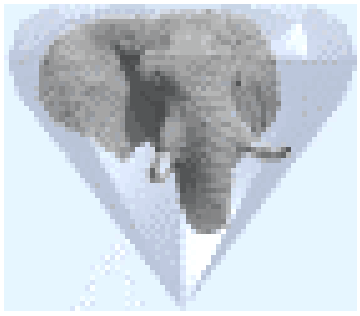


Manual de PostgreSQL

Curso “Manejadores de Bases de Datos”

Fernández Guzmán Helder Octavio
Auxiliar de Investigación
Grupo CIDAI-UMSS

A manera de Prologo

Este texto de respaldo fue desarrollado teniendo en cuenta la necesidad de tener una guía en la cual se puedan realizar consultas rápidas sobre puntos específicos como: Instalación de PostgreSQL, Creación de Base de Datos, Tareas básicas de Administración, Consultas y Comunicación con Aplicaciones (ODBC, JDBC), etc.

Consideramos que aquí puede encontrar la información necesaria para empezar con un nivel adecuado el uso de esta herramienta (PostgreSQL).

Helder O. Fernández Guzmán
helder@memi.umss.edu.bo
Auxiliar de Investigación.
Grupo CIDAI - UMSS

Convenciones

sitio se puede interpretar como la máquina en la que está instalada PostgreSQL. Dado que es posible instalar más de un conjunto de bases de datos PostgreSQL en una misma máquina, este término denota, de forma más precisa, cualquier conjunto concreto de programas binarios y bases de datos de PostgreSQL instalados.

El *superusuario* de PostgreSQL es el usuario llamado p*ostgreSQL* que es dueño de los ficheros de la bases de datos y binarios de PostgreSQL. Como superusuario de la base de datos, no le es aplicable ninguno de los mecanismos de protección y puede acceder a cualquiera de los datos de forma arbitraria. Además, al superusuario de PostgreSQL se le permite ejecutar programas de soporte que generalmente no están disponibles para todos los usuarios.

Nota: *El superusuario de PostgreSQL no es el mismo que el superusuario de Unix (que es conocido como root). El superusuario debería tener un identificador de usuario (UID) distinto de cero por razones de seguridad.*

El *administrador de la base de datos* (*database administrator*) o DBA, es la persona responsable de instalar PostgreSQL con mecanismos para hacer cumplir una política de seguridad para un sitio.

El postmaster es el proceso que actúa como una puerta de control (clearing-house) para las peticiones al sistema PostgreSQL. Las aplicaciones frontend se conectan al postmaster, que mantiene registros de los errores del sistema y de la comunicación entre los procesos backend

1. Introducción

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

1.1. ¿Que es PostgreSQL?

PostgreSQL es una mejora del sistema gestor de bases de datos POSTGRES, un prototipo de investigación de DBMS de nueva generación. Mientras PostgreSQL mantiene los potentes modelos de datos y riqueza de tipos de POSTGRES, reemplaza el lenguaje de consulta PostQuel con un subjuego extendido de SQL. PostgreSQL es gratis y las fuentes son accesibles públicamente.

El desarrollo de PostgreSQL está siendo realizado por un equipo de desarrolladores de Internet suscritos a la lista de correo de PostgreSQL. El coordinador actual es Marc G. Fournier (scrappy@postgreSQL.org). Este equipo es el actual responsable de todo el desarrollo actual y futuro de PostgreSQL.

El nombre original del software en Berkeley fue postgreSQL. Cuando se le añadió la funcionalidad SQL en 1995, su nombre se cambió a Postgres95. El nombre volvió a ser cambiado a finales de 1996 a PostgreSQL.

Se pronuncia Post-Gres-Q-L.

2. Características de PostgreSQL

Los sistemas de mantenimiento de Bases de Datos relacionales tradicionales (DBMS,s) soportan un modelo de datos que consisten en una colección de relaciones con nombre, que contienen atributos de un tipo específico. En los sistemas comerciales actuales, los tipos posibles incluyen numéricos de punto flotante, enteros, cadenas de caracteres, cantidades monetarias y fechas. Está generalmente reconocido que este modelo será inadecuado para las aplicaciones futuras de procesado de datos. El modelo relacional sustituyó modelos previos en parte por su "simplicidad espartana". Sin embargo, como se ha mencionado, esta simplicidad también hace muy difícil la implementación de ciertas aplicaciones.

PostgreSQL ofrece una potencia adicional sustancial al incorporar los siguientes cuatro conceptos adicionales básicos en una vía en la que los usuarios pueden extender fácilmente el sistema:

- ✓ clases
- ✓ herencia
- ✓ tipos
- ✓ funciones

Otras características aportan potencia y flexibilidad adicional:

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

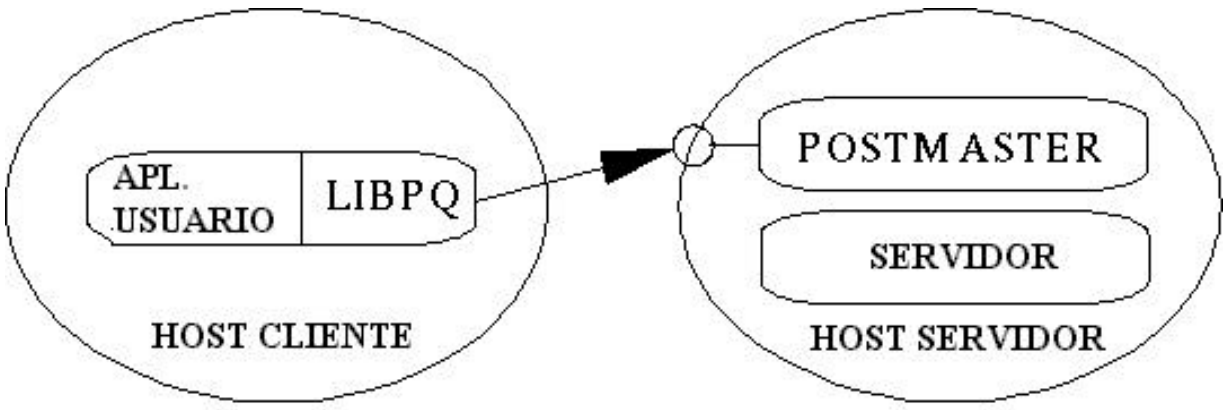
Estas características colocan a PostgreSQL en la categoría de las Bases de Datos identificadas como *objeto-relacionales*. Nótese que éstas son diferentes de las referidas como *orientadas a objetos*, que en general no son bien aprovechables

para soportar lenguajes de Bases de Datos relacionales tradicionales. PostgreSQL tiene algunas características que son propias del mundo de las bases de datos orientadas a objetos. De hecho, algunas Bases de Datos comerciales han incorporado recientemente características en las que PostgreSQL fue pionera.

2.1. Arquitectura

Es necesario conocer la arquitectura básica del sistema PostgreSQL (cómo interactúan las partes de PostgreSQL). En la jerga de las bases de datos, PostgreSQL utiliza un simple modelo cliente/servidor de "proceso por usuario". Una sesión de PostgreSQL consiste en los siguientes procesos Unix (programas) cooperando:

- Un proceso demonio supervisor (postmaster),
- la aplicación de interfaz del usuario (frontend en inglés) (por ejemplo, el programa psql), y
- los uno o más procesos servidores de acceso a la base de datos (backend en inglés) (el proceso PostgreSQL mismo).



