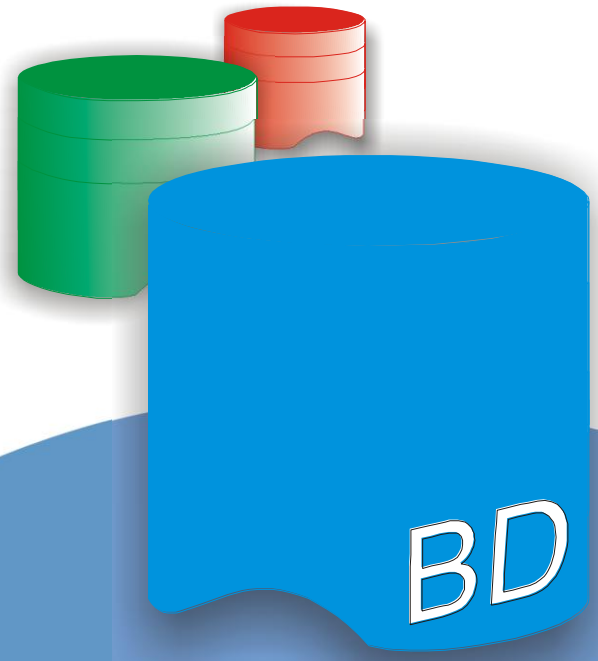
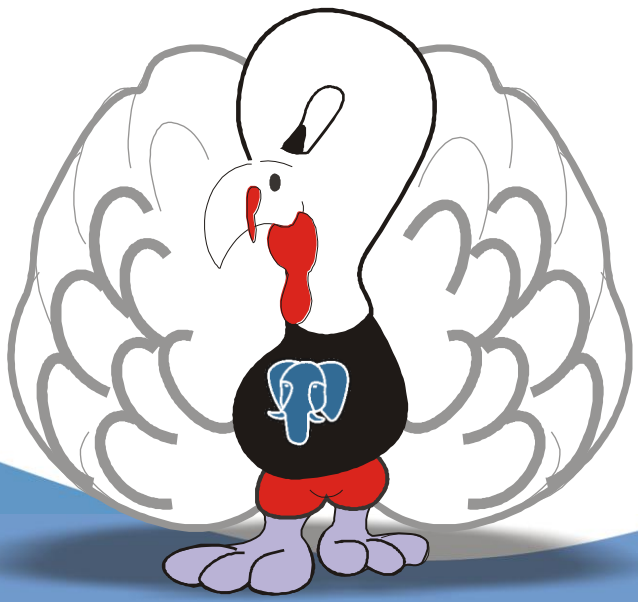
PostgreSQL 8.0.3

Manual Básico





Impreso en Colombia - Universidad Autónoma de Colombia

PRESENTACIÓN

Este documento es el Manual Básico del sistema de Bases de Datos PostgreSQL 8.0.3 que hace parte de la serie de Manuales FENEED[1]. Incluye temas originariamente desarrollados en la Universidad de California en Berkeley y temas de interés para el usuario que desea aprender PostgreSQL.
PostgreSQL está basada en Postgres release 4.2.[2] El proyecto Postgres, liderado por el Profesor Michael Stonebraker, fue responsabilizado por diversos organismos oficiales u oficiosos de los EE.UU. : la Agencia de Proyectos de Investigación Avanzada de la Defensa de los EE.UU. (DARPA), la Oficina de Investigación de la Armada (ARO), la Fundación Nacional para la Ciencia (NSF), y ESL, Inc.
Este manual hace parte del proyecto titulado –DISEÑO DEL MODELO DE NEGOCIO PARA LA ASESORÍA, CONSULTORÍA Y CAPACITACIÓN EN EL GESTOR DE BASE DE DATOS LIBRE PostgreSQLll elaborado por egresados de la Universidad Autónoma de Colombia.

TERMINOLOGÍA

En la documentación siguiente, *sitio* (o *site*) se puede interpretar como la máquina en la que está instalada Postgres. Dado que es posible instalar más de un conjunto de Bases de Datos Postgres en una misma máquina, este término denota, de forma más precisa, cualquier conjunto concreto de programas binarios y Bases de Datos de Postgres instalados.

* El *superusuario* de Postgres es el usuario llamado *postgres* que es dueño de los ficheros de la Bases de Datos y binarios de Postgres. 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 Postgres se le permite ejecutar programas de soporte que generalmente no están disponibles para todos los usuarios. Tenga en cuenta que el superusuario de Postgres *no* es el mismo que el superusuario de Linux (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 Postgres con mecanismos para hacer cumplir una política de seguridad para un site. El DBA puede añadir nuevos usuarios por el método descrito más adelante y mantener un conjunto de Bases de Datos plantilla para usar con CREATEDB.

* El postmaster es el proceso que actúa como una puerta de control (clearing-house), para las peticiones del sistema Postgres. Las aplicaciones frontend se conectan al postmaster,

[1] Nombre del equipo desarrollador de este proyecto; incluye las dos primeras letras de los nombres de los integrantes Fernando, Nelson, Edicson, FENEED INGENIEROS.
[2] http://db.cs.berkeley.edu//postgres.html

que mantiene registros de los errores del sistema y de la comunicación entre los procesos backend. El postmaster puede aceptar varios argumentos desde la línea de órdenes para poner a punto su comportamiento. Sin embargo, el proporcionar argumentos es necesario sólo si se intenta trabajar con varios sitios o con uno que no se ejecuta a la manera por defecto.

* El backend de Postgres (el programa ejecutable Postgres *real*) lo puede ejecutar el superusuario directamente desde el intérprete de órdenes de usuario de Postgres (con el nombre de la Base de Datos como un argumento) Sin embargo, hacer esto elimina el buffer pool compartido y bloquea la tabla asociada con un postmaster / sitio, por ello esto no está recomendado en un sitio multiusuario.

NOTACIÓN DE AYUDA

"..." o /usr/local/pgsql/ delante de un nombre de fichero se usa para representar el camino (path) al directorio home del superusuario de Postgres.
En la sinopsis, los corchetes ("[" y "]") indican una expresión o palabra clave opcional. Cualquier cosa entre llaves ("{" y "}") y que contenga barras verticales ("|") indica que debe elegir una de las opciones que separan las barras verticales.
En los ejemplos, los paréntesis ("(" y ")") se usan para agrupar expresiones booleanas. "|" Es el operador booleano OR.

Los ejemplos mostrarán órdenes ejecutadas desde varias cuentas y programas. Las órdenes ejecutadas desde la cuenta del root y el superusuario de Postgres pueden variar dependiendo de la distribución utilizada. Por Ejemplo, en la distribución Suse el root sé vera algo como: Linux: ~ # y el superusuario postgres: postgres@linux: ~ > Las órdenes de PostgreSQL en Suse estarán precedidas por el nombre de la Base de Datos y los símbolos = # por Ejemplo: BDCEMENTERIO =# o no estarán precedidas por ningún prompt, dependiendo del contexto.

AGRADECIMIENTO

Esta serie de Manuales no se habría podido realizar sin la generosa colaboración de Claudia Gonzáles Riaño quien estuvo a cargo de la revisión ortográfica y de contexto; desde el principio nos ha apoyado, aportado y criticado para desarrollar y mejorar la presente serie.

... y desde ya a todos aquellos que desde la Universidad Autónoma de Colombia y fuera de ella alientan a continuar.

GRACIAS.

OBJETIVO

Hemos escrito este Manual Básico para usuarios de PostgreSQL 8.0.3. Tenga en cuenta que comenzar por el principio de un libro es por lo general una buena idea para no pasar por alto los notas y consejos que contiene cada capitulo. Este manual está orientado a aquellas personas que tengan conocimientos esenciales de Base de Datos y a cualquier persona que quiera conocer una de las grandes herramientas que tiene el Software Libre.

Para facilitar el aprendizaje de este Manual se trabajará con diferentes ejemplos sencillos que ayudaran a dar soporte al desarrollo de una Base de Datos final llamada BDCEMENTERIO. Esta Base de Datos incluye la creación de un Diagrama Entidad – Relación, Construcción de tablas, Consultas, Utilización de PgAccess entre otras operaciones típicas para el desarrollo de Base de Datos.

Seguramente para la fecha del lanzamiento de esta serie de manuales ya exista una nueva versión Beta de PostgreSQL, sin embargo la versión 8.0.3 es una versión Estándar la cual no tiene problemas.

Creemos que este Manual cubre al detalle PostgreSQL en el nivel Básico mejor que cualquier otro que se encuentre actualmente en el mercado. Esperamos que lo encuentre de gran ayuda al trabajar con PostgreSQL y que disfrute su estilo innovador.

ORGANIZACIÓN DEL MANUAL

Este Manual contiene 34 secciones agrupadas en 6 Capítulos, incluyendo subsecciones en cada una. Él número de capítulo esta indicado con números romanos.
Los capítulos de este Manual son: En el Cáp. I. Se hace una introducción a PostgreSQL. Desde la sección 1 hasta la 5 se discuten temas de introducción como son la definición, historia y proyectos de PostgreSQL. En el Cáp. II. Se explica como se inicia el postmaster, como se conecta a PostgreSQL y adicionalmente se darán las instrucciones para crear y acceder a una Base de Datos. El siguiente Capítulo da una introducción a lo que será nuestro problema a desarrollar titulado BDCEMENTERIO, donde se define un problema y distintas observaciones a tener en cuenta, además, se encuentran temas como son los Join entre tablas, Actualizaciones y Eliminaciones de objetos. El Cáp. IV. Contiene las características avanzadas de PostgreSQL como son la Herencia, *Transacciones*, entre otras, además, de la creación de tablas y consultas para BDCEMENTERIO con todas sus características. En el Cáp. V. Se trata el tema de definición de datos, contiene todos los tipos de Constraints, Modificación de los elementos de una tabla, Shemas y las posibles modificaciones que se le pueden hacer a los datos. Por ultimo esta el Cáp. VI. Este capitulo esta dedicado a la interfaz grafica de PostgreSQL llamada PgAccess, con este programa se pueden realizar tareas como la creación de Base de Datos, Diagramas Entidad-Relación, Consultas y demás operaciones típicas. La aplicación de esta interfaz en BDCEMENTERIO dará por terminados los objetivos de este manual.

CONTENIDO

I. INTRODUCCIÓN A POSTGRESQL

En los últimos años, el software de bases de datos ha experimentado un auge extraordinario, a raíz de la progresiva informatización de casi la totalidad de las empresas de hoy día. No es extraño pues, que existan multitud de gestores de bases de datos, programas que permiten manejar la información de modo sencillo. De este modo tenemos Oracle™, Microsoft SQL Server™, Borland Interbase™ entre otras. Las soluciones software que hemos citado son comerciales. Como siempre, en el mundo del software libre, siempre que se necesita algo, tarde o temprano se implementa. Así tenemos MySQL™, gestor muy usado en la web (combinado con php y apache) o PostgreSQL™, que será el gestor que trataremos.

PostgreSQL es software libre. Concretamente está liberado bajo la licencia BSD, lo que significa que cualquiera puede disponer de su código fuente, modificarlo a voluntad y redistribuirlo libremente, PostgreSQL además de ser *libre* es gratuito y se puede descargar libremente de su página web para multitud de plataformas.

1. QUE ES POSTGRESQL

Postgres maneja los siguientes cuatro conceptos adicionales Básicos en la vía en que los usuarios pueden entender fácilmente el sistema
Clases – Herencia –Tipos – Funciones

Estas características colocan a PostgreSQL en la categoría de las Bases de Datos identificadas como *objeto-relaciónales*. 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. Postgres 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 Postgres fue pionera.

PostgreSQL es un sistema administrador de objeto relacionales de Base de Datos (Object-Relational DataBase Management System -* ORDBMS) basado en POSTGRES, versión 4,2[3], convertido en la universidad de California en el departamento de informática de Berkeley. PostgreSQL es un descendiente del open-source código original de Berkeley. Apoya una parte grande del estándar SQL: 2003 y ofrece muchas características modernas aportando potencia y flexibilidad adicional:

Complex queries
Foreign keys
Triggers

[3] http://db.cs.berkeley.edu//postgres.html

Views
Transactional integrity
Multiversion concurrency control

Además, PostgreSQL puede ser extendido por el usuario de muchas maneras, por Ejemplo agregando nuevos:

Tipos de Datos
Funciones
Operadores
Funciones agregadas
Métodos del índice
Idiomas procésales

Y debido a la licencia abierta, PostgreSQL puede ser utilizado, modificado, y distribuido gratuitamente para cualquier propósito, sea privado, comercial, o académico.

2. BREVE HISTORIA DE POSTGRESQL

El Sistema Gestor de Base de Datos Relaciónales Orientadas a Objetos conocida como PostgreSQL (y brevemente llamada Postgres95) está derivada del paquete Postgres escrito en Berkeley. Con cerca de una década de desarrollo tras ella, PostgreSQL es la Base de Datos de código abierto más avanzada hoy día, ofreciendo control de concurrencia multi-versión, soportando casi toda la sintaxis SQL (incluyendo subconsultas, *transacciones*, tipos y funciones definidas por el usuario), contando también con un amplio conjunto de enlaces con lenguajes de programación (incluyendo C, C++, Java, Perl, PHP, Tul y Python)

2.1 El proyecto postgres de Berkeley

La implementación del DBMS Postgres comenzó en 1986. Los conceptos iniciales para el sistema fueron presentados en *The Design of Postgres*[4] y la definición del modelo de Datos inicial apareció en *The Postgres Data Model*. El diseño del sistema de reglas fue descrito en ese momento en *The Design of the Postgres Rules System*. La lógica y arquitectura del gestor de almacenamiento fueron detalladas en *The Postgres Storage System*[5].

Postgres ha pasado por varias versiones mayores desde entonces. El primer sistema de pruebas fue operacional en 1987 y fue mostrado en la Conferencia ACM-SIGMOD de 1988. Lanzaron la Versión 1, descrita en *The Implementation of Postgres*, a unos pocos usuarios externos en Junio de 1989. En respuesta a una crítica del primer sistema de reglas (*A Commentary on the Postgres Rules System*), éste fue rediseñado (*On Rules, Procedures, Caching and Views in DataBase Systems*) y la Versión 2, que salió en Junio de 1990, lo tuvo incorporado. La Versión 3 apareció en 1991 y añadió una implementación

[4] http://es.tldp.org/PostgreSQL-es/web/navegable/todoPostgreSQL/STON86
[5] Ibid.,

para múltiples gestores de almacenamiento, un ejecutor de consultas mejorado y un nuevo sistema de reescritura de reglas. En su mayor parte, las siguientes versiones hasta el lanzamiento de Postgres95 se centraron en portabilidad y fiabilidad.

Postgres ha sido usado para implementar muchas aplicaciones de investigación y producción. Entre ellas: un sistema de análisis de Datos financieros, un paquete de monitorización de rendimiento de motores a reacción, una Base de Datos de seguimiento de asteroides y varios sistemas de información geográfica. Postgres se ha usado también como una herramienta educativa en varias universidades. Finalmente, Illustra Information Technologies[6] (posteriormente absorbida por Informix[7]) tomó el código y lo comercializó. Postgres llegó a ser el principal gestor de Datos para el proyecto científico de computación Sequoia 2000 a finales de 1992.
El tamaño de la comunidad de usuarios externos casi se duplicó durante 1993. Pronto se hizo obvio que el mantenimiento del código y las tareas de soporte estaban ocupando tiempo que debía dedicarse a la investigación. En un esfuerzo por reducir esta carga, el proyecto terminó oficialmente con la Versión 4.2.

