Introduccion a Base de Datos Con

Fundacion Codigo Libre Dominicano

Prefacio

Me he visto en la obligacion de recopilar informacion sobre la base de datos mas importante del mundo del software libre y una de las mas importantes del mundo en general, ya que no existe un manual preciso y conciso que de la información certera que mucha gente anda buscando.

Para acabar con la palabreria barata, PostGreSQL es un manejador y gestor de base de datos objeto-relacional que ha sido desarrollada de varias maneras desde 1977. Comenzo como un proyecto llamado Ingres en la Universidad de California en Berkeley. Ingres en si fue desarrollado de manera comercial por Relational Technologies / Ingres Corporation.

En 1986, otro equipo encabezado por el Dr. Michael Stonebraker de la Universidad de Berkeley, continuo con el desarrollo del codigo base de Ingres para crear un sistema gestor de base de datos usando un modelo Objeto-Relacional llamado Postgres. En 1996, debido al nuevo esfuerzo del software libre y la funcionalidad agregada de dicho sistema, Postgres fue renombrado a PostGreSQL, luego de tener por un corto tiempo el nombre de Postgres95. El proyecto PostGreSQL esta todavía bajo un muy activo desarrollo que cuenta con la participación de un equipo de desarrolladores y contribuidores a traves del mundo.

Introduccion a postgres

En éste capítulo tocaremos algunos puntos sueltos particulares a PostgreSQL. En las instalaciones basadas en RedHat estándares, PostgreSQL almacena los datos en el directorio **/var/lib/pgsql/base/** y a partir de ahí un directorio para cada base.

Cada tabla es un archivo, así como los indices. Los nombres de las tablas pertenecientes al sistema llevan el prefijo **pg_**. El archivo **PG_VERSION** (presente en cada base) contiene la versión mayor con la que fué creada la base. Al cambiar de versión de PostgreSQL es importante verificar que no haya sido cambiado el formato de alguna de estas tablas, en cuyo caso será necesario respaldarlas antes de instalar la nueva versión y luego volver a cargarlas.

Caracteristicas de PostGreSQL

Es un Gestor de datos Objeto-Relacional

Postgresql organiza los datos con un modelo obejto-relacional, es capaz de manejar procedimientos, rutinas complejas, y reglas. Ejemplos de su funcionalidad avanzada son consultas declarativas SQL, control de versiones multiples concurrentes, soporte multi-usuario, transacciones de dos fases, optimizacion de consultas, herencia de datos, arreglos y matrices multidimensionales.

Altamente Extensible

Postgresql soporta operadores, funciones, metodos de acceso y tipos de datos declarados por el usuario. Soporta también sobrecarga de operadores, sobrecarga de procedimientos, vistas materializables, y particionamiento de tablas y datos.

Soporte comprensivo de SQL

Postgresql soporta la especificación base SQL99, SQL:2003 y también incluye características avanzadas como las uniones SQL92.

Integridad Referencial

Postgresql soporta integridad referencial, la cual es usada para asegurar la validez de la información dentro de la base de datos.

API sumamente flexible

La flexibilidad del API de PostGreSQL ha permitido a entidades que provean soporte de facil desarrollo para la base de datos de PostGreSQL. Estas interfaces incluyen Object Pascal, Python, Perl, PHP, ODBC, Java/JDBC, Ruby, TCL, C/C++, Pike y un largo etcetera.

Lenguajes Procedimentales

PostGreSQL tiene soporte para lenguajes internos procedimentales, incluyendo un lenguaje nativo llamado PL/pgSQL. Este lenguaje es comparable con el lenguaje procedimental de Oracle, PL/SQL. Otra ventaja de PostGreSQL es la capacidad de usar PERL, Pitón, TCL, Ruby, R, PHP, Shell Scripting, etc.; como lenguajes procedimentales internos e integrados.

MVCC

MVCC, o control de versiones multiples concurrentes, es la tecnología que PostGreSQL usa para evitar los seguros (locking) innecesarios. Si usted ha usado cualquier otro gestor de base de datos SQL, como MySQL, MS SQL Server, Sybase ASE, etc.; se ha podido dar cuenta que existen veces cuando la lectura de un cliente tiene que esperar para accesar a la información en la base de datos. La espera es causada por los clientes que están escribiendo registros e información en la base de datos. En pocas palabras, los clientes que leen son bloqueados por los otros clientes que escriben o actualizan registros en la base de datos.

Usando MVCC, PostGreSQL evita este problema de manera definitiva. MVCC es considerado mucho mejor que los seguros a nivel de registros (row level-locking), debido a que el lector nunca es bloqueado por el escritor. En vez de eso, PostGreSQL mantiene un registro de todas las transacciones ejecutadas por los usuarios de la base de datos. PostGreSQL es entonces capaz de manejar registros sin hacer esperar a la gente a que los registros esten disponibles.