Un único Postmaster maneja una colección dada de bases de datos en un único host (server host). Tal colección se denomina una instalación o un sitio. Las aplicaciones de frontend que quieren acceder a una base de datos dada en una instalación realizan llamadas a la librería. La librería envía el requerimiento del usuario a través de la red al postmaster (*Cómo se establece una conexión*(a)), quien en su turno arranca un nuevo proceso

servidor de backend (*Cómo se establece una conexión*(b)). y se conecta el proceso cliente al nuevo servidor (*Cómo se establece una conexión*(c)). A partir de aquí, el proceso cliente y el servidor se comunican entre ellos sin intervención del postmaster. En consecuencia, el proceso postmaster está siempre corriendo, esperando llamadas, mientras que los procesos cliente y servidor vienen y van. La librería libpq permite a un único proceso cliente tener múltiples conexiones con procesos servidores. Sin embargo, la aplicación cliente sigue siendo un proceso mono-hebra. Las conexiones con multihebrado cliente/servidor no están soportadas en libpq. Una implicación de esta arquitectura es que el postmaster y los servidores siempre corren en la misma máquina (el servidor de base de datos), mientras que el cliente puede correr en cualquier sitio.

Se Debe tener esto en cuenta, ya que los ficheros que pueden estar accesibles en una máquina cliente, pueden no estarlo (o estarlo sólo con un nombre de fichero diferente) en la máquina servidor. También se debe tener en cuenta que postmaster y los servidores PostgreSQL corren bajo el user-id del "superusuario" de PostgreSQL. El superusuario de PostgreSQL definitivamente ¡no debe de ser el superusuario de Unix, "root"! En cualquier caso, todos los ficheros relacionados con una base de datos deben encontrarse bajo este superusuario de PostgreSQL.

3. Instalación y Configuración en Linux

Si aún no tiene la distribución de PostgreSQL, puede obtenerla (por ejemplo) en: ***ftp://ftp.rge.com//pub/database/postgresql/source/v7.2/postgresql-7.2.tar.gz***

3.1 Antes de comenzar

Para compilar PostgreSQL se requiere la utilidad GNU make. Otras utilidades similares *no funcionarán* en este caso. En los sistemas GNU/Linux, GNU make es la herramienta por defecto. En otros sistemas puede que encuentre que la herramienta GNU make se encuentre instalada con el nombre "gmake". De aquí en adelante, utilizaremos este nombre para referirnos a GNU make. Para probar GNU make teclee:

```
$ gmake -version
```

Si necesita obtener GNU make, lo puede encontrar en ftp://ftp.gnu.org.

En http://www.PostgreSQL.org/docs/admin/ports.htm puede encontrar información actualizada sobre las plataformas soportadas. En general, la mayoría de la plataformas compatibles con Unix que utilicen bibliotecas actualizadas debería ser capaz de ejecutar PostgreSQL. En el subdirectorio doc de la distribución existen varios documentos del tipo LEAME y otros con Preguntas de Uso Frecuente (FAQ en inglés) específicos para esa distribución, que pueden resultarle útiles si está teniendo problemas.

La cantidad mínima de memoria que se requiere para ejecutar PostgreSQL es de sólo 8 MB. Sin embargo, se verifica una notable mejora en la velocidad cuando ésta se expande a 96 MB o más. La regla es que, por más memoria que se instale en un sistema, nunca será demasiada. Verifique que exista suficiente espacio libre en el disco. Se Necesita

alrededor de 30 MB para los archivos con el código fuente durante la compilación, y unos 5 MB para el directorio de instalación. Una base de datos vacía ocupa aproximadamente 1 MB. De no estar vacía, la base ocupará unas cinco veces el espacio que ocuparía un archivo de texto que contuviera los mismos datos.

Para revisar el espacio libre en el disco, utilice:

```
$ df -k
```

3.2. Procedimiento de Instalación

Instalación de PostgreSQL (v7.2, Linux redhat)

Lo primero, es copiar los fuentes a un lugar donde se puedan descomprimir, en /usr/local/src por ejemplo:

```
$ cp postgresql-7.2.tar.gz /usr/local/src
```

luego, se procede a descomprimir:

```
$ tar zxvf postgresql-7.2.tar.gz
```

Una vez hecho esto se ha creado un nuevo directorio llamado postgresql-7.2, moverse a él (cd /usr/local/src/postgresql-7.2), ejecutar ..

```
$./configure
```

Luego de configurar el paquete para ser compilado, proceder a compilarlo..

```
$ gmake
```

Si se muestra la línea de texto “All of PostgreSQL is successfully made. Ready to install” al final de la compilación, ésta ha sido exitosa. Sólo queda instalar el producto en su destino final, para ello ejecutar ..

```
$ su
$ gmake install
```

Esto instalará los binarios y archivos de configuración necesarios en el directorio /usr/local/pgsql (o aquel directorio que se halla pasado como parámetro en el comando ./config). Es necesario ahora indicarle a nuestro sistema el lugar donde se encuentran las librerías compartidas de PostgreSQL. Editar el archivo

```
/etc/ld.so.conf
```

y agregar, al final del archivo, la línea

```
/usr/local/pgsql/lib
```

luego actualizar, ejecutando el comando

```
$/sbin/ldconfig
```

ahora, necesitamos crear al superusuario de postgreSQL ‘postgreSQL’ :

```
$ adduser postgreSQL
```

también podríamos asignarle una contraseña:

```
$ passwd  postgreSQL
```

Sólo resta crear el espacio de trabajo donde residirán los datos que manejemos, este directorio puede ser cualquiera que nosotros seleccionemos, en este caso lo vamos a crear dentro de la estructura ya

armada en el proceso de instalación. Vamos a instalar entonces en /usr/local/pgsql/data en primera instancia debemos crear el directorio y asignarle como 'dueño' al usuario 'postgreSQL' que ya habíamos creado en el sistema en un paso anterior.

```
$mkdir /usr/local/pgsql/data
$chown postgreSQL /usr/local/pgsql/data
```

y como este mismo usuario, luego inicializamos las bases necesarias para el funcionamiento del sistema..

```
$su - postgreSQL
$/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

la opción "-D" indica el directorio donde se van a generar las bases y archivos de trabajo, por ello si queremos seleccionar otro directorio para almacenar estos archivos, solamente lo indicamos luego de este parámetro.

3.3. Utilizando PostgreSQL desde Linux

3.3.1. Iniciando Postmaster

No le puede suceder nada a una base de datos a menos que esté corriendo el proceso postmaster. Si ha instalado PostgreSQL de una manera correcta, Sólo resta poner en funcionamiento el servidor, un forma podría ser...

```
$/usr/local/pgsql/bin/postmaster
```

Ocasionalmente, postmaster escribe mensajes que le serán de ayuda para resolver problemas. Si desea ver los mensajes de diagnóstico de postmaster, puede iniciarlo con la opción -D y redirigir la salida a un archivo de registro:

```
$/usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
>logfile 2>&1 &
```

esta forma arranca el servidor en 'background' y no mostrará en pantalla la actividad registrada por éste.

Si no desea ver los mensajes, inícielo de la forma

```
$/usr/local/pgsql/bin/postmaster –S
```

y postmaster será "S"ilencioso. Observe que al no haber el signo ampersand ("&") al final del último ejemplo, no se ejecuta como proceso de fondo.

4. Administración y Uso de PostgreSQL

4.1. Agregar y Eliminar Usuarios

createuser permite que usuarios específicos accedan a PostgreSQL.

dropuser elimina usuarios y previene que éstos accedan a PostgreSQL.

Estas órdenes sólo afectan a los usuarios con respecto a PostgreSQL; no tienen efecto en otros privilegios del usuario o en su estado con respecto al sistema operativo subyacente.

4.2. Configurando el Entorno

Esta sección discute la manera de configurar su propio entorno, para que puedas usar aplicaciones. Nosotros asumimos que PostgreSQL ha sido instalado e iniciado exitosamente; consulte la Guía del Administrador y las notas de instalación si desea instalar postgreSQL.

PostgreSQL es una aplicación cliente/servidor. Como usuario, únicamente necesita acceso a la parte cliente (un ejemplo de una aplicación cliente es el monitor interactivo **psql**) Por simplicidad, nosotros asumiremos que PostgreSQL ha sido instalado en el directorio /usr/local/pgsql. Por lo tanto, donde vea el directorio /usr/local/pgsql, deberá sustituirlo por el nombre del directorio donde PostgreSQL esté instalado realmente. Todos los programas de PostgreSQL son instalados en el directorio /usr/local/pgsql/bin. Por lo tanto, deberá añadir este directorio a la de su shell ruta de órdenes. Si usa una variante del C shell de Berkeley, tal como tcsh o csh, deberá añadir