2.2 Postgres 95

En 1994, Andrew Yu[8] y Jolly Chen añadieron un intérprete de lenguaje SQL a Postgres. Postgres95 fue lanzada a continuación a la Web para que encontrara su propio hueco en el mundo como un descendiente de dominio público y código abierto del código original Postgres de Berkeley.

El código de Postgres95 fue adaptado a ANSI C y reducido en tamaño en un 25%. Muchos cambios internos mejoraron el rendimiento y la facilidad de mantenimiento. Postgres95 v1.0.x se ejecutaba en torno a un 30-50% más rápido en el Wisconsin Benchmark comparado con Postgres v4.2. Además de la corrección de errores, éstas fueron las principales mejoras:

> El lenguaje de consultas Postgres fue reemplazado con SQL (implementado en el Servidor) Las subconsultas no fueron soportadas hasta PostgreSQL (ver más abajo), pero podían ser emuladas en Postgres95 con funciones SQL definidas por el usuario. Las funciones agregadas fueron implementadas nuevamente. También se añadió una implementación de la cláusula GROUP BY.
> La interfaz *libpq* permaneció disponible para programas escritos en C. Además del programa de monitorización, se incluyó un nuevo programa (*psql*) para realizar consultas SQL interactivas usando la Librería GNU *readline*.
> Una nueva Librería de interfaz, *libpgtcl*, soportaba Clientes basados en *Tcl*. Un *shell* de Ejemplo, *pgtclsh*, aportaba nuevas órdenes *Tcl* para interactuar con el motor Postgres95 desde programas *Tcl*.
> La interfaz con objetos grandes fue revisada. Los objetos grandes de Inversión fueron el único mecanismo para almacenar objetos grandes (el sistema de archivos de Inversión fue eliminado).

[6] http://www.illustra.com/
[7] http://www-306.ibm.com/software/data/informix/
[8] Contacto ayu@informix.com

El sistema de reglas al nivel de instancia fue eliminado. Las reglas estaban todavía disponibles como reglas de reescritura.
Se distribuyó con el código fuente un breve Tutorial introduciendo las características comunes de SQL y de Postgres95.

Se utilizó GNU make (en vez de BSD make) para la compilación. Postgres95 también podía ser compilado con un gcc sin parches (la alineación de variables de longitud doble fue corregida)

2.3 PostgreSQL

En 1996, se hizo evidente que el nombre "Postgres95" no resistiría el paso del tiempo. PostgreSQL, seria el nuevo nombre para reflejar la relación entre el Postgres original y las versiones más recientes con capacidades SQL. Al mismo tiempo, los números de versión partieron de la 6.0, volviendo a la secuencia seguida originalmente por el proyecto Postgres.

Durante el desarrollo de Postgres95 se hizo hincapié en identificar y entenderlos problemas en el código del motor de Datos. Con PostgreSQL, el énfasis ha pasado a aumentar características y capacidades, aunque el trabajo continúa en todas las áreas.
Las principales mejoras en PostgreSQL incluyen:

Los bloqueos de tabla han sido sustituidos por el control de concurrencia Multi - versión, el cual permite a los accesos de sólo lectura continuar leyendo Datos consistentes durante la actualización de registros, y permite copias de seguridad en caliente desde pg_dump mientras la Base de Datos permanece disponible para consultas.
Se han implementado importantes características del motor de Datos, incluyendo subconsultas, valores por defecto, restricciones a valores en los campos (*Constraints*) y disparadores (*Triggers*)
Se han añadido características adicionales que cumplen el estándar SQL92, incluyendo claves primarias, identificadores entrecomillados, forzado de tipos cadenas literales, conversión de tipos y entrada de enteros, binarios y hexadecimales.
Los tipos internos han sido mejorados, incluyendo nuevos tipos de fecha / hora de rango amplio y soporte para tipos geométricos adicionales.
La velocidad del código del motor de Datos ha sido incrementada aproximadamente en un 20-40%, y su tiempo de arranque ha bajado el 80% desde que la versión 6.0 fue lanzada.

3. Y2K STATEMENT (EFECTO 2000)

Autor: Escrito por Thomas Lockhart el 22-10-1998.

El Equipo de Desarrollo Global (o Global Development Team) de PostgreSQL proporciona el árbol de código de software de Postgres como un servicio público, sin garantía y sin responsabilidad por su comportamiento o rendimiento. Sin embargo, en el momento de la escritura:

> El autor de este texto, voluntario en el equipo de soporte de Postgres desde Noviembre de 1996, no tiene constancia de ningún problema en el código de Postgres relacionado con los cambios de fecha en torno al 1 de Enero de 2000 (Y2K)
> El autor de este informe no tiene constancia de la existencia de informes sobre el problema del efecto 2000 no cubiertos en las pruebas de regresión, o en otro campo de uso, sobre versiones de Postgres recientes o de la versión actual. Podríamos haber esperado oír algo sobre problemas si existiesen, dada la Base que hay instalada y dada la participación activa de los usuarios en las listas de correo de soporte.
> Para años escritos con dos números, la transición significativa es 1970, no el año 2000; Ej. "70-01-01" se interpreta como "1970-01-01", mientras que "69-01-01" se interpreta como "2069-01-01".
> Los problemas relativos al efecto 2000 en el SO (sistema operativo) sobre el que esté instalado Postgres relacionados con la obtención de "la fecha actual" se pueden propagar y llegar a parecer problemas sobre el efecto 2000 producidos por Postgres.

Diríjase a The Gnu Project[9] y a The Perl Institute[10] para leer una discusión más profunda sobre el asunto del efecto 2000, particularmente en lo que tiene que ver con el open source o código abierto, código por el que no hay que pagar.

4. COPYRIGHT Y MARCAS

[9] http://www.gnu.org/software/year2000.html
[10] http://www.perl.org/about/y2k.html

documentación, incluso si la Universidad ha sido notificada de la posibilidad de tales daños.
La Universidad de California se recusa específicamente a ofrecer cualquier garantía, incluyendo, pero no limitada únicamente a, la garantía implícita de comerciabilidad y capacidad para cumplir un determinado propósito. El software que se distribuye aquí se entrega "tal y cual", y la Universidad de California no tiene ninguna obligación de mantenimiento, apoyo, actualización, mejoramiento o modificación.
Unix es una marca registrada de X/Open, Ltd. Sun4, SPARC, SunOS y Solaris son marcas registradas de Sun Microsystems, Inc. DEC, DECstation, Alpha AXP y ULTRIX son marcas registradas de Digital Equipment Corp. PA-RISC y HP-UX son marcas registradas de Hewlett-Packard Co. OSF/1 es marca registrada de Open Software Foundation.

5. FUNDAMENTOS ARQUITECTÓNICOS

Arquitectura del sistema. PostgreSQL utiliza un modelo Cliente / Servidor. En PostgreSQL la sesión consiste en los siguientes procesos (programas):

> Un proceso del Servidor, que maneja los archivos de Base de Datos, acepta conexiones a la Base de Datos de usos del Cliente, y *real*iza acciones en la Base de Datos a nombre de los Clientes. El programa del Servidor de la Base de Datos se llama postmaster.
> El uso del Cliente del usuario (frontend) que desea realizar operaciones en la Base de Datos. Los usos del Cliente pueden ser muy diversos en naturaleza: un Cliente podría ser una herramienta centrada en el texto, un uso gráfico, un web Server que tiene acceso a la Base de Datos para exhibir las páginas de la tela, o una herramienta especializada del mantenimiento de la Base de Datos. Algunos usos del Cliente se proveen del PostgreSQL distribución; la mayoría son desarrolladas por los usuarios.

Al igual que típico de usos Client / Server, el Cliente y el Servidor pueden estar en diversos anfitriones. En ese caso se comunican sobre una conexión de red de TCP/IP. Usted debe tener esto presente, porque los archivos que se pueden alcanzar en una máquina del Cliente no pudieron ser accesibles (ni pudieron ser solamente accesibles con un diverso nombre del archivo) en la máquina del Servidor de la Base de Datos.

El PostgreSQL el Servidor puede manejar conexiones concurrentes múltiples de Clientes. Para ese propósito empieza (las "bifurcaciones" un nuevo proceso para cada conexión. De ese punto encendido, el Cliente y el nuevo proceso del Servidor se comunican sin la intervención por el proceso original del postmaster. Así, el postmaster está funcionando siempre, para las conexiones del Cliente que esperan, mientras que el Cliente y los procesos asociados del Servidor vienen y van. (todo el esto es por supuesto invisible al usuario. Lo mencionamos solamente aquí para lo completo).

I. INICIANDO POSTMASTER

Antes de usar PostgreSQL usted necesita instalarlo. Es posible que PostgreSQL ya este instalado en su estación de trabajo, porque se ha incluido junto con la distribución del sistema operativo o porque el administrador del sistema ya lo instaló. Si ese es el caso, usted debe obtener documentación del sistema operativo o instrucciones del administrador del sistema sobre cómo acceder a PostgreSQL.

Si no está seguro de tener disponible PostgreSQL usted puede instalarlo. Esta tarea no es difícil y puede ser un buen ejercicio. PostgreSQL puede ser instalado por cualquier usuario con permisos; ningún superusuario (root) requiere acceso.

Si usted está instalando PostgreSQL vea el Manual Intermedio de Feneed y devuélvase a esta guía cuando la instalación este completa. Asegúrese de seguir correctamente la sección para preparar las variables de ambiente apropiadas.

Si el administrador de la estación de trabajo no ha preparado las cosas de una manera predeterminada, usted puede tener un poco más de trabajo por hacer. Por Ejemplo, si la máquina de Servidor de la Base de Datos es una máquina remota, usted necesitará poner el ambiente de PGHOST con nombre de la Base de Datos en la máquina de Servidor. El ambiente PGPORT puede que también necesite ser activado. La línea de fondo es esta: si usted intenta empezar un programa con la aplicación y no se puede conectar a la Base de Datos, usted debe consultar al administrador de la estación.

Visite http://www.PostgreSQL.org/docs/8.0/static/index.html para asegurarse de que el ambiente si es el apropiado.

6. CONECTAR EL POSTMASTER

Siguiendo las instrucciones que amablemente nos da *initdb* nos conectamos al Servidor, podemos añadir el parámetro -i para admitir conexiones TCP/IP.

La siguiente instrucción es un ejemplo para conexiones TCP/IP:

```
linux:~#  /usr/bin/postmaster -o -i -D /home/jav/db/data &
```

Para trabajar de manera local simplemente nos ponemos como usuario postgres y digitamos la dirección donde ser encuentra el *initdb* y el *postmaster*.

```
Linux: ~# su -- postgres
postgres: ~# /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
postgres:~# /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
>logfile 2>&1 &1
```

7. CREAR UNA BASE DE DATOS

La primera prueba para considerar si usted puede tener acceso al Servidor de la Base de Datos es intentar crear una Base de Datos.

Un Servidor de PostgreSQL corriente puede manejar muchas Bases de Datos. Generalmente, una Base de Datos separada se utiliza para cada proyecto o para cada usuario. Para crear una nueva Base de Datos, usted debe utilizar el siguiente comando:

```
$ CREATEDB BDCEMENTERIO
```

Donde BDCEMENTERIO es el nombre que le daremos a la Base de Datos que utilizaremos como guía principal en este Manual.

Esto debe producir como respuesta:

```
CREATE DATABASE
```

Si es así, este paso es acertado y puede pasar esta sección.
Si usted ve un mensaje similar a:

```
CREATEDB: command not found
```

Entonces PostgreSQL no fue instalado correctamente o no fue instalado en la ruta fijada. Intente llamar el comando con una ruta distinta:

```
$ /usr/local/pgsql/bin/CREATEDB BDCEMENTERIO
```

La ruta en su estación de trabajo pudo ser diferente. Entre en contacto con el administrador de la estación o compruebe las instrucciones de instalación para corregir.

Otra respuesta podía ser ésta:

```
CREATEDB: could not connect TO DataBase template1: could not connect TO
server:
No such file or directory
Is the server running locally and accepting connections on Unix domain
socket "/tmp/.s.PGSQL.5432"
```

Esto significa que el Servidor no fue encendido o inicializado, donde CREATEDB contaba con él. Una vez más compruebe las instrucciones de instalación o consulte a administrador.

Otra respuesta podía ser ésta:

```
CREATEDB: could not connect TO DataBase template1: FATAL:    user
"postgres" does not exist
```

Donde no se menciona su nombre para la conexión. Esto sucederá si el administrador o usted no ha creado una cuenta de usuario PostgreSQL para usted. (Las cuentas de

usuario PostgreSQL son distintas a las cuentas de usuario del sistema operativo.) Ver Manual intermedio de FENNED para obtener ayuda en la creación de cuentas.
Para especificar el nombre de usuario postgres también puede consultar el interruptor -U o fijar la variable de entorno de PGUSER.

Si usted tiene una cuenta de usuario pero no tiene los privilegios requeridos para crear una Base de Datos, usted verá el siguiente error:

```
CREATEDB: DataBase creation failed: ERROR:  permission denied TO create
DataBase
```

No todos los usuarios tienen autorización para crear nuevas Bases de Datos. Si PostgreSQL rechaza crear Bases de Datos para su usuario entonces el administrador de la estación deberá concederle el permiso de crear Bases de Datos . Consulte al administrador de la estación si esto ocurre. Si usted mismo instaló PostgreSQL entonces usted debe abrir una sesión para los propósitos de este Manual bajo la cuenta de usuario postgres.

Usted también puede crear Bases de Datos con otros nombres. PostgreSQL le permite crear cualquier número de Bases de Datos en una estación dada. Los nombres de la Base de Datos deben tener un primer carácter alfabético y se limitan a 63 caracteres en longitud. Una opción conveniente es crear una Base de Datos con el mismo nombre actual del usuario. Muchas herramientas asumen ese nombre en la Base de Datos por defecto, así que puede salvarlo de digitar. Para crear una Base de Datos , digite simplemente

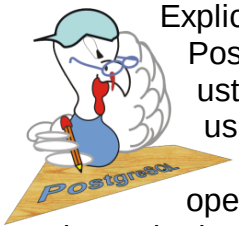
```
$ CREATEDB
```

8. BORRAR UNA BASE DE DATOS

Si no desea utilizar más su Base de Datos, usted puede quitarla. Por Ejemplo, si usted es el dueño (creador) de la Base de Datos BDCEMENTERIO, usted puede eliminarla usando el siguiente comando:

```
$ DROPBD BDCEMENTERIO
```

(Para este comando, el nombre de la Base de Datos no tiene como predeterminado el nombre de la cuenta del usuario. Usted siempre debe especificarlo.)



