- **xml**

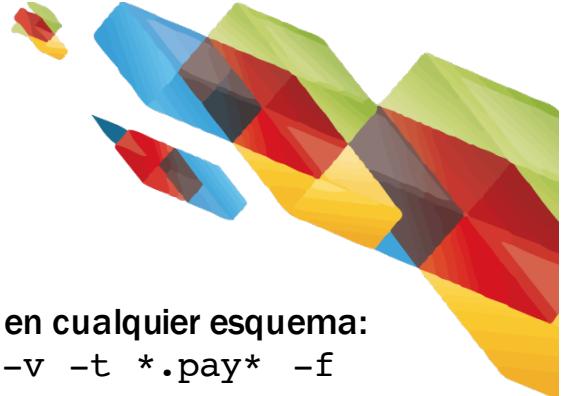
- Una extensión que agrega el tipo de dato XML, funciones y operadores relacionados
  - XML es parte integral de PostgreSQL y además es ANSI SQL



# Backup y restore



- PostgreSQL viene con dos utilitarios para realizar respaldos: pg\_dump y pg\_dumpall.
- Ambos se encuentran en el directorio bin
- pg\_dump se utiliza para respaldar una base de datos en particular
- pg\_dumpall para respaldar todas las bases de datos y los parámetros globales del servidor.  
Pg\_dumpall necesita ser ejecutado como super usuario
- Pg\_dump puede respaldar selectivamente tablas, esquemas y bases de datos. Puede respaldar a SQL plano, pero también puede generar archivos TAR,
- Los siguientes son los ejemplos de respaldo más típicos para una base de datos PostgreSQL:
  - Crear un archivo comprimido de una base de datos:
    - pg\_dump -h localhost -p 5432 -U usuario -F c -b -v -f archivo.backup mi\_base\_datos
  - Crear un archivo en texto plano que incluya la instrucción de creación de la base de datos:
    - pg\_dump -h localhost -p 5432 -U usuario -C -F p -b -v -f archivo.txt mi\_base\_datos



# Backup y restore

- Crear un archivo comprimido tar para todas las tablas que contienen “pay en cualquier esquema:  

```
-pg_dump -h localhost -p 5432 -U usuario -F c -b -v -t *.pay* -f  
pay.backup mibd
```
- Crear un archivo comprimido par los esquemas hr y payroll:  

```
-pg_dump -h localhost -p 5432 -U usuario -F c -b -v -n hr -n payroll  
-f hr.backup  
-mibd
```
- Crear un archvio compromido para todas las bases de datos menos para el esquema public  

```
-pg_dump -h localhost -p 5432 -U usuario-F c -b -v -N public -f  
all_sch_except_pub.backup mibd
```
- Para crear archivos SQL a partir de tablas:  

```
pg_dump -h localhost -p 5432 -U usuario -F p --column-inserts -f  
select_tables.backup mydb
```



# Backup y restore



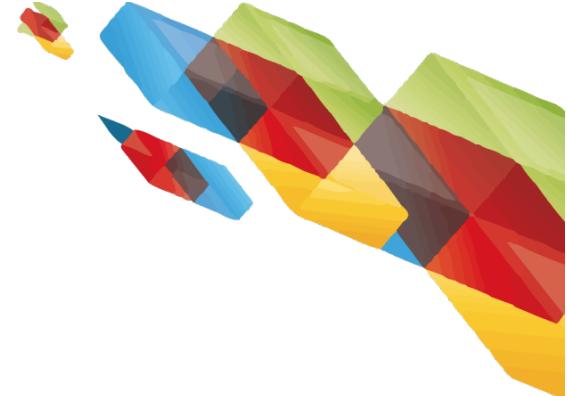
- La opción de directorio partió con la versión 9.1. Esta opción realiza el backup en un directorio donde cada tabla es un archivo. Esta opción realiza un backup de cada tabla en un archivo separado (GZIP)

```
-pg_dump -h localhost -p 5432 -U usuario -F d -f  
/somepath/directorio mydb
```

- Una opción de backup en paralelo se introdujo en la versión 9.3. Con la opción (-j) de los jobs.  
Asignando un valor a -jobs=3 ejecuta tres backups en paralelo.
- El backup en paralelo solo hace sentido en backup en directorio



# Backup y restore



- `pg_dumpall` permite backuppear todos los archivos en formato SQL

- Algunos ejemplos.

- Para respaldar valores globales y roles:

```
-pg_dumpall -h localhost -U postgres --port=5432 -f myglobals.sql --  
globals-only
```

```
-pg_dumpall -h localhost -U postgres --port=5432 -f myroles.sql --  
roles-only
```



# Backup y restore



- Hay 2 maneras de restaurar datos en PostgreSQL
  - Usar psql para restaurar los respaldos en texto plano generados por pg\_dumpall o pg\_dump
  - Usar el utilitario pg\_restore para restaurar respaldos TAR y de directorio
- Se utiliza psql para restaurar respaldos de tipo plain-text SQL:
  - Plain-text SQL: Archivos de texto con comandos SQL
  - Para restaurar un full backup e ignorar los errores:
    - `psql -U postgres -f myglobals.sql`
  - Para parar si es que ocurre un error
    - `psql -U postgres --set ON_ERROR_STOP=on -f myglobals.sql`
  - Para restaurar una base de datos específica
    - `psql -U postgres -d mibd -f select.sql`