```
$ set path = ( /usr/local/pgsql/bin path )
```

en el archivo .login de su directorio personal.
Si usa una variante del Bourne shell, tal como sh, ksh o bash entonces deberá añadir

```
% PATH=/usr/local/pgsql/bin:$PATH
% export PATH
```

en el archivo .profile de su directorio personal. Desde ahora, asumiremos que se ha añadido el directorio bin de postgreSQL a su path.

4.3. Administración de una Base de Datos

Nota: *Actualmente esta sección es una copia disfrazada del tutorial. A pesar de que el administrador local es responsable por la gestión general de la instalación de postgreSQL, algunas bases de datos instaladas pueden ser administradas por otra persona, llamada el administrador de la base de datos. La responsabilidad de la administración se delega en el momento en que se crea la base de datos. A un usuario se le puede dar privilegio para crear nuevas bases de datos y/o nuevos usuarios. Un usuario que tenga los dos tipos de privilegio puede realizar la mayoría de las labores administrativas en postgreSQL, pero normalmente no tendrá los mismos privilegios de sistema operativo que el administrador local. La Guía del Administrador del PostgreSQL (disponible en el sitio ftp) trata estos tópicos con mas detalle.*

4.3.1. Creación de Bases de Datos

Las bases de datos se crean dentro de postgreSQL con el comando **create base-de-datos**. createdb es un utilitario hecho para suministrar la misma función fuera de postgreSQL, a partir de la línea de comandos. El motor de postgreSQL debe estar corriendo para que cualquiera de los dos métodos funcione, y el usuario que da el comando debe ser el *supe-usuario* de postgreSQL, o haber obtenido privilegio por parte del super-usuario para crear bases de datos. Para crear una base de datos llamada “mibd” a partir de la línea de comandos, escriba

```
% createdb mibd
```

y para obtener el mismo resultado dentro de psql escriba

```
* CREATE DATABASE mibd;
```

Si no tiene el privilegio necesario para crear una base de datos, verá el siguiente mensaje:

```
% createdb mibd
WARN:user "your username" is not allowed to create/destroy
databases createdb: database creation failed on mibd.
```

PostgreSQL le permite crear cualquier número de bases de datos en un servidor y usted será automáticamente el administrador de la base de datos que acaba de crear. Los nombres de las bases de datos deben comenzar por una letra y están limitados a una longitud total de 32 caracteres.

4.3.2. Acceso a una Base de Datos

Una vez haya creado una base de datos, se puede acceder a esta ejecutando los programas monitores de postgreSQL (Por ejemplo psql) que le permite introducir, editar y ejecutar comandos SQL interactivamente.). Puede querer arrancar psql para experimentar los ejemplos en este manual. El psql puede ser activado para la base de datos mibd escribiendo el comando:

```
% psql mibd
```

Será saludado con el siguiente mensaje:

```
Welcome to the POSTGRESQL interactive sql monitor:
Please read the file COPYRIGHT for copyright terms
of POSTGRESQL
type \? for help on slash commands
type \q to quit
type \g or terminate with semicolon to execute query
You are currently connected to the database: mibd
mibd=>
```

Este símbolo indica que el monitor lo escucha y que puede escribir pedidos SQL dentro de un área de trabajo que mantiene el monitor. El programa psql responde a códigos de escape que comiencen con la barra invertida, "\" Por ejemplo, puede obtener ayuda sobre la sintaxis de varios comandos SQL de postgreSQL por medio de:

```
mibd=> \h
```

Una vez termine de introducir sus consultas en el área de trabajo, puede pasar el contenido al servidor de postgreSQL escribiendo:

```
mibd=> \g
```

Esto le dice al servidor que debe procesar su pedido. Si termina su pedido con punto y coma, no necesita el comando "\g". psql procesará automáticamente los pedidos que terminen con punto y coma. Para leer peticiones a partir de un fichero, digamos miFichero, en vez de introducirlas interactivamente, escriba:

```
mibd=> \i miFichero
```

Para salir de psql y regresar a Unix, escriba

```
mibd=> \q
```

y psql finalizará y lo hará regresar a su shell de comandos. (Para ver otros comandos de psql, escriba **\h** mientras ejecuta psql.) En los pedidos SQL se puede usar libremente espacio en blanco (espacio, tabuladores nuevas líneas). Comentarios de una línea se indican con "–". Todo lo que aparezca después de las dos rayas y hasta el fin de la línea será ignorado. Para comentarios de varias líneas o dentro de una línea se usa "/* ... */"

4.3.3. Destrucción de una Base de Datos

Si usted es el administrador de la base de datos mibd, puede destruirla usando el siguiente comando Unix:

```
% dropdb mibd
```

Esto retira físicamente todos los ficheros Unix asociados con la base de datos y no podrán ser recuperados, de manera que debe ser hecho con mucha premeditación.

5. Consultas SQL en PostgreSQL

El lenguaje de consultas de PostgreSQL es una variante del estándar SQL3 Tiene muchas extensiones, tales como tipos de sistema extensibles, herencia, reglas de producción y funciones. Estas son características tomadas del lenguaje de consultas original de postgreSQL (PostQuel). Aquí se proporciona un primer vistazo de cómo usar postgreSQL SQL para realizar operaciones sencillas. La intención es simplemente la de proporcionarle una idea de la versión de SQL de PostgreSQL y no es de ningún modo un completo tutorial acerca de SQL. Se han escrito numerosos libros sobre SQL, incluyendo [MELT93] and [DATE97]. Tenga en cuenta que algunas características del lenguaje son extensiones del estándar ANSI.

5.1. Monitor interactivo

En los ejemplos que siguen, asumimos que ha creado la base de datos midb como se describió en la subsección anterior y que ha arrancado psql. Los ejemplos que aparecen en este texto también se pueden encontrar en /usr/local/pgsql/src/tutorial/. Consulte el fichero README en ese directorio para saber cómo usarlos. Para empezar haga lo siguiente:

```
% cd /usr/local/pgsql/src/tutorial
% psql -s midb

Welcome to the POSTGRESQL interactive sql monitor:
Please read the file COPYRIGHT for copyright terms of POSTGRESQL
type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
   You are currently connected to the database: postgreSQL
   midb=> \i basics.sql
```

El comando \i lee en las consultas desde los ficheros especificados . La opción -s le pone en modo single step, que hace una pausa antes de enviar la consulta al servidor. Las consultas de esta sección están en el fichero basics.sql. psql tiene varios comandos \d para mostrar información de sistema. Consulte estos comando para ver más detalles y teclee \? desde el prompt psql para ver un listado de comandos.

5.2. Conceptos

La noción fundamental en postgreSQL es la de clase, que es una colección de instancias de un objeto. Cada instancia tiene la misma colección de atributos y cada atributo es de un tipo específico. Más aún, cada instancia tiene un *identificador de objeto* (OID) permanente, que es único a lo largo de toda la instalación. Ya que la sintaxis SQL hace referencia a tablas, usaremos los términos *tabla* y *clase* indistintamente. Asimismo ,una *fila*SQL es una *instancia* y las *columnas* SQL son *atributos*. Como ya se dijo anteriormente, las clases se agrupan en bases de datos y una colección de bases de datos gestionada por un único proceso postmaster constituye una instalación o sitio.

5.3. Creación de una nueva clase

Puede crear una nueva clase especificando el nombre de la clase , además de todos los nombres de atributo y sus tipos:

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- low temperature
    temp_hi         int,           -- high temperature
    prcp            real,          -- precipitation
    date            date
);
```