Explicación de porqué trabaja de esta manera: los nombres de usuario de PostgreSQL están a parte de las cuentas de usuario del sistema operativo. Si usted se conecta a una Base de Datos, usted debe elegir qué nombre de usuario PostgreSQL quiere conectar; si usted no lo hace, tendrá como valor predeterminado el mismo nombre que su cuenta actual del sistema operativo. La cuenta del usuario PostgreSQL que inicializa el Servidor y que tiene el mismo nombre que el usuario del Sistema Operativo, tendrá siempre permisos para crear Bases de Datos.

9. ACCEDER A UNA BASE DE DATOS

Una vez que usted haya creado una Base de Datos, usted puede tener acceso a ella:
Ejecutando la Terminal del programa interactivo PostgreSQL, llamada *psql, le* permite a usted entrar interactivamente, corregir, y ejecutar los comandos de SQL.
Usando una herramienta gráfica existente del frontend como PgAccess o una colección de oficina con la ayuda de ODBC puede crear y manipular una Base de Datos. Esta última posibilidad no la cubre este Manual.

Usted debe subir o iniciar *psql*, para probar los ejemplos de este Manual. La Base de Datos BDCEMENTERIO puede ser activada digitando el comando:

```
$ psql BDCEMENTERIO
```

Si usted uso el nombre de usuario de la Base de Datos entonces dejara predeterminado el nombre de cuenta del usuario en la línea de comando. Usted ya descubrió este esquema en la sección anterior.
psql, le saludarán con el mensaje siguiente:

```
Welcome TO psql 8.0.3, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon TO execute query
       \q TO quit

BDCEMENTERIO=>
```

La última línea podía ser:

```
 BDCEMENTERIO= #
```

Este símbolo en algunas distribuciones de Linux significaría que usted es un superusuario de la Base de Datos, probablemente aparecerá si usted mismo instaló PostgreSQL.

Si usted encuentra problemas al iniciar *psql* puede diagnosticar como CREATEDB el *psql* son similares, esto debe funcionar.

La última línea impresa por *psql* es el aviso, e indica que *psql* está conectado y que usted puede digitar las preguntas de SQL o PostgreSQL en un espacio de trabajo mantenido por el *psql*. Pruebe estos comandos:

```
BDCEMENTERIO=> SELECT version();
version
---------------------------------------------------------------
 PostgreSQL 8.0.3 on i586-pc-linux-gnu, compiled by GCC 2.96
(1 row)
```

```
BDCEMENTERIO=> SELECT current_date;
date
------------
 2002-08-31
(1 row)

BDCEMENTERIO=> SELECT 2 + 2;
 ?column?
----------
    4
(1 row)
```

El programa *psql* tiene un número de comandos internos que no son comandos SQL. Comienzan con el carácter backslash, " \ " . Algunos de estos comandos fueron enumerados en el mensaje de bienvenida. Por Ejemplo, usted puede conseguir ayuda de PostgreSQL digitando:

```
BDCEMENTERIO= > \h
```

Para salir de *psql*, digite

```
BDCEMENTERIO= > \q
```

Y *psql* parará y volverá a su línea normal de comando. (Para comandos, de tipo interno digite \? en *psql*.)
Pues bueno, parece que la cosa funciona. Ya tenemos todo en marcha y funcionando. En adelante pasaremos a cosas mayores.

III. EL LENGUAJE SQL

PostgreSQL es *un Sistema de Administración de Base de Datos Relacional* (RDBMS) Eso significa que es un sistema para la administración de Datos almacenados en relaciones. La relación es esencialmente un término matemático para la tabla. La noción de almacenar Datos en tablas es hoy tan común que podría parecer intrínsecamente obvio, sin embargo hay diferentes maneras de organizar las Bases de Datos . Los archivos y los directorios en los sistemas operativos análogos a Unix crean un Ejemplo de una Base de Datos jerárquica. Un desarrollo más moderno son las Bases de Datos orientadas a objetos.

Cada tabla es una colección nombrada de *filas*. Cada fila de una tabla dada tiene la misma colección de columnas *nombradas*, y cada columna estará con un tipo de dato específico. Mientras que las columnas tienen un orden fijo en cada fila, es importante recordar que SQL no garantiza de ninguna manera el orden de las filas dentro de la tabla (aunque pueden ser clasificadas explícitamente para mostrarlos).
Las tablas se agrupan en Bases de Datos, y una colección de Bases de Datos manejadas por un Servidor PostgreSQL en un Servidor constituye un cluster de la *Base de Datos*.

10. CREAR TABLAS

Antes de conocer los comandos para la creación de tablas, se enunciara el problema de la Base de Datos BDCEMENTERIO, el cual utilizaremos como guía principal en este Manual y revisaremos las características de una tabla.

Es conveniente que usted vaya pensando en el posible Diagrama E-R de este ejercicio mientras contextualizar el problema.

10.1 Enunciado del problema BDCEMENTERIO

Se desea implementar una Base de Datos para facilitar la gestión y administración de un cementerio, en dicha Base de Datos se contemplan diferentes categorías laborales, distintos tipos de enterramiento, facturas por los servicios prestados, incluso se permite que una familia posea su propio panteón para un determinado número de personas.
El BDCEMENTERIO está dividido en sectores, teniendo estos una capacidad y extensión variable que ha de quedar reflejada.
Asimismo se ha quiere tener información sobre los empleados mediante Datos personales como nombre y apellidos, dirección, teléfono, salario, antigüedad, etc.
Las categorías en las que se dividen los empleados son:

Enterradores – Jardineros - Administrativos

Los jardineros se ocuparán del cuidado de los sectores, de tal forma que un jardinero está al cuidado de un sector, aunque del cuidado de un sector pueden encargarse varios jardineros.
Asimismo, cada sector contendrá un determinado número de tumbas. Una tumba pertenece a un sector.
Las Tumbas pueden ser de uno de los siguientes tipos:

Nicho – Panteón - Fosa Común

Es necesario, además, almacenar información sobre el fallecido, así como de la persona (familiar) que se hará cargo de los costes del servicio (todo ello, obviamente identificado mediante los Datos personales y de interés para la empresa).
Cada fallecido es enterrado por un único enterrador, lógicamente el enterrador puede enterrar a mas de un fallecido durante su jornada laboral.
Los nichos tienen capacidad para una sola persona.
Sin embargo un panteón tiene capacidad para varias personas siendo lo normal 4, siendo por eso de tipo *smallint*.
La capacidad de una Fosa Común es superior a la de un panteón, y es de tipo *integer*. En este caso y en los dos anteriores asumimos la indivisibilidad del fallecido.
Además, los administrativos emiten facturas para los familiares, de tal forma que un administrativo puede emitir facturas a varios familiares, y un familiar puede recibir varias facturas. El único tipo de tumba que puede ser propiedad de un familiar es el panteón, siendo propiedad de una única persona, y dicha persona puede poseer varios panteones.

10.2 Las Tablas

Una tabla en una Base de Datos relacional es mucho más que una tabla en el papel: Consiste en filas y en columnas. El número y orden de las columnas son fijos, y cada columna tiene un nombre. El número de filas es variable -- refleja cuánto Datos se almacenan en un momento dado. SQL no garantiza el orden de las filas en una tabla. Cuando se lee una tabla, las filas aparecerán en un orden aleatorio, a menos que solicite explícitamente la clasificación. Además, SQL no asigna identificadores únicos a las filas, así que es posible tener varias filas totalmente idénticas en una tabla. Ésta es una consecuencia del modelo matemático que esta debajo de SQL pero generalmente no es deseable.

Cada columna tiene un tipo de Datos. El tipo de Datos obliga al sistema a valores posibles que se pueden asignar a una columna y puede asignarse la semántica a los Datos almacenados en la columna para poder utilizarlos en los cómputos. Por Ejemplo, una columna declarada para ser de un tipo numérico no aceptará secuencias de texto arbitrarias, y los Datos almacenados en tal columna se pueden utilizar para los cómputos matemáticos. Por el contrario, una columna declarada para ser de un tipo *string* aceptará casi cualquier clase de Datos pero no se presta para los cálculos matemáticos, aunque otras operaciones tales como encadenamiento *string* están disponibles.

PostgreSQL incluye un sistema importante de tipos de Datos incorporados que tienen bastantes usos. Los usuarios pueden también definir sus propios tipos de Datos. La mayoría de los tipos de Datos incorporados tienen nombres y semántica obvios. Algunos de los tipos de Datos frecuentemente usados son: *integer* para los números enteros, *numeric* para los números posiblemente fraccionarios, text para las cadenas de caracteres, *date* para las fechas, *time* para los valores del *time*-of-day, y *timestamp* para los valores que contienen la fecha y la hora.

Cuando usted crea muchas tablas relacionadas es sabio elegir un perfil de nombre consistente para las tablas y las columnas. Una opción es usar los nombres singulares o plurales para los nombres de la tabla.

Existe un límite en cuántas columnas puede contener una tabla. Dependiendo de los tipos de la columna, esta entre 250 y 1600. Sin embargo, definiendo una tabla en cualquier parte con muchas columnas son altamente inusuales y a menudo con un diseño cuestionable.

10.3 Comandos

Los siguientes ejemplos le servirán como guía para continuar desarrollando la Base de Datos BDCEMENTERIO. En estas tablas se especifica el nombre de cada tabla que desee crear, junto con todos los nombres de la columna y los tipos de Datos que manejara cada columna.

```
CREATE TABLE tiempo
 (  city            varchar(80),
```

```
temp_lo         int,            - - low temperature
temp_hi         int,            - - high temperature
prcp            real,           - - precipitation
date            date  );
```

Usted puede entrar a ello con *psql*. *psql* reconocerá si la línea de orden no está terminada con punto y coma.

En este primer Ejemplo el espacio blanco (es decir, espacios, tabs, y nuevas líneas) se puede utilizar Libremente en comandos SQL. Eso significa que usted puede digitar los comandos alineados indistintamente como se encuentra en el Ejemplo, o aún, todo en una línea.

Dos guiones (" -- ") introducen los comentarios, ello deja comentariado lo seguido hasta el final de la línea. SQL es insensible sobre las palabras claves y los identificadores.

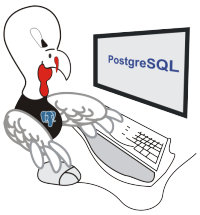
varchar(80) especifica un tipo de Datos que pueda almacenar cadenas de caracteres arbitrarias hasta de 80 caracteres de longitud. *int* es el tipo normal del número entero. El tipo *real* permite almacenar números de precisión *floating-point*. *Date* es el tipo de dato de fecha, si se define una columna con este nombre, puede prestarse para confusiones.

PostgreSQL soporta los tipos estándares del SQL *int, smallint, real, double precision, char(N), varchar(N), date, time, timestamp e interval*, así como otros tipos de utilidad general y un sistema rico de tipos geométricos. PostgreSQL puede ser modificado por el usuario para requisitos particulares con un número arbitrario de los tipos de Datos. Sin embargo, los nombres del tipo no son palabras claves sintácticas, excepto donde se requiera apoyar casos especiales en el estándar del SQL.

El segundo Ejemplo almacenará ciudades y su localización geográfica asociada:

```
CREATE TABLE ciudades
 (name            varchar (80),
    location        point);
```

El tipo *point* es un Ejemplo de tipo de dato específico en PostgreSQL.



Si usted no necesita una tabla creada de más o si desea reconstruirla usted puede quitarla son el siguiente comando:

DROP TABLE *tablename*;

11. DIAGRAMA E-R BDCEMENTERIO

El siguiente diagrama E-R es una alternativa atendiendo al enunciado del problema, la estructura del diagrama puede quedar como sigue:

Entidades:

- Empleado (que constará a su vez de tres tipos: Jardinero, Enterrador y Administrativo).
- Sector (en los que está dividido el cementerio).
- Tumba (puede ser de tres tipos: Nicho, Panteón y Fosa Común).
- Fallecido (Representa a la persona muerta mediante los atributos detallados mas tarde).
- Familiar (Es necesario enviar la factura a los familiares del fallecido).

Relaciones:

- JarSec (Indica la relación de los jardineros con los sectores del cementerio)
- TumSec (Relación que se da entre las tumbas y los sectores en que están ubicadas)
- EnFa (Es la relación que se establece entre los enterradores y los fallecidos).
- Factura (Representa la venta de una tumba a la familia del fallecido).
- NiFa (Indica si el fallecido tiene asignado un Nicho).
- FoFa (Indica si el fallecido se encuentra en una Fosa Común).
- PanFa (Indica si el fallecido se encuentra en un Panteón).
- FamFa (Es la relación establecida entre el fallecido y su familia).
- PaFam (Relación que indica la posesión de un panteón por parte de una familia).

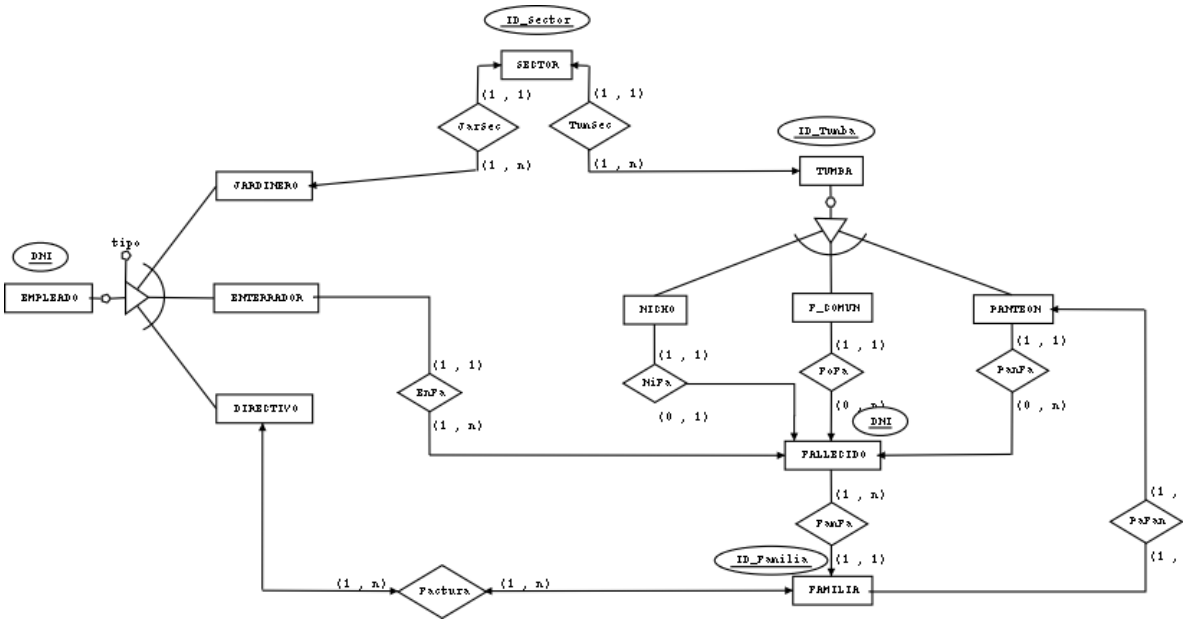


Figura Nº 1 Diagrama E-R BDCEMENTERIO

El único inconveniente en la creación del Diagrama E-R que hemos observado, es la notación que utiliza para la representación de los atributos, enmarcándolos en una elipse.