# Backup y restore

- Al realizar un backup con `pg_dump` y formato tar, o directory, se puede utilizar `pg_restore` para restaurar la base de datos.
- Se puede utilizar la opción `-j` para controlar la cantidad de threads a utilizar. Esto permite restaurar la base de datos en paralelo
- `pg_restore` puede también restaurar de manera selectiva (Esto lo veremos más adelante)
- Para utilizar `pg_restore` primero se debe crear la base de datos con un comando SQL:  
`--CREATE DATABASE mibd;`
- Luego restaurar:  
`--pg_restore --dbname=mibd --jobs=4 --verbose mibd.backup`
- Si la base de datos es la misma sobre la cual se realizó el respaldo, se puede crear y restaurar en un paso:  
`--pg_restore --dbname=postgres --create --jobs=4 --verbose mibd.backup`
- Al usar la opción `--create`, el nombre de la base de datos es el mismo sobre el cual se hizo respaldo. No se puede cambiar el nombre. En la opción `--dbname`, el nombre de la base de datos debe ser distinto de la base de datos que se está restaurando. Se utiliza habitualmente `postgres`.



# Backup y restore

- En la versión 9.2 o posterior, se puede hacer uso de la opción –section para restaurar solo la estructura y no los datos.

- Util para generar una plantilla a partir de una base de datos

- Creamos la base de datos

- CREATE DATABASE mydb2;

- Luego usamos pg\_restore

- pg\_restore --dbname=mydb2 --section=pre-data --jobs=4 mibd.backup



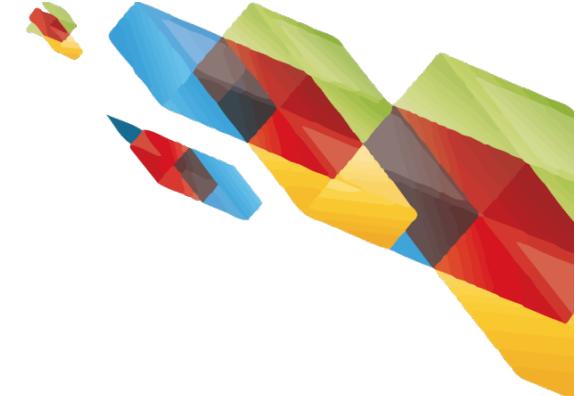
# Tablespaces



- PostgreSQL utiliza los tablespaces para asignar nombres lógicos a ubicaciones y espacios físicos en disco
- Al instalar PostgreSQL automáticamente crea dos tablespaces
  - pg\_default, que almacena los datos del usuario
  - pg\_global que almacena los datos del sistema
- Ambos tablespaces están ubicados en el mismo directorio de datos por omisión del servidor.
- Los tablespaces se pueden crear en cualquier disco del servidor.
- Cualquier objeto puede ser asignado a un tablespace en particular
- Los objetos se pueden mover de un tablespace a otro.
- Para crear un tablespace se debe proporcionar un nombre y un directorio físico del servidor:
- Ejemplo en Windows
  - CREATE TABLESPACE secondary LOCATION 'C:/pgdata94\_secondary';
- Para Unix o Linux
  - CREATE TABLESPACE secondary LOCATION '/usr/data/pgdata94\_secondary';



# Tablespaces



- Para asignar una base de datos a otro tablespace:

- ALTER DATABASE mibd SET TABLESPACE secundario;

- Para mover una tabla

- ALTER TABLE mi\_tabala SET TABLESPACE secundario;

- Una característica nueva en PostgreSQL 9.4 es la capacidad de mover un grupo de objetos desde un tablespace a otro. Si el rol es un superusuario se puede mover todos los objetos. Si no es un superusuario solo se pueden mover los objetos de los cuales el rol es dueño.

- ALTER TABLESPACE pg\_default MOVE ALL TO secundario;

- Durante el movimiento las tablas están bloqueadas.



# Algunas recomendaciones

- **Borrando archivos:**

- Existen directorios como pg\_log, pg\_xlog, y pg\_clog todos parecen logs y en teoría “Borrables”
  - El directorio pg\_log se puede eliminar (se regenera), pero no así pg\_xlog que no se debe borrar nunca (Es el log de transacciones)

- **Ojo con los antivirus de Windows, borran binarios de PostgreSQL**

- **La cuenta de usuario de postgres a nivel de SO no necesita permisos administrativos excepcionales o root.**

- **Si se tienen trabajos tipo batch que acceden a archivos que están en otros directorios permisos adicionales se pueden requerir.**

- **No es la idea establecer un valor de shared\_buffers igual a la memoria física RAM**

- Si se hace de esa manera el servidor podría no partir.
  - Bajo Windows de 32 bit más de 512 MB produce inestabilidad.
  - Bajo Windows de 64 bits el valor puede exceder 1GB sin problemas
  - En algunos sistemas Linux el valor de shared\_buffer no puede exceder el valor de la variable SHMMAX, el cual es bastante bajo.



# Algunas recomendaciones



- Al iniciar PostgreSQL en un puerto que está en uso los errores se verán reflejados en los archivos pg\_log de la siguiente forma: make sure PostgreSQL is not already running.
- Esto sucede por:
  - Se inició previamente un servicio
  - Existe otro servicio usando el puerto
  - Un servicio postres se inició y dejó un archivo huérfano postgresql.pid en el directorio de datos.
- Cuando todo falla matar todos los procesos PostgreSQL y eliminar el archivo postgresql.pid es una opción