Tenga en cuenta que las palabras clave y los identificadores son sensibles a las mayúsculas y minúsculas. Los identificadores pueden llegar a ser sensibles a mayúsculas o minúsculas si se les pone entre dobles comillas, tal como lo permite SQL92. postgreSQL SQL soporta los tipos habituales de SQL como: int, float, real, smallint, char(N), varchar(N), date, time, and timestamp, así como otros de tipo general y otros con un rico conjunto de tipos geométricos. Tal como veremos más tarde, postgreSQL puede ser configurado con un número arbitrario de tipos de datos definidos por el usuario. Consecuentemente, los nombres de tipo no son sintácticamente palabras clave, excepto donde se requiera para soportar casos especiales en el estándar SQL92 . Yendo más lejos, el comando postgreSQL **CREATE**es idéntico al comando usado para crear una tabla en el sistema relacional de siempre . Sin embargo, veremos que las clases tienen propiedades que son extensiones del modelo relacional.

5.4. Llenando una clase con instancias

La declaración **insert** se usa para llenar una clase con instancias:

```
INSERT INTO weather
        VALUES ('San Francisco', 46, 50, 0.25, '11/27/1994');
```

También puede usar el comando **copy** para cargar grandes cantidades de datos desde ficheros (ASCII) . Generalmente esto suele ser más rápido porque los datos son leídos (o escritos) como una única transacción directamente a o desde la tabla destino. Un ejemplo sería:

```
COPY weather FROM '/home/user/weather.txt'
    USING DELIMITERS '|';
```

donde el path del fichero origen debe ser accesible al servidor backend , no al cliente, ya que el servidor lee el fichero directamente

5.5. Consultar a una clase

La clase weather puede ser consultada con una selección relacional normal y consultas de proyección. La declaración SQL **select** se usa para hacer esto. La declaración se divide en una lista destino (la parte que lista los atributos que han de ser devueltos) y una cualificación (la parte que especifica cualquier restricción). Por ejemplo, para recuperar todas las filas de weather, escriba:

```
SELECT * FROM weather;
```

La salida será:

```
+--------------+---------+---------+------+------------+
|city          | temp_lo | temp_hi | prcp | date       |
+--------------+---------+---------+------+------------+
|San Francisco | 46      | 50      | 0.25 | 11-27-1994 |
+--------------+---------+---------+------+------------+
|San Francisco | 43      | 57      | 0    | 11-29-1994 |
+--------------+---------+---------+------+------------+
```

Puede especificar cualquier expresión en la lista de destino. Por ejemplo, puede hacer:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

Los operadores booleanos (**and**, **or** and **not**)) se pueden usar en la cualificación de cualquier consulta. Por ejemplo,

```
SELECT * FROM weather
    WHERE city = 'San Francisco'
    AND prcp > 0.0;
```

da como resultado:

```
+--------------+---------+---------+------+------------+
|city          | temp_lo | temp_hi | prcp | date       |
+--------------+---------+---------+------+------------+
|San Francisco | 46      | 50      | 0.25 | 11-27-1994 |
+--------------+---------+---------+------+------------+
```

Como apunte final, puede especificar que los resultados de un select puedan ser devueltos de *manera ordenada* o con *instancias duplicadas* borradas.

```
SELECT DISTINCT city
    FROM weather
    ORDER BY city;
```

5.6. Redireccionamiento de consultas SELECT

Cualquier consulta select puede ser redireccionada a una nueva clase:

```
SELECT * INTO TABLE temp FROM weather;
```

Esto forma de manera implícita un comando **create**, creándose una nueva clase temp con el atributo names y types especificados en la lista destino del comando **select into**. Entonces podremos , por supuesto, realizar cualquier operación sobre la clase resultante como lo haríamos sobre cualquier otra clase.

5.7. Joins entre clases

Hasta ahora, nuestras consultas sólo accedían a una clase a la vez. Las consultas pueden acceder a múltiples clases a la vez, o acceder a la misma clase de tal modo que múltiples instancias de la clase sean procesadas al mismo tiempo . Una consulta que acceda a múltiples instancias de las mismas o diferentes clases a la vez se conoce como una consulta join. Como ejemplo,

digamos que queremos encontrar todos los registros que están en el rango de temperaturas de otros registros. En efecto, necesitamos comparar los atributos temp_lo y temp_hi de cada instancia EMP con los atributos temp_lo y temp_hi de todas las demás instancias EMP.

> ***Nota:*** *Esto es sólo un modelo conceptual. El verdadero join puede hacerse de una manera más eficaz, pero esto es invisible para el usuario.*

Podemos hacer esto con la siguiente consulta:

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
    W2.city, W2.temp_lo AS low, W2.temp_hi AS high
    FROM weather W1, weather W2
    WHERE W1.temp_lo < W2.temp_lo
    AND W1.temp_hi > W2.temp_hi;
```

```
+--------------+-----+------+---------------+-----+------+
|city          | low | high | city          | low | high |
+--------------+-----+------+---------------+-----+------+
|San Francisco | 43  | 57   | San Francisco | 46  | 50   |
+--------------+-----+------+---------------+-----+------+
|San Francisco | 37  | 54   | San Francisco | 46  | 50   |
+--------------+-----+------+---------------+-----+------+
```

> ***Nota:*** *: Los matices de este join están en que la cualificación es una expresión verdadera definida por el producto cartesiano de las clases indicadas en la consulta. Para estas instancias en el producto cartesiano cuya cualificación sea verdadera, postgreSQL calcula y devuelve los valores especificados en la lista de destino. postgreSQL SQL no da ningún significado a los valores duplicados en este tipo de expresiones. Esto significa que postgreSQL en ocasiones recalcula la misma lista de destino varias veces. Esto ocurre frecuentemente cuando las expresiones booleanas se conectan con un "or". Para eliminar estos duplicados, debe usar la declaración* ***select distinct*** *.*

En este caso, tanto W1 como W2 son sustituidos por una instancia de la clase weather y se extienden por todas las instancias de la clase. (En la terminología de la mayoría de los sistemas de bases de datos W1 y W2 se conocen como *range variables (variables de rango)*.) Una consulta puede contener un número arbitrario de nombres de clases y sustituciones.

5.8. Actualizaciones

Puede actualizar instancias existentes usando el comando update. Suponga que descubre que la lectura de las temperaturas el 28 de Noviembre fue 2 grados superior a la temperatura real. Puede actualizar los datos de esta manera:

```
UPDATE weather
    SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
    WHERE date > '11/28/1994';
```

5.9. Borrados

Los borrados se hacen usando el comando **delete**:

```
DELETE FROM weather WHERE city = 'Hayward';
```

Todos los registros de weather pertenecientes a Hayward son borrados. Debería ser precavido con las consultas de la forma