[11] Debido a esto, decidimos representar en el diagrama sólo los atributos que son clave primaria (para consultar detalles sobre atributos, ver lista anterior)

Por lo que se refiere a la especialización de la entidad Empleado, podemos decir que consta de tres tipos: Jardinero, Enterrador y Administrativo. En cuanto a la ligadura, diremos que se trata de conjuntos de entidades disjuntos cuya ligadura de completitud es de tipo total, posee además un atributo llamado Tipo que identifica cada una de las tres categorías.
La entidad Tumba también posee una especialización en la que se diferencian tres tipos de subclases, a saber: Nicho, Panteón y Fosa Común, se trata de entidades disjuntas cuya participación es total, al pasarlo a tablas aparecerá un atributo en la tabla Tumba que identificará el tipo de Tumba de que se trata.

12. POBLAR UNA TABLA

Las tablas que se nombran a continuación se encuentran definidas en la sección 2.2 Comandos Cap. III. La declaración INSERT se utiliza para poblar una tabla, en el Ejemplo de la tabla tiempo tenemos:

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

Observe que todos los tipos de Datos utilizan formatos algo obvios en la entrada. Las constantes que no son valores numéricos simples se deben rodear generalmente por apóstrofes (), como en el Ejemplo. El tipo *date* es absolutamente flexible para la inserción de Datos, pero para este Manual en particular usaremos el formato mostrado aquí, para evitar ambigüedades.
El tipo *point* requiere un par coordinado como entrada, según lo mostrado aquí:

```
INSERT INTO ciudades VALUES ('San Francisco', '(-194.0,53.0)');
```

La notación usada anteriormente requiere que usted recuerde el orden de las columnas. UnA notación alternativa permite que usted liste las columnas explícitamente:

```
INSERT INTO tiempo (city, temp_lo, temp_hi, prcp, date)
    VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

Si usted desea puede listar las columnas en distinto orden u omitir algunas columnas, si el dato es desconocido:

```
 INSERT INTO tiempo (date, city, temp_hi, temp_lo) no incluye prc VALUES
('1994-11-29', 'Hayward', 54, 37);
```

Muchos desarrolladores consideran que es mejor listar explícitamente las columnas, pues es más confiable que el orden implícito.

[11] Visite http://dia-installer.sourceforge.net/ Descargue una herramienta para el diseño de diagramas muy conocida por los usuarios de Linux como es Dia, se puede descargar en su versión para Windows 95/98/NT/ME/XP.

Incorpore por favor Datos con los comandos mostrados anteriormente, así usted tendrá ciertos Datos para trabajar en las secciones siguientes.

Usted también puede utilizar COPY para cargar cantidades grandes de Datos desde archivos planos de texto. Esto es generalmente más rápido porque el comando COPY es optimizado para este uso, mientras el comando INSERT permite menos flexibilidad.

13. CONSULTAR UNA TABLA

Para recuperar Datos de una tabla, se ~~in~~terrogall la tabla. La declaración SELECT de SQL se utiliza para hacer esto. La declaración se divide en un SELECT (la lista de columnas elegida se devolverá), un FROM (las tablas de las cuales se recupera los Datos) y una calificación opcional (especifica cualquier restricción) Por Ejemplo, para recuperar todas las filas de la tabla tiempo:

```
SELECT * FROM tiempo;
```

Aquí * es digitado para definir "todas las columnas". El mismo resultado sería obtenido con:

```
SELECT city, temp_lo, temp_hi, prcp, date FROM tiempo;
```

Mientras que SELECT * es útil para las consultas espontáneas, se considera una mal estilo para la producción de código, pues desde la adición de una columna en una tabla cambiaría los resultados.

La salida debe ser:

```
  city       | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      43 |      57 |    0 | 1994-11-29
 Hayward       |      37 |      54 |      | 1994-11-29
(3 rows)
```

Usted puede escribir las expresiones, no justamente referenciado columnas simples. Por Ejemplo, usted puede hacer:

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

Esto debe dar:

```
   city       | temp_avg |    date
---------------+----------+------------
 San Francisco |       48 | 1994-11-27
 San Francisco |       50 | 1994-11-29
 Hayward       |       45 | 1994-11-29
(3 rows)
```

La cláusula AS se utiliza para renombrar la columna de la salida. (La cláusula AS es opcional)

Una consulta se puede generar agregando una cláusula WHERE que especifica que filas se desean. La cláusula WHERE contiene (una expresión boleana de valor verdadero), y solamente para las filasen donde la expresión boleana es verdadera se devuelve el resultado. Se permite el uso de los operadores boléanos generales (Y, O, y NO). Por Ejemplo, si se quiere recuperar el tiempo de San Francisco en días lluviosos podemos hacer lo siguiente:

Resultado:

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

Usted puede solicitar que los resultados de una consulta sean devueltos en un orden:

```
SELECT * FROM tiempo
    ORDER BY city;
```

Resultado:

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 Hayward       |      37 |      54 |      | 1994-11-29
 San Francisco |      43 |      57 |    0 | 1994-11-29
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
```

En este Ejemplo, la orden de la clase no se especifica completamente, así que usted puede conseguir las filas de San Francisco en cualquier orden. Pero usted conseguiría siempre los resultados que desee especificando como sigue:

```
SELECT * FROM tiempo
    ORDER BY city, temp_lo;
```

Usted puede solicitar que las filas duplicadas sean quitadas en el resultado de una consulta:

```
SELECT DISTINCT city
    FROM tiempo;
```

Resultado:

```
     city
---------------
 Hayward
```

```
 San Francisco
(2 rows)
```

Aquí otra vez, el orden de las filas en el resultado pudo variar. Usted puede asegurar resultados constantes usando DISTINCT y ORDER BY juntos:

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

En algunos sistemas de Base de Datos, incluyendo las más viejas versiones de PostgreSQL la puesta en práctica de DISTINCT pide automáticamente las filas y ORDER BY es redundante. Esto no es requerido por el estándar SQL, pero en las versiones actuales de PostgreSQL no se garantiza que las filas con DISTINCT sean ordenadas.

14. JOINS ENTRE TABLAS

15. FUNICONES

16. ACTUALIZACIONES

17. ELIMINACIONES

Flata traducir todo esto

IV. CARACTERÍSTICAS AVANZADAS

En los capítulos anteriores hemos cubierto los fundamentos SQL para almacenar y tener acceso a Datos en PostgreSQL. Ahora discutiremos más características avanzadas de SQL que faciliten la administración y prevención de pérdida de Datos.
Este capítulo tiene como referencia los ejemplos encontrados en la sección 2.2 Comandos Cáp. III para cambiarlos o para mejorar, así que será una ventaja si usted ha leído esa sección.

18. VISTAS

Suponga que el listado combinado de los expedientes del tiempo y de la localización de la ciudad son de interés para un particular uso, pero usted no desea digitar la consulta cada vez que la necesite. Usted puede crear un *view* sobre la consulta, con un nombre aleatorio llamándola como una tabla ordinaria.

```
CREATE VIEW myview AS
    SELECT city, temp_lo, temp_hi, prcp, date, location
        FROM tiempo, ciudades
        WHERE city = name;
```

```
SELECT * FROM myview;
```

Haciendo uso libre de los *views* es un aspecto clave del buen diseño en una Base de Datos SQL. Los *views* le permiten encapsulan los detalles de la estructura de sus tablas, que pueden cambiar mientras que su uso se desarrolla, detrás de interfaces constantes.
Los *views* se pueden utilizar en casi cualquier lugar que una tabla. La construcción de *views* sobre otros *views* es frecuente.

19. FOREIGN KEY

Considere el problema siguiente: Usted quiere asegurarse de que nadie pueda insertar filas en la tabla tiempo y que no tiene una entrada que empareje en la tabla ciudades. Esto se llama mantener la *integridad referencial* de sus Datos. En sistemas de Base de Datos simplistas esto se llevaría a cabo (si en todo) si primero se mirara la tabla ciudades para verificar si existe un registro que empareje, y después insertando o rechazando los nuevos registros tiempo. Este acercamiento tiene un número de problemas y es muy incómodo, PostgreSQL puede hacer esto para usted.

La nueva declaración de las tablas sería algo como esto:

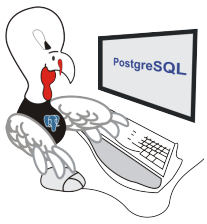
```
CREATE TABLE ciudades (
        city     varchar (80) primary Key,
        location point);

CREATE TABLE tiempo (
        city      varchar(80) references ciudades(city),
        temp_lo   int,
        temp_hi   int,
        prcp      real,
        date      date );
```

Ahora intente insertar un registro inválido:

```
INSERT INTO tiempo VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');
ERROR:  INSERT or update on table "tiempo" violates Foreign Key
constraint "tiempo_city_fkey"
DETAIL:  Key (city) = (Berkeley) is not present in table
"ciudades".
```

El comportamiento de Foreign Key puede tratar finamente su aplicación. No iremos más allá de este Ejemplo simple en este capitulo, ver la sección tal al capítulo 5 para más información.



La creación y uso correcto de las Foreign Key mejorará definitivamente la calidad en los usos de las Bases de Datos, así que anímese fuertemente a aprender sobre ellas.

20. *TRANSACCIONES*

Las *transacciones* son un concepto fundamental de todos los sistemas de Base de Datos . El punto esencial de una *transacción* es que enlaza pasos múltiples en una sola operación, todo o nada. Los estados intermedios entre los pasos no son visibles a otras *transacciones* concurrentes, y si ocurre una cierta falta que evita que la *transacción* termine, entonces ninguno de los pasos afecta la Base de Datos .
Por Ejemplo, considere una Base de Datos de un banco que contenga los balances para las varias cuentas de cliente, así como depósito total. Suponga que deseamos registrar un pago de $100,00 de la cuenta de Alicia a la cuenta de Bob. Simplificando, los comandos SQL para esto pueden ser:

```
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
WHERE name = (SELECT branch_name FROM accounts WHERE name   = 'Alice');
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

Los detalles de estos comandos no son importantes aquí; el punto importante es que hay varias actualizaciones separadas implicadas para lograr esta operación algo simple. Nuestros funcionarios de banco desearán asegurasen que sucedan todas estas actualizaciones, o ningunas de ellas. Ciertamente para que un fallo del sistema dé lugar, por Ejemplo si Bob recibe $100,00 que no fueron cargados por Alicia, Alicia a lo largo seguiría siendo un cliente feliz si le cargaron a Bob con el nombre de ella. Necesitamos una garantía, que si algo anda mal hasta cierto punto con la operación, ningunos de los pasos ejecutados anteriormente hagan efecto. Agrupar las actualizaciones en una *transacción* nos da esta garantía. Se dice que una *transacción* es *atómica*: desde el punto de vista de otras *transacciones*, con cualquiera suceso total o absoluto.

También deseamos una garantía que una vez que una *transacción* sea terminada y reconocida por el sistema de la Base de Datos, se haya registrado de hecho permanentemente y no se perderá aún cuando se presente una caída después de eso. Por Ejemplo, si estamos registrando un retiro de fondos de Bob, no deseamos en ninguna ocasión que el debito de su cuenta desaparezca en una caída solo momentos después que él sale de la puerta del banco. Una Base de Datos transaccional garantiza que todas las actualizaciones hechas por una *transacción* son almacenadas en una entrada permanente (es decir, en disco) antes de que la *transacción* se reporte por completo.

Otra característica importante de las Bases de Datos transaccionales se relaciona estrechamente con la noción de actualizaciones atómicas: cuando las *transacciones* múltiples están funcionando concurrentemente, cada una no debe poder ver los cambios incompletos realizados por otra. Por Ejemplo, si una *transacción* es la suma de los equilibrios de una sucursal, no incluiría el debito de la sucursal de Alicia sino no el crédito de la sucursal Bob, o viceversa. Así que las *transacciones* no deben ser todo o nada en términos de su efecto permanente sobre la Base de Datos, si no también en términos de su visibilidad mientras esto sucede. Las actualizaciones hechas hasta ahora por una

transacción abierta son invisibles a otras *transacciones* hasta que la *transacción* termine, después de lo cual todas las actualizaciones llegan a ser visibles simultáneamente.

En PostgreSQL una *transacción* es instalada rodeando los comandos SQL de la *transacción* con BEGIN y COMMIT. En nuestra *transacción* bancaria aparecería algo así:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
-- etc etc
```

Si, hasta cierto punto de la *transacción*, decidimos no seguir confiando (quizás acabamos de notar que el balance de Alicia fue negativo), nosotros podemos emitir la orden ROLLBACK en vez de COMMIT, y todas nuestras actualizaciones serán canceladas hasta ahora.
PostgreSQL trata realmente cada declaración SQL como si se ejecutara dentro de una *transacción*. Si usted no emite el comando BEGIN, entonces cada declaración individual tiene implícito un BEGIN y (sí es acertado) COMMIT envuelto alrededor de él. Un grupo de declaraciones rodeado de BEGIN y COMMIT a veces se llama un bloque de *transacción*.

Algunas las bibliotecas del cliente emiten las ordenes BEGIN y COMMIT automáticamente, de modo que usted pueda conseguir el efecto de los bloques de la *transacción* sin preguntar. Verifique la documentación para la interfaz que usted está utilizando.

Es posible controlar las declaraciones en una *transacción* de una manera más sencilla con el uso *de savepoints*. Savepoints permite que usted deseche partes de la *transacción* SELECT. Después de definir un *savepoint* con SAVEPOINT, usted puede volver de nuevo al *savepoint* con ROLLBACK TO. Los cambios de la Base de Datos de toda la *transacción* se definen con el *savepoint* y el rolling back lo desecha, y los cambios anteriores los guarda el *savepoint*.

Después de rodar de nuevo un *savepoint*, continúa siendo definido, así que usted puede volver de nuevo a él varias veces. Inversamente, si usted esta seguro usted no necesitará rodar de nuevo un *savepoint* particular, puede soltarlo, para que el sistema pueda liberar algunos recursos. Tenga presente que el lanzar o el rodar de nuevo a un *savepoint* automáticamente lanzará todos los *savepoints* que fueron definidos después de él.

Todo esto está sucediendo dentro del bloque de la *transacción*, así que ninguna de estas es visible a otras sesiones de la Base de Datos. Si usted compromete el bloque de la *transacción*, las acciones comprometidas llegan a ser visibles como unidad a otras sesiones, mientras que las acciones rolled-back nunca llegan a ser visibles en absoluto.

Recordando la Base de Datos del banco, supongamos que el debito $100,00 de la cuenta de Alicia, y el crédito de la cuenta de Bob, sólo para encontrar más adelante que debemos haber acreditado la cuenta de Wally. Nosotros podríamos hacerla usando *savepoints* así:

```
BEGIN;
```

```
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

Este Ejemplo, se simplifica demasiado, pero hay muchos controles a tener en cuenta cuando se usa *savepoints* sobre un bloque de la *transacción*. Por otra parte, ROLLBACK TO es la única manera de recuperar el control de un bloque de la *transacción*, este fue puesto en el sistema en el estado suspendido debido a un error, o cortocircuito, ocasionando el inicio total de la operación.

21. HERENCIA

La herencia es un concepto de Bases de Datos orientadas a objetos. Abre nuevas posibilidades interesantes de diseño de Base de Datos.
Vamos a crear dos tablas: Una tabla ciudades y una tabla capitals. Naturalmente, las capitales son también ciudades, así que usted deseara en algún momento listar las capitales implícitas cuando usted enumere todas las ciudades. Si usted es realmente listo usted puede crear un esquema diferente a este:

```
CREATE TABLE capitals (
  name       text,
  population real,
  altitude   int,     -- (in ft)
  state      char (2));

CREATE TABLE non_capitals (
  name       text,
  population real,
  altitude   int      -- (in ft));

CREATE VIEW ciudades AS
  SELECT name, population, altitude FROM capitals
    UNION
  SELECT name, population, altitude FROM non_capitals;
```

Este trabaja BIEN, pero se pone feo cuando usted necesita actualizar varias filas.

Una solución mejor es ésta:

```
CREATE TABLE ciudades (
  name       text,
  population real,
  altitude   int      -- (in ft));

CREATE TABLE capitals (
```

```
  state      char (2))
  INHERITS (ciudades);
```

En este caso, una fila de capitals hereda todas las columnas (nombre, población y altitud) de su *padre*, ciudades. El tipo del nombre de la columna es texto, en PostgreSQL nativo digite *string* para las cadenas de caracteres de longitud inconstantes. Las Capitales del Estado tienen una columna adicional, state, que muestra su estado. En PostgreSQL una tabla puede heredar desde cero o más otra tabla.

La jerarquía de la herencia es realmente un gráfico acíclico dirigido.

Por Ejemplo, la siguiente consulta encuentra los nombres de todas las Ciudades, incluyendo las Capitales del Estado, que están situadas en una altitud superior a 500 pies:

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

Resultado:

```
  name     | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
(3 rows)
```

Por otra parte, la siguiente consulta encuentra todas las Ciudades que no son Capitales del Estado y se sitúan en una altitud superior a 500 pies:

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

Resultado:

```
   name     | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
(2 rows)
```

Aquí el ONLY antes de ciudades indica que la consulta debe estar funcionando solamente en la tabla ciudades, y no debajo de las tablas de ciudades en la jerarquía de herencia. Muchos de los comandos que ya hemos discutido como el — SELECT, UPDATE y DELETE — soportan la notación ONLY.

Aunque la herencia es con frecuencia útil, no se ha integrado con *Constraints* o Foreign Key, que limita su utilidad. Vea la sección 5,5 para más detalle.

Una tabla puede heredar de más de una tabla padre, en este caso tiene la unión de las columnas definidas por las tablas padre (más cualquier columna declarada específicamente para la tabla hijo)

Una limitación de la herencia seria la característica índice que se hereda (incluyendo Unique Constraints) y los Constraints Foreign Key que aplican solamente a tablas singulares, no a los hijos de la herencia. Esto es verdad en ambos lados del referido y del referente de un Constraint Foreign Key. Así, en los términos del Ejemplo anterior:
Si declaramos ciudades.name para ser ÚNIQUE o PRIMARY KEY, ésta no detendría la tabla capitals de tener filas con los nombres que duplican filas en ciudades. Y esas filas duplicadas por defecto se mostrarían por defecto en las consultas ciudades. De hecho, las capitales por defecto no tendrían ningún constraint unique absoluto, así que podría contener múltiples filas con el mismo nombre. Usted podría agregar un constraint unique a capitals, pero esto no prevendría la duplicación comparada a ciudades.
Igualmente, si especificáramos ciudades.name REFERENCES a alguna otra tabla, este constraint no propagaría automáticamente a capitals. En este caso usted podría trabajar alrededor de él Manualmente agregando el mismo constraint REFERENCES a capitals.
Especificando que columnas de una tabla con REFERENCES ciudades (name) permitirían que la otra tabla contuviera nombres de ciudades, pero no los nombres capitals. No hay buen workaround para este caso.
Estas deficiencias probablemente serán arregladas en algún futuro, pero en entre tanto tenga cuidado y decida si la herencia es útil para su problema.

22. TABLAS BDCEMENTERIO

Ahora bien, ya usted conoce como se crea y define una tabla, y conoce una alternativa para el diagrama E-R en BDCEMENTERIO; la ayuda que le brindaremos a continuación solucionara pormenores tanto de las entidades como de las relaciones, especificando atributos, tipos de relaciones, cardinalidades y todo aquello que sea interesante destacar en BDCEMENTERIO.

El formato que encontraremos a continuación define los atributos propios de cada entidad y luego el código SQL para digitarlo en la línea de comando, posteriormente nos ayudaremos de PgAccess, una aplicación gráfica que permite gestionar PostgreSQL de un modo sencillo. Más adelante comentaremos un poco PgAccess.

La entidad Familiar tiene 5 atributos:

Nombre: Nombre del familiar al que se envía la factura.
Apellidos: Contiene los apellidos del familiar.
Telefono: Teléfono de contacto del familiar.
Direccion: Almacena la dirección (calle, numero, piso, etc).

ID_Familia: Código identificador de un familiar, es la clave primaria de esta tabla.

```
CREATE TABLE Familiar
    (Nombre varchar (25),
    Apellidos varchar (60),
    Telefono char (9),
    Direccion varchar (70),
         ID_Familia integer PRIMARY KEY);
```

La entidad Enterrador tiene 8 atributos:

Nombre: Representará el nombre del empleado.
Apellidos: Contienen los apellidos del empleado.
Dirección: Almacena la dirección (calle, numero, piso, etc).
Teléfono: Número de teléfono de contacto.
Telef_Movil: Número de teléfono móvil.
Antigüedad: Años de servicio en la empresa.
Salario: Sueldo en Euros.
DNI: Contiene el número del DNI, es la clave primaria de esta entidad.

```
CREATE TABLE Enterrador
    (Nombre varchar (25),
    Apellidos varchar (60),
    Direccion varchar (70),
    Telefono char (9),
    Telef_Movil char (9),
    Antiguedad integer,
    Salario numeric (10, 2) CHECK ((Salario < 2000) and (Salario
>1400)),
    Dni varchar (9) PRIMARY KEY);
```

La entidad Administrativo tiene 8 atributos:

Nombre: Representará el nombre del empleado.
Apellidos: Contienen los apellidos del empleado.
Dirección: Almacena la dirección (calle, numero, piso, etc).
Teléfono: Número de teléfono de contacto.
Telef_Movil: Número de teléfono móvil.
Antigüedad: Años de servicio en la empresa.
Salario: Sueldo en Euros.
DNI: Contiene el número del DNI, es la clave primaria de esta entidad.

```
CREATE TABLE Administrativo
    (Nombre varchar (25),
    Apellidos varchar (60),
    Direccion varchar (70),
    Telefono char (9),
    Telef_Movil char (9),
    Antiguedad integer,
    Salario numeric (10,2) CHECK ((Salario <=1400) and (Salario
>1000)),
    Dni varchar (9) PRIMARY KEY);
```

La entidad Sector tiene 4 atributos:

Nombre: Nombre de cada sector o zona del cementerio.
ID_Sector: Código identificador de zona
Superficie: Extensión en m2
Capacidad: Número de fallecidos que puede alojar.

```
CREATE TABLE Sector
    (ID_Sector integer PRIMARY KEY,
    Nombre varchar (30),
    Superficie integer CHECK (Superficie < 1000),
    Capacidad integer);
```

La entidad Jardinero tiene 9 atributos:

Nombre: Representará el nombre del empleado.
Apellidos: Contienen los apellidos del empleado.
Dirección: Almacena la dirección (calle, numero, piso, etc).
Teléfono: Número de teléfono de contacto.
Antigüedad: Años de servicio en la empresa.
Salario: Sueldo en Euros
Sector: El sector del BDCEMENTERIO donde trabaja. Clave ajena tomada de Sector.
DNI: Contiene el número del DNI, es la clave primaria de esta entidad.

```
CREATE TABLE Jardinero
    (Nombre varchar (25),
    Apellidos varchar (60),
    Direccion varchar (70),
    Telefono char (9),
    Telef_Movil char (9),
    Antiguedad integer,
    Salario numeric (10, 2) CHECK ((Salario <=1000) and (Salario
>850)),
    Sector integer NOT NULL DEFAULT 0,
    CONSTRAINT ajena_Sector FOREIGN KEY (Sector)
        REFERENCES Sector (ID_Sector)
        ON DELETE SET DEFAULT
        ON UPDATE CASCADE,
    Dni varchar (9) PRIMARY KEY);
```

La entidad Tumba tiene 4 atributos:

ID_Tumba: Código identificador de tumba.
Tipo: Puede ser de tres tipos: Nicho, Panteón o Fosa Común.
Sector: Sector en que se encuentra la tumba. Clave ajena tomada de Sector.

```
CREATE TABLE Tumba
    (ID_Tumba integer PRIMARY KEY,
```

```
    Tipo varchar (10),
    Sector integer NOT NULL DEFAULT 0,
    CONSTRAINT ajena_Sector FOREIGN KEY (Sector)
        REFERENCES Sector (ID_Sector)
        ON DELETE SET DEFAULT
        ON UPDATE CASCADE);
```

La entidad Nicho tiene 3 atributos:

Altura: Altura del nicho
ID_Nicho: Código identificador de nicho. Clave primaria y Clave Ajena tomada de Tumba (ID_Tumba).
Inscripcion: Texto que figura en el.

```
CREATE TABLE Nicho
    (Altura smallint CHECK (Altura <= 5),
    ID_Nicho integer PRIMARY KEY,
    CONSTRAINT ajena_ID_Nicho FOREIGN KEY(ID_Nicho)
        REFERENCES Tumba (ID_TUMBA)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    Inscripcion text);
```

La entidad FosaComun tiene 3 atributos:

ID_Fosa: Código identificador de Fosa Común. Clave primaria y Clave Ajena tomada de Tumba (ID_Tumba).
Capacidad: Número de fallecidos que puede contener.

```
CREATE TABLE FosaComun
    (ID_Fosa integer PRIMARY KEY,
    CONSTRAINT ajena_ID_Fosa FOREIGN KEY(ID_Fosa)
    REFERENCES Tumba(ID_TUMBA)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    Capacidad integer CHECK (Capacidad <= 200) );
```

La entidad Panteon tiene 4 atributos:

ID_Panteon: Código identificador de panteon. Clave primaria y Clave Ajena tomada de Tumba (ID_Tumba).
ID_Familia: Código identificador de familia Clave ajena tomada de Familiar (ID_Familia).
Inscripcion: Texto que figura en el.
Capacidad: Número de fallecidos que puede contener.

```
CREATE TABLE Panteon
    (ID_Panteon integer PRIMARY KEY,
    CONSTRAINT ajena_ID_Panteon FOREIGN KEY(ID_Panteon)
        REFERENCES Tumba(ID_TUMBA)
        ON DELETE CASCADE
```

```
        ON UPDATE CASCADE,
    ID_Familia integer,
    CONSTRAINT ajena_ID_Familia FOREIGN KEY(ID_Familia)
        REFERENCES Familiar(ID_Familia)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    Capacidad smallint Check (Capacidad not > 6),
    Inscripcion text );
```

La entidad Factura tiene 5 atributos:

Cantidad: Total a pagar por la familia.
Fecha: Fecha en que se emite la factura.
Clave_Factura: Clave primaria (Fecha,ID_Familia,ID_Admin).
ID_Familia: Código identificador de familia. Clave ajena tomada de Familiar.
ID_Admin:Código identificador de Administrativo. Clave ajena tomada de Administrativo (DNI).

```
CREATE TABLE Factura
        (Cantidad numeric(10,2),
        Fecha Date,
        ID_Familia integer,
        ID_Admin varchar(9),
        CONSTRAINT Clave_Factura PRIMARY
KEY(Fecha,ID_Familia,ID_Admin),
        CONSTRAINT ajena_ID_Admin FOREIGN KEY(ID_Admin)
                REFERENCES Administrativo(Dni)
                ON DELETE CASCADE
                ON UPDATE CASCADE,
        CONSTRAINT ajena_ID_Familia FOREIGN KEY(ID_Familia)
                REFERENCES Familiar(ID_Familia)
                ON DELETE CASCADE
                ON UPDATE CASCADE );
```

La entidad Fallecido tiene 7 atributos:

Nombre: Representará el nombre del fallecido.
Apellidos: Contienen los apellidos del fallecido.
FechaNacimiento: Almacena la fecha de nacimiento del fallecido.
FechaMuerte: Almacena la fecha de la muerte del fallecido.
Enterrador: Código q identifica al enterrador encargado del entierro. Clave Ajena tomada de Enterrador (DNI).
ID_Familia: Código q identifica a la familia del fallecido. Clave Ajena tomada de Familiar (ID_Familia).
Tumba: Código q identifica la tumba del fallecido. Clave Ajena tomada de Tumba (ID_Tumba).