```
DELETE FROM classname;
```

Sin una cualificación, **delete** simplemente borrará todas las instancias de la clase dada, dejándola vacía. El sistema no pedirá confirmación antes de hacer esto.

5.10. Uso de funciones de conjunto

Como otros lenguajes de consulta, PostgreSQL soporta funciones de conjunto. Una función de conjunto calcula un único resultado a partir de múltiples filas de entrada. Por ejemplo, existen funciones globales para calcular count(contar) ,sum (sumar), avg (media), max (máximo) and min (mínimo) sobre un conjunto de instancias.

Es importante comprender la relación entre las funciones de conjunto y las cláusulas SQL **where** y **having** . . La diferencia fundamental entre **where** y **having** es que: **where** selecciona las columnas de entrada antes de los grupos y entonces se computan las funciones de conjunto (de este modo controla qué filas van a la función de conjunto), mientras que **having** selecciona grupos de filas después de los grupos y entonces se computan las funciones de conjunto. De este modo la cláusula **where** puede no contener funciones de conjunto puesto que no tiene sentido intentar usar una función de conjunto para determinar qué fila será la entrada de la función. Por otra parte, las cláusulas **having** siempre contienen funciones de conjunto. (Estrictamente hablando, usted puede escribir una cláusula **having** que no use funciones de grupo, pero no merece la pena. La misma condición podría ser usada de un modo más eficaz con **where** .)

Como ejemplo podemos buscar la mínima temperatura en cualquier parte con

```
SELECT max(temp_lo) FROM weather;
```

Si queremos saber qué ciudad o ciudades donde se dieron estas temperaturas, podemos probar

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);
```

pero esto no funcionará debido a que la función max() no puede ser usada en **where**. Sin embargo y como es frecuente, la consulta puede ser replanteada para llevar a cabo lo que se buscaba. En este caso usando una *subseleccion*:

```
SELECT city
FROM weather
WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
```

Esto es correcto, ya que la subselección es una operación independiente que calcula su propia función de grupo separadamente de lo que ocurre en el select exterior.

Las funciones de grupo son también muy útiles combinándolas con cláusulas *group by* . Por ejemplo, podemos obtener la temperatura mínima tomada en cada ciudad con :

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city;
```

que nos devuelve una fila por ciudad. Podemos filtrar estas filas agrupadas usando **having**:

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city
    HAVING min(temp_lo) < 0;
```

que nos da los mismos resultados, pero de ciudades con temperaturas bajo cero. Finalmente, si sólo nos interesan las ciudades cuyos nombres empiecen por 'P', deberíamos hacer :

```
SELECT city, max(temp_lo)
    FROM weather
    WHERE city like 'P%'
    GROUP BY city
    HAVING min(temp_lo) < 0;
```

Tenga en cuenta que podemos aplicar la restricción del nombre de ciudad en **where**, ya que no necesita funciones de conjunto. Esto es más eficaz que añadir la restricción a **having**,debido a que evitamos hacer los cálculos de grupo para todas las filas que no pasan el chequeo de **where** .

6. Características Avanzadas de SQL en PostgreSQL

Habiendo cubierto los aspectos básicos de Postgre SQLpara acceder a los datos, discutiremos ahora aquellas características de postgreSQL que los distinguen de los administradores de datos convencionales. Estas características incluyen herencia, time travel (viaje en el tiempo) y valores no-atómicos de datos (atributos basados en vectores y conjuntos).

6.1. Herencia

Creemos dos clases. La clase capitals contiene las capitales de los estados, las cuales son también ciudades. Naturalmente, la clase capitals debería heredar de cities.

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

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

En este caso, una instancia de capitals *hereda* todos los atributos (name, population y altitude) de su padre, cities. El tipo del atributo name (nombre) es text, un tipo nativo de postgreSQL para cadenas ASCII de longitud variable. El tipo del atributo population (población) es float, un tipo de datos, también nativo de postgreSQL , para números de punto flotante de doble precisión. Las capitales de los estados tiene un atributo extra, state, que muestra a qué estado pertenecen. En postgreSQL, una clase puede heredar de ninguna o varias otras clases, y una consulta puede hacer referencia tanto a todas las instancias de una clase como a todas las instancias de una clase y sus descendientes.

Nota: *La jerarquía de la herencia es un gráfico acíclico dirigido.*

Por ejemplo, la siguiente consulta encuentra todas aquellas ciudades que están situadas a un altura de 500 o más pies:

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

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

Por otro lado, para encontrar los nombres de todas las ciudades, incluídas las capitales estatales, que estén situadas a una altitud de 500 o más pies, la consulta es:

```
SELECT c.name, c.altitude
    FROM cities* c
    WHERE c.altitude > 500;
```

Retorna:

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

Aquí el "*" después de cities indica que la consulta debe realizarse sobre cities y todas las clases que estén por debajo de ella en la jerarquía de la herencia. Muchos de los comandos que ya hemos discutido (**select**, **and>up**and> and **delete**) brindan soporte a esta notación de "*" al igual que otros como **alter**.

6.2. Valores No-Atómicos

Uno de los principios del modelo relacional es que los atributos de una relación son atómicos postgreSQL no posee esta restricción; los atributos pueden contener sub-valores que pueden ser accesados desde el lenguaje de consulta. Por ejemplo, usted puede crear atributos que sean vectores de alguno de los tipos base.

Vectores

postgreSQL permite que los atributos de una instancia sean definidos como vectores multidimensionales de longitud fija o variable. Puede crear vectores de cualquiera de los tipos base o de tipos definidos por el usuario. Para ilustrar su uso, creemos primero una clase con vectores de tipos base.

```
CREATE TABLE SAL_EMP (
    name            text,
    pay_by_quarter  int4[],
    schedule        text[][]
);
```

La consulta de arriba creará una clase llamada SAL_EMP con una cadena del tipo *text* (name),un vector unidimensional del tipo *int4* (pay_by_quarter), el cual representa el salario trimestral del empleado y un vector bidimensional del tipo *text* (schedule), que representa la agenda semanal del empleado. Ahora realizamos algunos *INSERTS*s; note que cuando agregamos valores a un vector, encerramos los valores entre llaves y los separamos mediante comas. Si usted conoce *C*, esto no es distinto a la sintaxis para inicializar estructuras.

```
INSERT INTO SAL_EMP
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {}}');

INSERT INTO SAL_EMP
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"talk", "consult"}, {"meeting"}}');
```

postgreSQL utiliza de forma predeterminada la convención de vectores "basados en uno" -- es decir, un vector de n elementos comienza con vector[1] y termina con vector[n]. Ahora podemos ejecutar algunas consultas sobre SAL_EMP. Primero mostramos como acceder a un solo elemento del vector por vez. Esta consulta devuelve los nombres de los empleados cuyos pagos han cambiado en el segundo trimestre:

```
SELECT name
    FROM SAL_EMP
    WHERE SAL_EMP.pay_by_quarter[1] <>
    SAL_EMP.pay_by_quarter[2];
```

```
+------+
|name  |
+------+
|Carol |
+------+
```

La siguiente consulta recupera el pago del tercer trimestre de todos los empleados:

```
SELECT SAL_EMP.pay_by_quarter[3] FROM SAL_EMP;
```

```
+---------------+
|pay_by_quarter |
+---------------+
|10000          |
+---------------+
|25000          |
+---------------+
```

También podemos acceder a cualquier porción de un vector, o subvectores. Esta consulta recupera el primer item de la agenda de Bill para los primeros dos días de la semana.

```
SELECT SAL_EMP.schedule[1:2][1:1]
    FROM SAL_EMP
    WHERE SAL_EMP.name = 'Bill';

+-------------------+
|schedule           |
+-------------------+
|{{"meeting"},{""}} |
+-------------------+
```

6.3. Más características avanzadas

PostgreSQL posee muchas características que no se han visto en este texto, el cual ha sido orientado principalmente hacia usuarios nuevos en SQL. Las mencionadas características se discuten tanto en la Guía del Usuario como en la del Programador (disponibles en el CD o en el sitio de postgresql) .

7. Interfaz ODBC

Nota: *[1]ODBC (Open Database Connectivity / Conectividad Abierta para Bases de Datos) es un API abstracto que permite escribir aplicaciones que pueden ínteroperar con varios servidores RDBMS. ODBC facilita un interfaz de producto neutral entre aplicaciones de usuario final y servidores de bases de datos, permitiendo ya sea a un usuario o desarrollador escribir aplicaciones transportables entre servidores de diferentes fabricantes.*

7.1. Trasfondo

El API ODBC se conecta en la capa inferior de la aplicación con una fuente de datos compatible con ODBC. Ésta (la fuente de datos) podría ser desde un fichero de texto a una RDBMS Oracle o postgreSQL. El acceso en la capa inferior de aplicación se produce gracias a drivers ODBC, o drivers específicos del fabricante que permiten el acceso a los datos. psqlODBC es un driver, junto con otros que están disponibles, como los drivers ODBC Openlink. Cuando se escribe una aplicación ODBC usted, *debería* ser capaz de conectar con cualquier base de datos, independientemente del fabricante, siempre y cuando el esquema de la base de datos sea el mismo. Por ejemplo. Usted podría tener servidoresMS SQL Server y postgreSQL que contuvieran exactamente los mismos datos. Usando ODBC, su aplicación Windows podría hacer exactamente las mismas llamadas y la fuente de datos a nivel interno sería la misma (para la aplicación cliente en Windows). Insight Distributors (http://www.insightdist.com/) dan soporte continuo y actual a la distribución psqlODBC. Suministran un FAQ (http://www.insightdist.com/psqlodbc/), sobre el desarrollo actual del código base, y participan activamente en la lista de correo de interfaces (mailto:interfaces@postgresql.org).

[1] Información de fondo realizada originalmente por Tim Goeke (mailto:tgoeke@xpressway.com)

7.2. Aplicaciones Windows

En la actualidad, las diferencias en los drivers y el nivel de apoyo a ODBC reducen el potencial de ODBC:

- ✓ Access, Delphi, y Visual Basic soportan directamente ODBC.
- ✓ Bajo C++, y en Visual C++, puede usar el API ODBC de C++.
- ✓ En Visual C++, puede usar la clase CRecordSet, la cual encapsula el API ODBC dentro de una clase MFC 4.2. Es el camino más fácil si está desarrollando en C++ para Windows bajo Windows NT.

7.3. Instalando y Configurando el Driver ODBC para PostgreSQL sobre Windows

Para poder usar (comunicarse con) postgresql via odbc, se debe instalar el driver que se encuentra en el archivo comprimido (disponible en el CD):

```
postgreSQL ODBC.zip
```

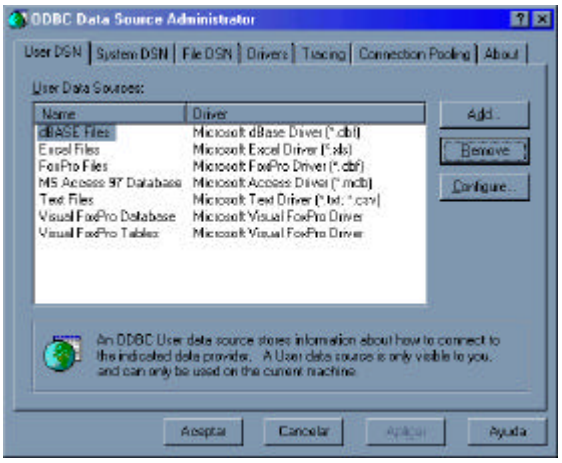
Al descomprimir, se puede observar que existen 2 archivos, *psqlkoi8.dll* y *postdrv.exe*, ejecutar el ultimo *(postdrv.exe)*, seguir las instrucciones de instalacion, Seleccionar la opcion: ***install driver manager (with version checking)*** cuando se le pregunte (esta opcion permite que se puedan hacer mas cosas en lo que respecta a la versión de postgresql que se vaya a usar o se este usando.

7.4. Configuración

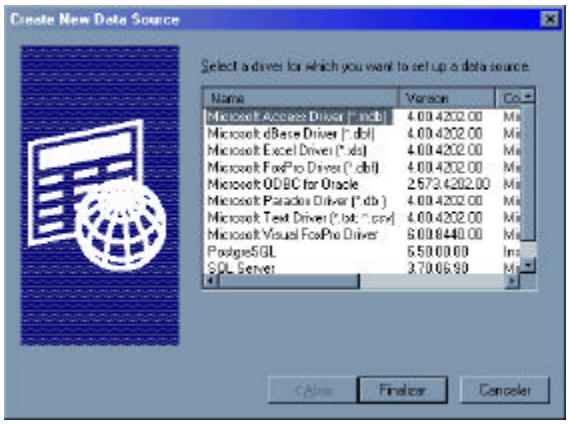
Una vez instalado el driver, lo unico que resta es configurarlo para poder tener disponible la Base de Datos via ODBC. Ya sea para comunicarlo con aplicaciones o para observar los datos mediante un explorador de base de datos (por ejemplo el que viene con Delphi).

Activar el Panel de Control (inicio → Configuracion → Panel de Control).

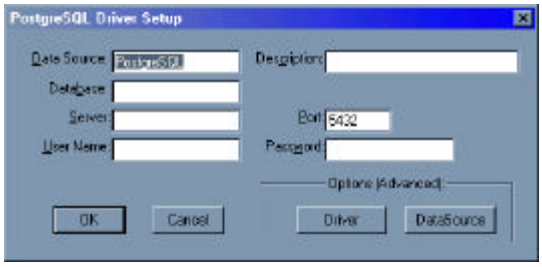
Seleccionar el icono ODBC Data Sources (32bit).



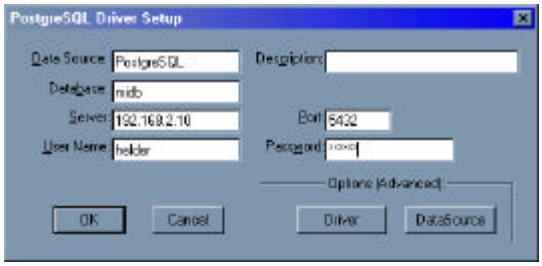
Se pueden observar los controladores de otros manejadores de bases de datos disponibles en el sistema. Seleccionar el botón añadir(Add)



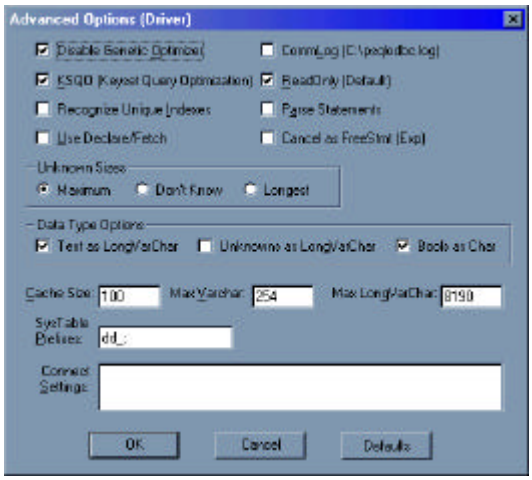
Seleccionar el Driver PostgreSQL, y luego presionar el botón finalizar.



Llenar los campos Databas (nombre de la base de datos), Server (nombre del servidor donde se encuentra instalado postgresql o su direccion IP), user name (nombre del usuario de la base de datos, ademas debe ser un usuario valido en linux), password (la contraseña del usuario), el numero de puerto para postgresql es por defecto 5432, consulte al administrador si es que se ha configurado con otro numero.