```
CREATE TABLE Fallecido
        (
        Nombre varchar (25),
        Apellidos varchar (60),
```

```
FechaNacimiento date,
FechaMuerte date CHECK (FechaMuerte >= FechaNacimiento),
Enterrador varchar(9) NOT NULL,
ID_Familia integer,
Tumba integer NOT NULL,
CONSTRAINT ajena_Enterrador FOREIGN KEY(Enterrador)
        REFERENCES Enterrador(Dni)
        ON DELETE SET NULL
        ON UPDATE CASCADE,
CONSTRAINT ajena_Tumba FOREIGN KEY(Tumba)
        REFERENCES Tumba(ID_Tumba)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
CONSTRAINT ajena_ID_Familia FOREIGN KEY(ID_Familia)
        REFERENCES Familiar(ID_Familia)
        ON DELETE SET NULL
        ON UPDATE CASCADE,
Dni varchar(9) PRIMARY KEY );
```

23. CONSULTAS BDCEMENTERIO

En esta sección mostraremos la manera de realizar consultas en la Base de Datos BDCEMENTERIO mediante unos ejemplos, tomados como salida de pantalla.
Un primer Ejemplo será una consulta que muestra todos los campos de la tabla Familiar.
Chango
A continuación mostramos las relaciones nombres y apellidos de Familiares y Fallecidos. Como puede observarse, es una consulta que afecta a dos tablas relacionadas por el campo ID_Familia.
Chango
Un Ejemplo de lo laborioso que puede resultar hacer una consulta para conocer el Dni del enterrador que se ocupó de un fallecido llamado Restituto

```
SELECT Fallecido.Nombre, Fallecido.Apellidos, Enterrador.Dni
FROM Fallecido INNER JOIN Enterrador
ON Fallecido.Enterrador = Enterrador.Dni
WHERE Fallecido.Nombre = 'Restituto';
```

Resultado:

```
  nombre   |   apellidos    |   dni
-----------+----------------+----------
 Restituto | Gracia de Dios | 10657493
(1 row)
```

Ejemplo que muestra la salida de una consulta para listar el nombre de aquellos enterradores cuyo sueldo sea superior a 160000 Pesos

```
SELECT Nombre, Apellidos, Salario
FROM Enterrador
```

```
WHERE Salario > 160000;
```

Resultado:

```
 nombre  |     apellidos      | salario
---------+--------------------+---------
 Marc    | Pérez              | 180000
 Gonzalo | González González   | 190000
 Luis    | Anté Bajo          | 175000
 Frutos  | Rojo del Bosque    | 186900
(4 rows)
```

Veamos ahora el uso de Alias para las columnas, mientras que en SQL es suficiente con ponerlo a continuación del campo o con comilla doble, en PostgreSQL es necesario utilizar AS de la siguiente manera:

```
SELECT Nombre as "Nombre de Pila", Apellidos, Dni AS "Código Indentificador"
FROM Enterrador;
```

Resultado:

```
 Nombre de Pila |     apellidos      | Código Indentificador
----------------+--------------------+-----------------------
 Juán Felipe    | García Sierra      | 71659874
 Marc           | Pérez              | 71545545
 Jacinto        | Rodríguez López    | 10657493
 Gonzalo        | González González  | 71234321
 Luis           | Anté Bajo          | 9632236
 Frutos         | Rojo del Bosque    | 10653298
(6 rows)
```

V. DEFINICIÓN DE DATOS

Este capitulo cubre cómo se crean las estructuras de la Base de Datos para que se mantengan y se relacionaran sus Datos. En una Base de Datos relacional, la información en bruto se almacena en tablas, así que se dedican la mayor parte de este capítulo para explicar cómo se crean y se modifican las tablas y qué características están disponibles para controlar esos Datos que se almacenan en las tablas. Posteriormente, discutimos cómo las tablas se pueden organizar en esquemas, y cómo se pueden asignar privilegios o permisos a las tablas. Finalmente, miraremos brevemente otras características que afecten el almacenaje de Datos, tal como vistas, funciones, y *Triggers*.

24. CONSTRAINTS

Las tablas a utilizar en este capitulo son las de product. Los tipos de Datos son una manera de limitar la clase de Datos que se pueden almacenar en una tabla. Para muchos usos, sin embargo, el *Constraints* que ellos proporcionan es demasiado tosco. Por Ejemplo, una columna que contiene un precio de producto debe aceptar probablemente

solo valores positivos. Pero no hay tipo de Datos que acepte solamente números positivos. Otro problema es que usted quiere obligar Datos de una columna con respecto a otras columnas o filas. Por Ejemplo, en una tabla que contiene información de productos, debe haber solamente una fila para cada número de producto.

A tal efecto, SQL permite que usted defina los *Constraints* en columnas y tablas. Los *Constraints* le dan tanto control sobre los Datos en sus tablas como usted desea. Si un usuario intenta almacenar Datos en una columna que violara un *Constraints*, se levanta un error. Esto se aplica incluso si el valor viene de la definición del valor predefinido.

25. CONSTRAINTS CHECK

Un check *Constraints* es del tipo *Constraints* más genérico. Le permite a usted especificar que un valor en cierta columna satisfaga una expresión boleana (con valor verdadero). Por Ejemplo, para solicitar los precios positivos del producto, usted podría utilizar:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0));
```

Como usted ve, la definición *Constraints* viene después del tipo de Datos, como una definición del valor predefinido. Los valores predefinidos y los *Constraints* se pueden enumerar en cualquier orden. Un check *Constraints* consiste en la palabra clave check seguido de una expresión en paréntesis. La expresión del check *Constraints* debe involucrar la columna, si no el *Constraints* no tendría demasiado sentido.

Usted puede también dar al *Constraints* un nombre separado. Esto clarifica los mensajes de error y permite referir al *Constraints* cuando usted necesita cambiarlo. La sintaxis es:

```
CREATE TABLE products (
    product_no integer,
    name text,
     price numeric CONSTRAINT positive_price
     CHECK (price > 0)  );
```

Así pues, para especificar un *Constraints* nombrado, utilice la palabra clave *Constraints* seguido de un identificador de la definición *Constraints*. (Si usted no especifica un nombre *Constraints* de esta manera, el sistema elige un nombre para usted)

Un check *Constraints* puede también referir a varias columnas. Si se almacena un precio normal y un precio descontado y usted desea asegurarse de que el precio descontado sea más bajo que el precio normal, se hará así:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric CHECK (discounted_price > 0),
    CHECK (price > discounted_price) );
```

Los dos primeros *Constraints* le deben ser familiares. El tercero utiliza una nueva sintaxis. No se une a una columna particular, pero aparece como una línea separada en la lista. Las definiciones con columna y estas definiciones del *Constraints* se pueden enumerar en orden aleatorio.

Decimos que los dos primeros *Constraints* son *Constraints* de la columna, considerando que el tercer *Constraints* es de la tabla porque se escribe por separado en cualquier lugar de la definición. Los *Constraints* de la columna se pueden también escribir como *Constraints* de la tabla, mientras que al contrario no es necesariamente posible, puesto que un *Constraints* de columna se supone, refiere solamente a la columna que une. (PostgreSQL no hace cumplir esa regla, pero usted debe seguirla si quiere que sus definiciones de tabla trabajen con otros sistemas de Base de Datos). El Ejemplo anterior también se pode escribir como:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price) );
```

O incluso

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND
     price > discounted_price) );
```

Es una cuestión de gusto.

Los nombres se pueden asignar a los *Constraints* de la tabla de la misma manera que para los *Constraints* de la columna:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
     CONSTRAINT valid_discount CHECK (price >   discounted_price));
```

Debe notarse que un check *Constraints* satisface si la expresión del check evalúa un valor verdadero o un valor nulo. Para asegurarse de que una columna no contiene valores nulos, el *Constraints* no-nulo descrito en la sección siguiente puede ser utilizado.

26. CONSTRAINTS NO- NULOS

Un *Constraints* not-null simplemente especifica que una columna no debe asumir el valor nulo. Un Ejemplo de la sintaxis es:

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric );
```

Un *Constraints* not-null se escribe siempre como un *Constraints* de la columna. Un *Constraints* not-null es funcionalmente equivalente a crear un check del *Constraints* check (column_name IS NOT NULL), pero en PostgreSQL crear un *Constraints* not-null explícito es más eficiente. El inconveniente es que usted no puede dar nombres explícitos a los *Constraints* not-null creados de esa manera.

Por supuesto, una columna puede tener más de un *Constraints*. Simplemente escriba los *Constraints* uno después de otro:

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0));
```

El orden no importa. No se determina en qué orden se comprueban los *Constraints*.
El *Constraints* not-null es lo contrario a *Constraints* null. Esto no significa que la columna debe ser nula, pues sería seguramente inútil. En cambio, este simplemente selecciona el comportamiento predefinido para que la columna pueda ser nula. El *Constraints* null no se define en el estándar SQL y no se debe utilizar en aplicaciones portátiles. (Esto se agrego a PostgreSQL para ser compatible con algunos otros sistemas de Base de Datos) Algunos usuarios, sin embargo, lo usan porque hace fácil accionar la palanca del *Constraints* en un archivo de la escritura. Por Ejemplo, usted podría comenzar con:

```
CREATE TABLE products (
    product_no integer NULL,
    name text NULL,
    price numeric NULL);
```

Y entonces inserte la palabra clave donde desee.

En muchos diseños de Base de Datos la mayoría de columnas no debe marcar nulo.

27. UNIQUE CONSTRAINTS

Los Unique Constraints aseguran de que los Datos contenidos en una columna o un grupo de columnas sean únicos con respecto a todas las filas de la tabla.
La sintaxis es:

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric);
```

Cuando está escrito como Constraints de la columna, y

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no) );
```

Cuando está escrito como Constraints de la tabla.

Si un único Constraints se refiere a un grupo de columnas, las columnas se enumeran separándolas por comas:

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c) );
```

Esto especifica que la combinación de valores en las columnas indicadas es única en la tabla completa.
Usted puede asignar su propio nombre para Unique Constraints, de la siguiente manera:

```
CREATE TABLE products (
    product_no integer CONSTRAINT must_be_different UNIQUE,
    name text,
    price numeric);
```

En general, un Unique Constraints se viola cuando hay dos o más filas en la tabla donde los valores son iguales en todas las columnas incluidas en el constraint. Sin embargo, los valores nulos no se consideran iguales en esta comparación. Eso significa que en la presencia de un Unique Constraints puede almacenar un número ilimitado de filas que contienen un valor nulo en por lo menos una de las columnas obligadas. Este comportamiento conforma el estándar SQL, pero hemos oído que otras Bases de Datos SQL no pueden seguir esta regla.

28. PRIMARY KEY

Técnicamente, un constraint Primary Keys es simplemente una combinación de un Constraint Unique y de un Constraint Not-Null. Así pues, que las siguientes dos definiciones de tabla aceptan los mismos Datos:

```
CREATE TABLE products (
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric);

CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric );
```

Las Primary Keys también se pueden usar para un grupo de columnas; la sintaxis es similar a la de Constraint Unique:

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c) );
```

Una Primary Key indica que una columna o un grupo de columnas pueden usarse como un identificador único para las filas de una tabla. (Esta es una consecuencia directa de la definición de una llave primaria. Observe que un Constraint Unique no lo hace, solo, proporcionar un identificador único porque no excluye valores nulos). Esto es útil para propósitos de documentación y para los usos del cliente. Por Ejemplo, una aplicación de GUI que permite modificar el valor de una fila probablemente necesite saber la Primary Key de una tabla para poder identificar únicamente filas.

Una tabla puede tener a lo sumo una Primary Key (mientras que puede tener muchos Constraints Unique y Not – Null). La teoría de Bases de Datos relacionales dicta que cada tabla debe tener una Primary Key.

PostgreSQL no hace cumplir esta regla, pero, generalmente es mejor seguirla.

29. FOREIGN KEY

Un constraint Foreign Key especifica que los valores en una columna (o un grupo de columnas) deben relacionar los valores que aparecen en cierta fila de otra tabla. Decimos que esto mantiene la *integridad referencial* entre dos tablas relacionadas.

Suponga que usted tiene la tabla products que ya hemos utilizado varias veces:

```
CREATE TABLE products (
```

```
    product_no integer PRIMARY KEY,
    name text,
    price numeric );
```

También asuma que tiene una tabla donde almacena órdenes de esos productos. Ahora queremos asegurarnos que la tabla orders contenga solamente pedidos de los productos que realmente existen. Definimos un Constraint Foreign Key en la tabla orders que refiera a la tabla products:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
    quantity integer );
```

Ahora es imposible crear órdenes con las entradas product_no que no aparecen en la tabla products. Decimos que en esta situación la tabla orders es la tabla que refiere y la tabla products es la tabla referida. Igualmente, hay columnas referentes y referidas.

Usted también puede acortar el comando anterior a:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products,
    quantity integer );
```

Porque en ausencia de una lista de columnas la Primary Key de la tabla referida se utiliza como la columna(s) referida.

Una Foreign Key también puede obligar y referirse a un grupo de columnas.

Como ya es usual, necesitamos escribirla en forma de constraint de tabla. Aquí está un Ejemplo ideado con la sintaxis:

```
CREATE TABLE t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2) );
```

30. MODIFICAR TABLAS

Cuando usted crea una tabla y comete un error, o los requisitos de la aplicación cambian, usted puede eliminar la tabla y crearla otra vez. Pero esto no es una opción conveniente si la tabla ya esta llena de Datos, o si la tabla es referida por objetos de otra Base de Datos (Por Ejemplo un Foreign Key constraint). PostgreSQL soluciona este problema proporcionando una familia de comandos para hacer modificaciones en las tablas existentes. Observe que esto es conceptualmente distinto a alterar los Datos contenidos en una tabla: aquí estamos interesados en alterar la definición o estructura de la tabla.

Usted puede: Agregar columnas, Quitar columnas, Agregue *Constraints*, Quitar los *Constraints*, Cambiar los valores predefinidos, Cambiar el tipo de Datos de una columna, Renombrar columnas, Renombrar tablas. Todas estas acciones se *reali*zan usando el comando ALTER TABLE.

30.1 Adicionar una columna

Para agregar una columna, utilice un comando así:

```
ALTER TABLE products ADD COLUMN description text;
```

La columna nueva esta llena inicialmente por valores predefinidos (null sí usted no especifica una cláusula DEFAULT)
Usted también puede definir simultáneamente *Constraints* en la columna, usando la siguiente sintaxis:

```
ALTER TABLE products ADD COLUMN description text CHECK (description <>
'');
```

De hecho todas las opciones que se puedan aplicar a una descripción de la columna con CREATE TABLE puede ser utilizado así. Tenga presente sin embargo que el valor predefinido debe satisfacer los *Constraints* dados, o el ADD fallará. Alternativamente, usted puede agregar *Constraints* (véase abajo) después de que haya rellenado la nueva columna correctamente.

30.2 Eliminar una columna

Para eliminar una columna, utilice el comando así:

```
ALTER TABLE products DROP COLUMN description;
```

Cualquier dato que estuviese en la columna desaparece. Los *Constraints* de una tabla que implican esa columna también son eliminados. Sin embargo, si la columna es referida por un Foreign Key constraint de otra tabla, PostgreSQL no eliminara silenciosamente ese constraint. Usted puede autorizar la eliminación de todo lo que dependa de la columna agregando CASCADE:

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

Vea la sección 5,10 para una descripción del mecanismo general que hay detrás de esto.

30.3 Agregar un constraint

Para agregar un constraint, observe la sintaxis del constraint para una tabla. Por Ejemplo:

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE
(product_no);
```

```
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES
product_groups;
```

Para agregar un constraint not- null, que no se puede escribir como constraint de una tabla, utilice esta sintaxis:

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

El constraint será comprobado inmediatamente, así que los Datos de la tabla deben satisfacer el constraint antes de que pueda ser agregado.

30.4 Eliminar un constraint

Para quitar un constraint usted necesita saber su nombre. Si usted le dio un nombre entonces le será fácil. Si no el sistema asignó un nombre aleatorio, que usted necesita descubrir. El comando *psql* \d tablename puede ayudarle; otras interfaces también le pueden proporcionar una manera de examinar los detalles de la tabla. Entonces, el comando se usa:

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

Si usted se está ocupando en la generación de un nombre para el constraint como $2, no se le olvide que deberá citarlo doble para que sea un identificador válido

Para eliminar un constraint not-null use:

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

Recuerde que los constraint not-null no tienen nombres

30.5 Cambiar el valor predefinido de una columna

Para fijar un nuevo nombre predefinido a una columna, utilice un comando así:

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

Observe que esto no afecta ninguna de las filas existentes en la tabla, para estos cambios se usara el comando INSERT.

Para eliminar cualquier valor predefinido, utilice:

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

Éste es eficientemente igual a poner el valor predefinido en nulo. Por consiguiente, no es un error eliminar un valor donde no se había definido, porque el valor predefinido esta implícito al valor nulo.

30.6 Cambiar el tipo de Datos de una columna

Para cambiar el tipo de Datos de una columna, utilice el comando así:

```
ALTER TABLE products ALTER COLUMN price TYPE numeric (10,2);
```

Esto tendrá éxito solamente si cada entrada existente en la columna se puede convertir al nuevo tipo de dato implícito. Si necesita realizar una conversión más compleja, usted puede agregar la cláusula USING que especifique cómo computar los nuevos valores desde los valores viejos.

PostgreSQL procurará convertir los valores predefinidos de la columna (cualquiera) al nuevo tipo, así como cualquier constraint que involucre la columna. Pero estas conversiones pueden fallar, o pueden producir resultados sorprendentes. Es a menudo mejor eliminar cualquier constraint de la columna antes de alterar su tipo y después agrega adecuadamente los constraint modificados.

30.7 Renombrar columnas

Para renombrar una columna use:

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

30.8 Renombrar tablas

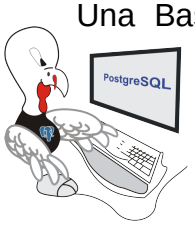
Para renombrar una tabla use:

```
ALTER TABLE products RENAME TO items;
```

31. SHEMAS

Un PostgreSQL los cluster de la Base de Datos contiene una o más Bases de Datos nombradas. Los usuarios y grupos de usuarios comparten el cluster completo, pero ningún otro dato es compartido es compartido por la Bases de Datos. Cualquier conexión del cliente dada al servidor puede tener acceso solamente a los Datos de una sola Base de Datos, la especificada en la conexión de petición.

> Los usuarios de un cluster no tienen necesariamente el privilegio de acceder a cada Base de Datos del cluster. El compartir los nombres del usuario significa que no puede haber diversos usuarios nombrados, por Ejemplo, Joe en dos Bases de Datos del mismo cluster; pero el sistema se puede configurar para permitir el acceso a Joe solamente a algunas de las Bases de Datos.



Una Base de Datos contiene uno o más esquemas *nombrados*, que a su vez contienen las tablas. Los esquemas también contienen otras clases de objetos nombrados, incluyendo tipos de Datos, de funciones y de operadores. El mismo nombre del objeto se puede utilizar en diversos esquemas sin conflicto; por Ejemplo: schema1 y myschema pueden contener las tablas nombradas mytable. En las Bases de Datos diferentes, los esquemas no se separan rígidamente: un usuario puede tener acceso a objetos en cualquiera de los esquemas en la Base de Datos donde se encuentra conectado, siempre en cuando tenga privilegios para hacer esto.

Hay varias razones por las que uno desearía utilizar esquemas:

Para permitir que muchos usuarios utilicen una Base de Datos sin interferir con otra.
Para organizar objetos de la Base de Datos en grupos lógicos para hacerlos más manejables.
Los usos de tercera persona se pueden poner en esquemas separados, así que no podrán generar conflicto con otros objetos.

Los esquemas son análogos a los directorios en el nivel del sistema operativo, a menos que los esquemas no puedan ser jerarquizados.

31.1 Crear un SHEMA

Para crear un esquema, utilice el comando CREATE SCHEMA. Dé un nombre opcional al esquema. Por Ejemplo:

```
CREATE SCHEMA myschema;
```

Para crear o para tener acceso a objetos en un esquema, escriba *un nombre cualificado* que consiste en el nombre del esquema y el nombre de la tabla separados por un punto:

```
schema.table
```

Esto trabaja dondequiera con el nombre de la tabla especificado, incluyendo los comandos de modificación de una tabla y los comandos de acceso a los Datos discutidos en los capítulos siguientes. (Para ser breves solo hablaremos de las tablas, pero las mismas ideas se aplican a otras clases de objetos nombrados, tales como tipos y funciones).

Realmente, la sintaxis aún más general es:

```
DataBase.schema.table
```

Puede ser utilizado también, pero esto en la actualidad esto simplemente es para *favorecer al* estándar SQL. Si usted escribe un nombre de Base de Datos , debe ser igual a la Base de Datos que usted se conecta.

Para crear una tabla con el nuevo esquema, use:

```
CREATE TABLE myschema.mytable (
```

```
  ...
);
```

Para eliminar un esquema si esta vacío (todos los objetos en él serán eliminados), utilice:

```
DROP SCHEMA myschema;
```

Para eliminar un esquema incluyendo todos los objetos contenidos, en cascada, utilice

```
DROP SCHEMA myschema CASCADE;
```

Vea la sección 5,10 para una descripción del mecanismo general que existe detrás de esto.

A menudo usted querrá crear un esquema poseído por algún otro (puesto que ésta es una de las maneras de restringir las actividades de sus usuarios a los namespaces bien definidos). La sintaxis para eso es:

```
CREATE SCHEMA schemaname AUTHORIZATION username;
```

Usted puede omitir el nombre del esquema, en caso de que el nombre del esquema sea igual al nombre del usuario. Vea la sección 5,8,6 para observar la utilidad de esto.

Los nombres del esquema que comienzan con pg_ son reservados para los propósitos del sistema y no pueden ser creados por los usuarios.

31.2 SHEMA publico

31.3 Buscar en un fichero un chema

31.4 SHEMAS y privilegios

31.5 SHEMAS del catalogo del sistema

31.6 Patrones de uso

31.7 Portabilidad traducir

32. MANIPULACIÓN DE DATOS

32.1 Insertar Datos

32.2 Actualizar Datos

32.3 Eliminar Datos

VI. INTERFAZ GRAFICA DE POSTGRESQL

Hasta ahora, hemos *real*izado toda la configuración de PostgreSQL mediante órdenes en una línea de comandos. Esto suele intimidar a algunos usuarios (los no acostumbrados a sistemas Unix) Para ellos se desarrolló una potente herramienta, llamada PgAccess. Está programado utilizando las librerías *Tcl*/*tk* por lo que puede correr en cualquier plataforma a la que haya sido portado *Tcl*/*tk* (Windows, Unix, Mac...) PgAccess es *libre* como todo el software que estamos utilizando. Si quiere más información puede consultar la página http://www.pgaccess.org/.

Un detalle importante para utilizar PgAccess es que a la hora de poner en marcha el servidor PostgreSQL hemos de habilitar conexiones a dicho servidor por TCP/IP, ya que PgAccess utiliza este tipo de conexión. El modo de hacer esto es añadir el parámetro -i al iniciar el servidor.

PgAccess permite hacer casi de todo. Desde crear tablas y modificarlas hasta realizar esquemas de la Base de Datos. Nosotros sólo veremos una pequeña muestra de su potencia. Incluso incluye un diseñador visual para realizar consultas, uno de formularios y otro de informes.

33. PGACCESS

PgAccess es una interfaz gráfica para el gestor de Bases de Datos PostgreSQL escrito por Constantin Teodorescu en el lenguaje *Tcl*/*Tk*. Permite al usuario interactuar con PostgreSQL de una manera similar a muchas aplicaciones de Bases de Datos para PC, con menús de opciones y diversas herramientas gráficas. Esto significa que el usuario puede evitar la línea de comandos para la mayoría de las tareas. PgAccess no cambia el modo de actuar de PostgreSQL, sólo hace más fácil su uso para aquellos que estén habituados a interfaces gráficas.

Obviamente, debes tener instalado y en marcha (corriendo) PostgreSQL, y *Tcl*/*Tk* en tu sistema antes de poder usar PgAccess.

PgAccess es una aplicación "open source" (código abierto) El código está disponible para el usuario, y puede ser modificado. El usuario puede arreglar fallos, o cambiar la manera de operar de una función. Puede que no estés interesado en cambiar su programación, pero tienes la opción de hacerlo. Si sientes que has hecho una mejora al programa, se te anima a compartirlo con los demás.

33.1 Crear una Bases de Datos

A tu derecha está la ventana que deberías ver al iniciar. La primera tarea para muchos usuarios debería ser el crear una Base de Datos.

Pincha el botón *Nuevo(a)* para lanzar la ventana que ves abajo. Esto te especificará la estructura de una nueva tabla. Es importante hacer notar que si no has especificado una Base de Datos al comenzar PgAccess, la tabla se creará en la Base de Datos llamada <username>, tu username.

Supón que quieres crear una tabla con entradas describiendo referencias bibliográficas en el campo chemistry. Elige un nombre para la tabla, por Ejemplo chemref que será fácil de encontrar e invocar en una lista. Introduce el nombre de la tabla en el primer campo de entrada.

Cuando ya tienes tablas en una Base de Datos, puedes usar el botón *Heredar* para desplegar una lista de tablas existentes para heredar características de otras tablas. En este Ejemplo, no debería haber previamente ninguna tabla para usar.

Introduce cada campo, dándole un nombre, tipo de campo y tamaño, si el tipo de campo no implica su tamaño. Esto es, si tu tipo de campo no es un número de secuencia, y has seleccionado *int2* como tipo de campo, no necesitarías especificar el tamaño del campo. Sin embargo, si tu segundo campo va a contener el autor de la referencia, y es del tipo *varchar*, tendrías que especificar cuántos caracters serán permitidos en le campo.

Conforme introduces cada campo, pincha el botón Añadir para añadirlo a la lista a la derecha de la ventana. Puedes cambiar la posición de los campos usando los botones de Mover arriba y Mover abajo, o borrar un campo si decides que no era lo que querías. Cuando hayas acabado de especificar los campos, pincha en el botón *Crear*.

33.2 Crear una tabla

A la derecha está la ventana que deberías ver cuando comienza PgAccess. La primera tarea para la mayoría de usuarios será la de crear una Base de Datos. Observa los 'botones' en la derecha de la ventana principal. Pinchando en ellos podrás ver diferentes nombres de *objects (objetos)* que se han almacenado en tu Base de Datos, los cuales deberían estar vacíos en este momento.

Pincha el botón *Tablas* y *Nueva* para lanzar la ventana que ves abajo. Esto te especificará la estructura de una nueva tabla. Es importante hacer notar que si no has especificado una Base de Datos al comenzar PgAccess, la tabla se creará en la Base de Datos llamada <username>, tu username.

Supón que quieres crear una tabla con entradas describiendo referencias bibliográficas en el campo psychology. Elige un nombre para la tabla, por Ejemplo psyref que será fácil de encontrar e invocar en una lista. Introduce el nombre de la tabla en el primer campo de entrada.

Cuando ya tienes tablas en una Base de Datos, puedes usar el botón *heredar* para desplegar una lista de tablas existentes para heredar características de otras tablas. En este Ejemplo, no debería haber previamente ninguna tabla para usar.
Introduce cada campo, dándole un nombre, tipo de campo y tamaño, si el tipo de campo no implica su tamaño. Esto es, si tu tipo de campo no es un número de secuencia, y has seleccionado *int2* como tipo de campo, no necesitarías especificar el tamaño del campo. Sin embargo, si tu segundo campo va a contener el autor de la referencia, y es del tipo *varchar*, tendrías que especificar cuántos caracteres serán permitidos en el campo.

Conforme introduces cada campo, pincha el botón *Añadir* para añadirlo a la lista a la derecha de la ventana. Puedes cambiar la posición de los campos usando los botones de *Mover arriba* y *Mover abajo*, o borrar un campo si decides que no era lo que querías. Cuando hayas acabado de especificar los campos, pincha en el botón *Crear* tabla.

33.3 Editar tablas

Una vez que tienes una tabla, puedes empezar a meter Datos. En la ventana principal de PgAccess, cuando pinchas en la opción *Tablas*, deberías ver la nueva tabla en la lista de tablas. Resaltando el nombre de la tabla pinchando en ella y pinchando en la opción *Abrir* abrirá esa tabla en otra ventana, como se muestra más abajo.

La manera más directa de añadir registros en una tabla es tecleando la información directamente en los campos. En la tabla que se muestra se han introducido dos registros. Como es común en esta interfaz de usuario, pulsar el ratón mientras el puntero está en un campo te permitirá introducir Datos con el teclado en ese campo. Este tipo de entradas es adecuado cuando llega en pequeñas porciones y de manera infrecuente, por Ejemplo al mantener una tabla de información de contactos de otras investigaciones. Sin embargo, ¿Qué haces cuando alguien te manda un e-mail con una lista entera de referencias de sus disertaciones de doctorado?

Esta situación se maneja mejor con el comando SQL COPY. Primero, la información se ha de pasar a formato de texto plano ASCII. Simplemente es un fichero de texto en el cual cada línea es un registro, y cada campo en cada registro está separado por un *delimitador* tal como una tilde (~). Los campos han de estar en el mismo orden que en los de tu tabla, y tiene que haber el mismo número de campos en cada registro que los que haya en tu tabla, sino podrías obtener resultados inesperados o ningún dato en absoluto. Digamos que creas un fichero de texto llamado newref.txt que comience así:

```
Cassileth, BR~Unorthodox Cancer Medicine~Cancer
Investigation~~1986~4~6~591-598 ...
```

Observa que hay dos tildes consecutivas para permitir el hecho de que en esa entrada en particular no hay ningún dato en el campo Editor. Entonces puedes ejecutar una *consulta (Query)* como sigue:

```
COPY psyref FROM '/home/jim/newref.txt' USING DELIMITERS '~';
```

Esto leerá los registros de newref.txt y los insertará en la tabla psyref.

33.4 Consultas

Supongamos que quiero saber todas las referencias en la tabla Ejemplo *psyref* en las cuales aparezca la palabra "alternative" en el título.

Pinchando en *Consultas*, después en *Nueva*, aparecerá una ventana Generador de consultas. Pinchando en el área debajo de los botones se permite al usuario introducir una *query (consulta)* SQL. La especificación de la *query (consulta)* debe ser exacta o PostgreSQL devolverá un mensage de error.

Lo que *query* hará será SELECT (seleccionar) aquellos registros de *psyref* que contengan la palabra "alternative" en cualquier lugar del campo *titulo*. El '*' indica que se deben devolver todos campos. Podrías querer que, por Ejemplo, sólo devolviera el campo *autor* si sólo estás interesado en qué autores han usado esa palabra en el título de su obra.

A la derecha está el registro que cumple esas condiciones, mostradas en Vista de tablas. Si quisieras guardar ésa consulta para usarla de nuevo, teclearías *Guardar definición de* la *consulta*. Después verás *altern* listado bajo *Queries* cuando vuelvas a la ventan principal. Pinchando en la opción *Guardar esta consulta como una vista* en Generador de consultas, el resultado de tu consulta se guardará como una *View (vista)* a la cual puedes acceder desde *Views* en la ventana principal. Pinchando la opción *Cerrar* saldrás de Query builder (constructor de consultas).

34. PGACCES BDCEMENTERIO

A continuación se presentan las tablas y algunas pantallas que generaría PgAccess, si todos los pasos anteriores, incluyendo la definición en SQL se *real*izaron adecuadamente, terminando así con los objetivos de este Manual Básico.

34.1 TABLAS

Aquí está la pantalla que muestra las tablas presentes en nuestra Base de Datos cementerio:

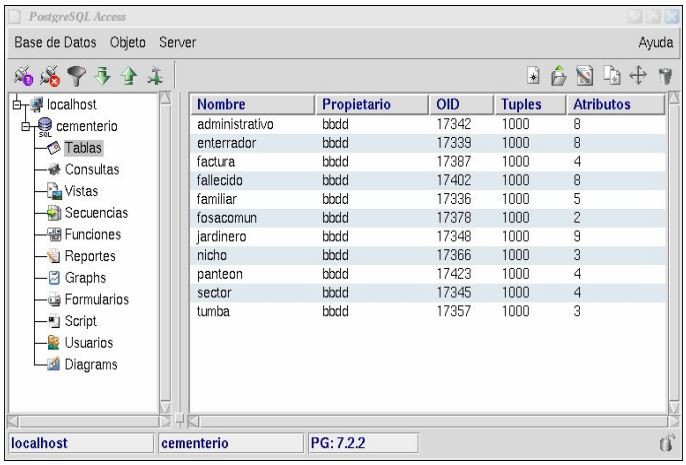


Figura Nº 2 Tablas BDCEMENTERIO

34.2 Diagrama entidad - relación

En la Figura N° 2 no indicamos los campos no nulos ni las claves primarias. De cualquier modo, el resultado es lo bastante claro como para hacernos una idea de la estructura que tendrá la Base de Datos.

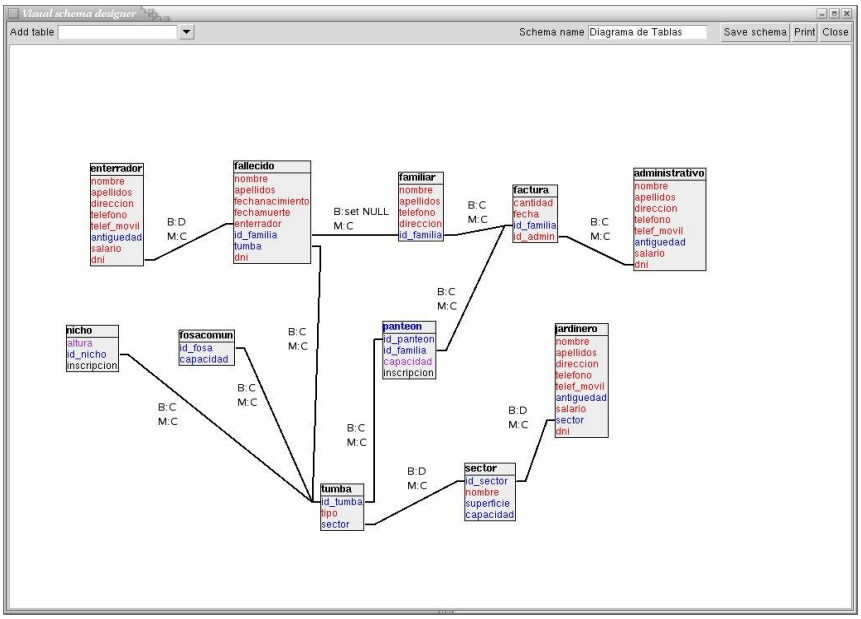


Figura N° 3 Diagrama Entidad – Relación BDCEMENTERIO

34.3 Varios BDCEMENTERIO

Veamos ahora de lo que es capaz PgAccess. Vamos a realizar una sencilla consulta para averiguar qué enterradores han enterrado a qué muertos. Según nuestro esquema, esto implica una consulta sobre dos tablas, la de fallecidos y la de muertos. Arrancamos el generador visual de consultas y arrastramos los campos que necesitamos al formulario:

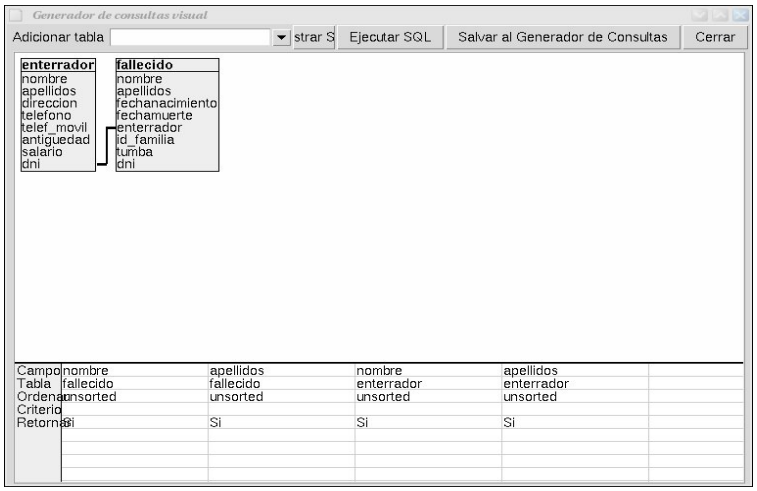
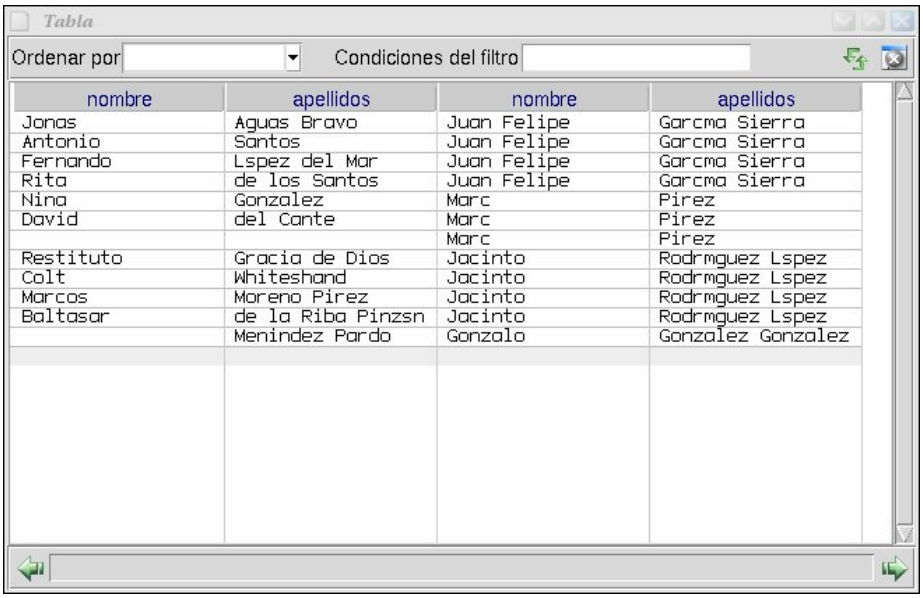


Figura N° 4 Consulta 1 BDCEMENTERIO

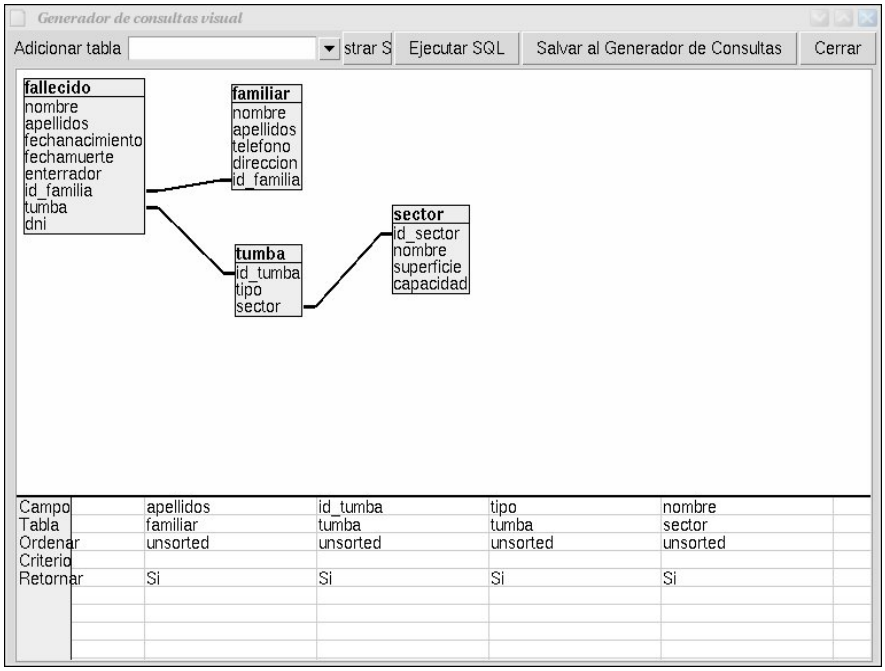
Una vez configurada nuestra consulta, podemos ver la sentencia SQL resultante o ejecutarla directamente. Los resultados se presentan del siguiente modo:



nombre	apellidos	nombre	apellidos
Jonas	Aguas Bravo	Juan Felipe	Garcma Sierra
Antonio	Santos	Juan Felipe	Garcma Sierra
Fernando	Lspez del Mar	Juan Felipe	Garcma Sierra
Rita	de los Santos	Juan Felipe	Garcma Sierra
Nina	Gonzalez	Marc	Pirez
David	del Cante	Marc	Pirez
		Marc	Pirez
Restituto	Gracia de Dios	Jacinto	Rodrmguez Lspez
Colt	Whiteshand	Jacinto	Rodrmguez Lspez
Marcos	Moreno Pirez	Jacinto	Rodrmguez Lspez
Baltasar	de la Riba Pinzsn	Jacinto	Rodrmguez Lspez
	Menindez Pardo	Gonzalo	Gonzalez Gonzalez

Figura N° 5 Resultados Consulta 1 BDCEMENTERIO

Esta era una consulta sencilla. Supongamos ahora que algún familiar visita el cementerio. Querrá saber dónde está enterrado su pariente. Realizaremos ahora una vista ya que esta parece una situación habitual. Éste será el esquema que diseñemos para implementar esta consulta:



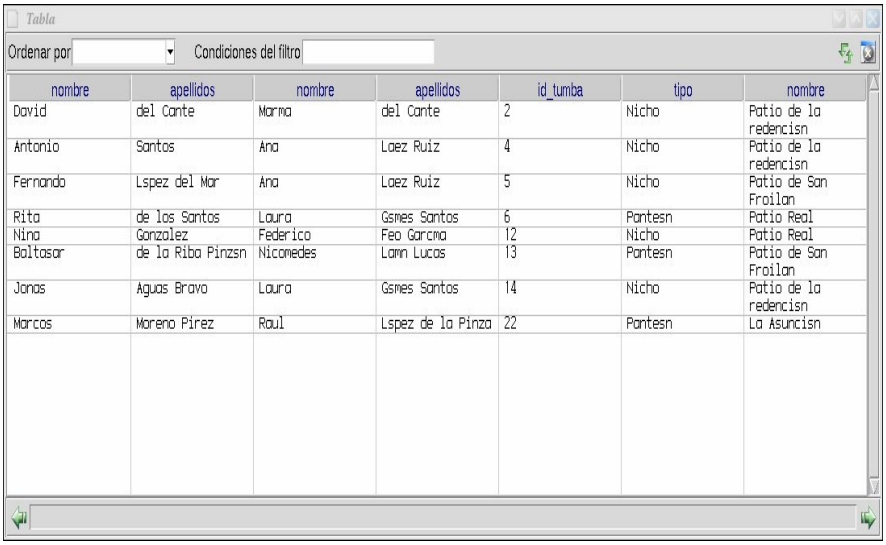
Campo		apellidos	id_tumba	tipo	nombre
Tabla		familiar	tumba	tumba	sector
Ordenar		unsorted	unsorted	unsorted	unsorted
Criterio					
Retornar		Si	Si	Si	Si

Figura N° 6 Consulta 2 BDCEMENTERIO

El código SQL resultante de esta consulta sería:

```
SELECT t0."nombre", t0."apellidos", t1."nombre", t1."apellidos",
t2."id_tumba", t2."tipo", t3."nombre"
FROM "fallecido" t0,"familiar" t1,"tumba" t2,"sector" t3
WHERE   (t2."id_tumba"=t0."tumba") and   (t3."id_sector"=t2."sector") and
(t1."id_familia"=t0."id_familia")
```

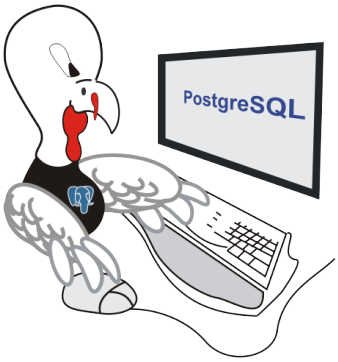
Los resultados de esta consulta serán:



nombre	apellidos	nombre	apellidos	id_tumba	tipo	nombre
David	del Cante	Marma	del Cante	2	Nicho	Patio de la redencisn
Antonio	Santos	Ana	Laez Ruiz	4	Nicho	Patio de la redencisn
Fernando	Lspez del Mar	Ana	Laez Ruiz	5	Nicho	Patio de San Froilan
Rita	de los Santos	Laura	Gsmes Santos	6	Pantesn	Patio Real
Nina	Gonzalez	Federico	Feo Garcma	12	Nicho	Patio Real
Baltasar	de la Riba Pinzsn	Nicomedes	Lamn Lucas	13	Pantesn	Patio de San Froilan
Jonas	Aguas Bravo	Laura	Gsmes Santos	14	Nicho	Patio de la redencisn
Marcos	Moreno Pirez	Raul	Lspez de la Pinza	22	Pantesn	La Asuncisn

Figura N° 7 Resultados Consulta 2 BDCEMENTERIO

Esto es sólo una pequeña muestra de lo que PgAccess puede hacer.



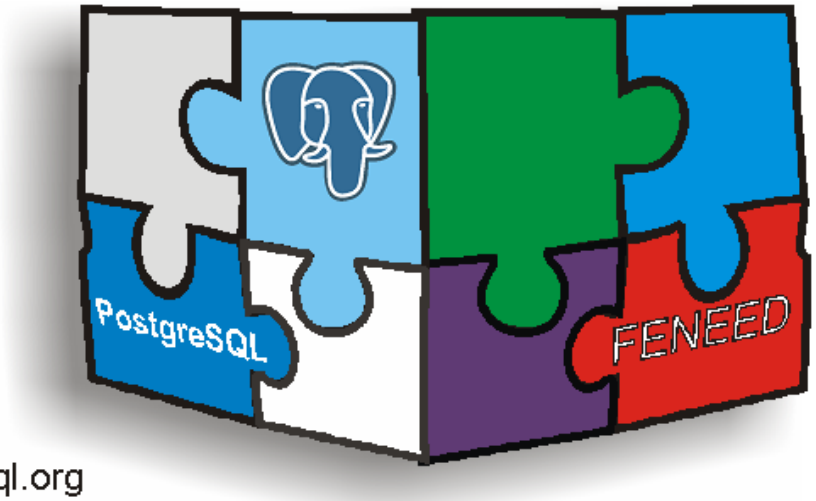
Si quiere ampliar su conocimiento en PostgreSQL lo invitamos a ver los siguientes manuales de esta serie y le recomendamos ver los siguientes portales:
http://www.postgresql.cl
http://www.varlena.com/varlena/GeneralBits/44es.php
http://www.randrade.com.mx/palm/index.php

REFERENCIA EN INTERNET

http://users.servicios.retecal.es/tjavier/docfinal/Introducci%F3n_a_PostgreSQL.html
http://www.linuxfocus.org/Castellano/May1998/article38.html
http://www.postgresql.org/docs/8.1/static/index.html

PostgreSQL 8.0.3

FENEED - Colombia
Tel 7764188
Tel 4181012



Sitio web:
Http://www.postgresql.org
E-mail: Informacion: postgresql.8.0.3@gmail.com