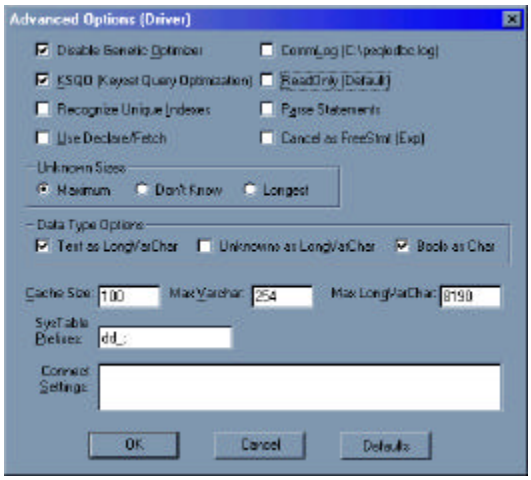


Por defecto el acceso a la base de datos es de solo lectura, si se quiere un



acceso total (lectura / escritura), se debe indicar. Para esto hacer clic en Driver (Options Advanced).

Asegurarse que ReadOnly(Default) este deshabilitado.



Ir aceptando las opciones indicadas. (OK, OK, Aceptar).

8. Interfaz JDBC

8.1. Construyendo la interfaz JDBC

8.2. Compilando el Driver

Los Fuentes del Driver se encuentran en el Directorio src/interfaces/jdbc del árbol de fuentes (src). Para compilarlo, simplemente debe cambiar su directorio a ese directorio y, escribir en el prompt:

```
% make
```

al finalizar, se puede encontrar el archivo postgresql.jar en el directorio actual. Este es el Driver JDBC.

8.3. Instalando el Driver.

Para usar el Driver, el archivo postgresql.jar necesita ser incluido en CLASSPATH.
Ejemplo:
Tenemos una aplicación que usa el Driver JDBC para acceder a una base de datos grande que contiene objetos grandes. Tenemos la aplicación y el Driver jdbc instalados en el directorio /usr/local/lib , y java jdk instalado en /usr/local/jdk1.1.6.
Para correr la aplicación se deberia usar:

```
ExportCLASSPATH=/usr/local/lib/finder.jar:/usr/local/lib/postgr
esql.jar:.java uk.org.retep.finder.Main
```

8.4. Preparando la base de Datos para JDBC.

Dado que Java puede trabajar solamente con conecciones TCP/IP el Postmaster postgresql deberia estar funcionando con la bandera –i.
El archivo pg_hba.conf también debe ser configurado. Esta ubicado en el Directorio PGDATA. En una instalacion por defecto, este archivo permite acceso solamenta por sockets con dominio Unix. Para conectar el Driver JDBC al mismo host local, se necesita añadir algo como:

```
host all 127.0.0.1 255.255.255.255 password
```

indica acceso a todas (all) las bases de datos posibles de la maquina local con JDBC. El Driver JDBC soporta metodos de autenticación como trust, ident, password y crypt.

8.5. Usando JDBC

No es el fin proporcionar un guia completa de programacion JDBC, pero se pretende proporcionar ayuda basica para que usted pueda empezar. Para mayor informacion, refierase a la documentacion Estandar JDBC API.

8.6. Importando JDBC

Algunos programas que usan JDBC necesitan que se importe el paquete java.sql, usando:

```
import java.sql.*;
```

Nota. *No importe el paquete postgresql directamente. Si hace esto, su codigo puede no compilar (conflictos con javac).*

8.7. Cargando el Driver.

Antes de conectarse a una Base de Datos, se necesita cargar el driver. Existen dos maneras disponibles, depende de la naturaleza de su programa para definir cual seria el optimo.
En la primera forma, el codigo implicitamente carga el driver usando el metodo Class.forName(). Para Postgresql deberia usarse:

```
Class.forName("postgresql.Driver");
```

Esto debería cargar el Driver, y mientras esto sucede, el Driver debería registrarse automáticamente, por si mismo con JDBC.

> ***Nota:*** *La función forName()puede ocasionar una excepción (ClassNotFoundException), así que es necesario capturarla si no esta disponible el drive.*

Este es el método mas usado, pero restringe al código a usar precisamente postgresql. Si pretende que su programa acceda a otras bases de datos en el futuro, sin hacer extensiones. Entonces podría usar la segunda forma.
En la segunda forma, se pasa el Driver como un parámetro a la maquina virtual de Java (JVM de aquí en adelante) cuando esta se inicia, usando el argumento –D. Ejemplo:

```
% java -Djdbc.drivers=postgresql.Driver example.ImageViewer
```

en el ejemplo, JVM es preparada para cargar el Driver como parte de su inicialización. Solo si se carga satisfactoriamente, inicia la maquina virtual. Este método es mejor desde el punto de vista de que permite que el programa pueda ser usado con otras bases de datos, sin tener que recompilar el código. El único cambio que se debe hacer es el de la URL que se vera mas adelante. Por ultimo, cuando el programa intenta establecer

una conexión, y resulta que no existe un driver disponible, ocurre una excepcion (SQLException), probablemente ocasionada por que el driver no se encuentra en el classpath o por que el valor en el parametro no es correcto.

8.8. Conectando a la Base de Datos.

Con JDBC, una base de datos es representada por una URL (Localizador Uniforme de Recursos) . Con Postgresql esto toma una de las siguientes formas:

- `jdbc:postgresql:database`
- `jdbc:postgresql://>hos>/database`
- `jdbc:postgresql://>hos>">poe>/database`

Donde:

host
es el nombre de host del servidor. Por defecto "localhost".

port
El numero de puerto en el que se encuentra escuchando el Servidor. El puerto por defecto para Postgresql es el numero 5432.

database
Nombre de la Base de Datos.

Para conectarse, se necesita tomar una instancia de concecion de JDBC. Para hacer esto se debe usar la funcion DriverManager.getConnection() :

```
Connection db = DriverManager.getConnection(url,user,pwd);
```

8.9. Usando la Interfaz ResulSet

Lo siguiente deberia ser considerado cuando se usa la interfaz ResulSet:

- Antes de leer cualquier valor se debe llamar a la funcion next(). Por que retorna True si es que existe un resultado, pero mas importante, prepara a la tupla para procesamiento.
- Bajo JDBC, se debe acceder a un campo solamente una vez, Under the JDBC spec, you should access a field only once.
- Se debe cerrar un ResultSet llamando a la funcion close() solo si realmente se ha terminado.
- Si requiere hacer otra consulta con la sentencia usada para crear un ResulSet, la instancia actual es cerrada.

Ejemplo:

```
Statement st = db.createStatement();
ResultSet rs = st.executeQuery("select * from mitabla");
while(rs.next()) {
System.out.print("retornando columna 1");
System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

8.10. Haciendo Actualizaciones (Updates).

Para hacer un update (u otra sentencian SQL que no retorna valores como resultado). Simplemente se usa la función executeUpdate() como en el siguiente ejemplo:

```
st.executeUpdate("create table basic (a int2, b int2)");
```

8.11. Cerrando la Conexión.

Para cerrar la conexión a la Base de Datos, simplemente se debe llamar a la funcion close() de la conexión:

```
db.close();
```

9. Triggers

En español se llaman o están traducidos por desencadenadores o Disparadores. Son lo mismo que los Stored Procedures pero éstos se ejecutan **desatendidamente** y automáticamente cuando un usuario realiza una acción con la tabla de una base de datos que lleve asociado este trigger.

postgreSQL tiene algunas interfaces cliente como Perl, Tcl, Python y C, así como dos *Lenguajes Procedurales* (PL). También es posible llamar a funciones C como acciones trigger. Notar que los eventos trigger a nivel STATEMENT no están soportados en la versión actual. Actualmente es posible especificar BEFORE o AFTER en los INSERT, DELETE o UPDATE de un registro como un evento trigger.

9.1. Creación de Triggers

Si un evento trigger ocurre, el administrador de triggers (llamado Ejecutor) inicializa la estructura global TriggerData *CurrentTriggerData (descrita más abajo) y llama a la función trigger para procesar el evento. La función trigger debe ser creada antes que el trigger, y debe hacerse como una función sin argumentos, y códigos de retorno opacos.

La sintaxis para la creación de triggers es la siguiente:

```
CREATE TRIGGER <trigger name> <BEFORE|AFTER>
<INSERT|DELETE|UPDATE>
ON <relation name> FOR EACH <ROW|STATEMENT>
EXECUTE PROCEDURE <procedure name> (<function args>);
```

El nombre del trigger se usará si se desea eliminar el trigger. Se usa como argumento del comando DROP TRIGGER. La palabra siguiente determina si la función debe ser llamada antes (BEFORE) o después (AFTER) del evento.

El siguiente elemento del comando determina en que evento/s será llamada la función. Es posible especificar múltiples eventos utilizado el operador OR. El nombre de la relación (relation name) determinará la tabla afectada por el evento.

La instrucción FOR EACH determina si el trigger se ejecutará para cada fila afectada o bien antes (o después) de que la secuencia se haya completado. El nombre del procedimiento (procedure name) es la función C llamada. Los argumentos son pasados a la función en la estructura CurrentTriggerData. El propósito de pasar los argumentos a la función es permitir a triggers diferentes con requisitos similares llamar a la misma función. Además, la función puede ser utilizada para disparar distintas relaciones (estas funciones son llamadas "general trigger funcions").

Como ejemplo de utilización de lo descrito, se puede hacer una función general que toma como argumentos dos nombres de campo e inserta el nombre del usuario y la fecha (timestamp) actuales en ellos. Esto permite, por ejemplo, utilizar los triggers en los eventos INSERT para realizar un seguimiento automático de la creación de registros en una tabla de transacciones. Se podría utilizar también para registrar actualizaciones si es utilizado en un evento UPDATE.

Las funciones trigger retornan un área de tuplas (HeapTuple) al ejecutor. Esto es ignorado para trigger lanzados tras (AFTER) una operación INSERT, DELETE o UPDATE, pero permite lo siguiente a los triggers BEFORE: - retornar NULL e ignorar la operación para la tupla actual (y de este modo la tupla no será insertada/actualizada/borrada); - devolver un puntero a otra tupla (solo en eventos INSERT y UPDATE) que serán insertados (como la nueva versión de la tupla actualizada en caso de UPDATE) en lugar de la tupla original. Notar que no hay inicialización por parte del CREATE TRIGGER handler. Esto será cambiado en el futuro. Además, si más de un trigger es definido para el mismo evento en la misma relación, el orden de ejecución de los triggers es impredecible. Esto puede ser cambiado en el futuro.

Si una función trigger ejecuta consultas SQL (utilizando SPI) entonces estas funciones pueden disparar nuevos triggers. Esto es conocido como triggers en cascada. No hay ninguna limitación explicita en cuanto al número de niveles de cascada. Si un trigger es lanzado por un INSERT e inserta una nueva tupla en la misma relación, el trigger será llamado de nuevo (por el nuevo INSERT). Actualmente, no se proporciona ningún mecanismo de sincronización (etc) para estos casos pero esto puede cambiar. Por el momento, existe una función llamada funny_dup17() en los tests de regresión que utiliza algunas técnicas para parar la recursividad (cascada) en si misma...

9.2. Interacción con el Trigger Manager

Como se ha mencionado, cuando una función es llamada por el administrador de triggers (trigger manager), la estructura TriggerData *CurrentTriggerData no es NULL y se inicializa. Por lo cual es mejor verificar que CurrentTriggerData no sea NULL al principio y asignar el valor NULL justo después de obtener la información para evitar llamadas a la función trigger que no procedan del administrador de triggers. La estructura TriggerData se define en src/include/commands/trigger.h:

```
typedef struct TriggerData
{
TriggerEvent tg_event;
Relation tg_relation;
HeapTuple tg_trigtuple;
HeapTuple tg_newtuple;
Trigger *tg_trigger;
} TriggerData;
```

tg_event

describe los eventos para los que la función es llamada. Puede utilizar las siguientes macros para examinar tg_event:

TRIGGER_FIRED_BEFORE(event) devuelve TRUE si el trigger se disparó antes;
TRIGGER_FIRED_AFTER(event) devuelve TRUE si se disparó después;
TRIGGER_FIRED_FOR_ROW(event) devuelve TRUE si el trigger se disparó para un evento a nivel de fila;
TRIGGER_FIRED_FOR_STATEMENT(event) devuelve TRUE si el trigger se disparó para un evento a nivel de sentencia.
TRIGGER_FIRED_BY_INSERT(event) devuelve TRUE si fue disparado por un INSERT;
TRIGGER_FIRED_BY_DELETE(event) devuelve TRUE si fue disparado por un DELETE;
TRIGGER_FIRED_BY_UPDATE(event) devuelve TRUE si fue disparado por un UPDATE.

tg_relation

es un puntero a una estructura que describe la relación disparadora. Mirar en src/include/utils/rel.h para ver detalles sobre esta estructura. Lo más interesante es tg_relation->rd_att (descriptor de las tuplas de la relación) y tg_relation->rd_rel->relname (nombre de la relación. No es un char*, sino NameData. Utilizar SPI_getrelname(tg_relation) para obtener char* si se necesita una copia del nombre).

tg_trigtuple

es un puntero a la tupla por la que es disparado el trigger, esto es, la tupla que se está insertando (en un INSERT), borrando (DELETE) o actualizando (UPDATE). En caso de un INSERT/DELETE esto es lo que se debe devolver al Ejecutor si no se desea reemplazar la tupla con otra (INSERT) o ignorar la operación.

tg_newtuple

es un puntero a la nueva tupla en caso de UPDATE y NULL si es para un INSERT o un DELETE. Esto es lo que debe devolverse al Ejecutor en el caso de un UPDATE si no se desea reemplazar la tupla por otra o ignorar la operación.

tg_trigger

es un puntero a la estructura Trigger definida en src/include/utils/rel.h:

```
typedef struct Trigger
{
Oid tgoid;
char *tgname;
FmgrInfo tgfunc;
int16 tgtype;
bool tgenabled;
bool tgisconstraint;
bool tgdeferrable;
bool tginitdeferred;
int16 tgnargs;
int16 tgattr[FUNC_MAX_ARGS];
char **tgargs;
} Trigger;
```

tgname es el nombre del trigger, tgnargs es el número de argumentos en tgargs, tgargs es un array de punteros a los argumentos especificados en el CREATE TRIGGER. Otros miembros son exclusivamente para uso interno.

9.3. Visibilidad de Cambios en Datos

Regla de visibilidad de cambios en postgreSQL: durante la ejecución de una consulta, los cambios realizados por ella misma (vía funciones SQL O SPI, o mediante triggers) le son invisibles. Por ejemplo, en la consulta INSERT INTO a SELECT * FROM a las tuplas insertadas son invisibles para el propio SELECT. En efecto, esto duplica la tabla dentro de sí misma (sujeto a las reglas de índice único, por supuesto) sin recursividad. Pero hay que recordar esto sobre visibilidad en la documentación de SPI:

Los cambios hechos por la consulta Q son visibles por las consultas que empiezan tras la consulta Q, no importa si son iniciados desde Q (durante su ejecución) o una vez ha acabado.

Esto es válido también para los triggers, así mientras se inserta una tupla (tg_trigtuple) no es visible a las consultas en un trigger BEFORE, mientras que esta tupla (recién insertada) es visible a las consultas de un trigger AFTER, y para las consultas en triggers BEFORE/AFTER lanzados con posterioridad!