Cliente / Servidor

PostGresSQL usa una arquitectura cliente / servidor basada en un proceso por usuario. Esto es similar al proceso que utiliza el servidor web Apache para el manejo de procesos. Existe un proceso maestro que se desprende para proveer conexiones adicionales por cada cliente que se intenta conectar a PostGreSQL.

Write Ahead Logging (WAL)

La caracteristica de PostGreSQL conocida como Write Ahead Logging incrementa la confiabilidad de la base de datos registrando los cambios antes de ser escritos a la base de datos. Esto asegura que, en caso de ocurrir un fallo critico en la base de datos, existira un registro de transacciones del cual se pueda restaurar. Esto puede ser de gran beneficio en caso de cualquier fallo, como cualquier cambio que no haya sido totalmente escrito a la base de datos pueden ser recuperados usando los datos que fueron registrados anteriormente. Una vez el sistema es restaurado, el usuario puede continuar con su trabajo justo a partir del momento antes ocurrir el fallo.

Principios de Diseño de Base de Datos

Qué es una Base de Datos

En rigor, una Base de Datos es el conjunto de datos almacenados con una estructura lógica. Es decir, tan importante como los datos, es la estructura conceptual con la que se relacionan entre ellos. En la práctica, podemos pensar esto como el conjunto de datos más los programas (o software) que hacen de ellos un conjunto consistente.

Si no tenemos los dos factores unidos, no podemos hablar de una base de datos, ya que ambos combinados dan la coherencia necesaria para poder trabajar con los datos de una manera sistemática.

Definiciones

Las siguientes son las definiciones que necesitamos manejar con claridad para comprender el resto de los conceptos en las bases de datos relacionales. Algunas son definiciones dadas a la ligera, que serán extendidas y formalizadas en las secciones correspondientes.

Tupla:

Es una hilera o fila en una tabla.

Atributo:

Es una columna en una tabla.

Dominio:

Es el conjunto de valores de los cuales los atributos obtienen sus valores.

Llave:

Es un atributo con una característica de relevancia para identificar la tupla.

Llave primaria:

Es una llave con valores únicos, es decir, no ocurren más de una vez en el atributo.

Cardinalidad:

Es el número de tuplas en una tabla.

Grado:

Es el número de atributos en una tabla.

Relación:

Una definición simple es que se corresponde con una tabla y en ocasiones es preferible pensarlo de esta manera. La definición canónica es que una relación es el producto cartesiano de dos o varios dominios.

Podemos también decir que un dominio es un conjunto de valores escalares del mismo tipo, dónde un valor escalar es la mínima unidad semántica de información en el sentido de que son valores atómicos.

Tabla base:

Es una relación autónoma a diferencia de las vistas y las tablas intermedias construidas a partir de una consulta.

Vista:

Es una relación virtual, que se construye a partir de tablas base o incluso otras vistas, formada por atributos de estas otras tablas de forma directa o como resultado de una consulta.

Estructura lógica vs. estructura física.

Es claro que la forma física como estén almacenados los datos es independiente del concepto que tengamos de ellos. Son el conjunto de programas que saben como traer, unir y mostrar los datos, así como aquellos encargados de almacenarlos, los que le dan coherencia al concepto Base de Datos.

Digamos que es como la diferencia entre harina, levadura, sal y agua por separado y una pieza de pan. Quién le dá coherencia a esa pieza de pan es el proceso que se sigue para elaborarlo.

Es importante conceptualizar esto, porque del diseño de la estructura lógica depende toda la funcionalidad del sistema. Almacenar datos en una base de datos aprovechando solamente la estructura física no ofrece, relativamente, ninguna ventaja. En cambio un buen diseño de acuerdo a la naturaleza de los datos y a la forma como serán explotados hace toda la diferencia.

Un gran problema es la inconsistencia de datos. Digamos que empleamos un archivo secuencial para almacenar la información de clientes. Supongamos que tenemos varios programas que utilizan esa información y que en un momento dado se pueden tener registros duplicados con atributos diferentes. Por ejemplo, una persona cambia de dirección y al no tener una estructura bien definida, no alteramos el registro, sino que lo damos de alta de nuevo con la nueva dirección. De esta manera, tendremos a la misma persona con dos datos diferentes y sin posibilidad de garantizar que todos los programas tendrán en cuenta que la dirección válida es la del segundo registro que aparece.

Puede ocurrir también que dos personas estén modificando simultáneamente atributos distintos del mismo registro. Sin un sistema de manejo de concurrencia, no podemos garantizar que ambos cambios permanezcan.

Bajo la misma suposición de uno o más archivos con la información y varios programas independientes que la explotan, es fácil ver que cualquier nueva explotación de la información implica un nuevo programa y que mantener un sistema como este conlleva toda la complejidad de mantener varios programas cuando se añade o elimina una columna a los registros.

Otro ejemplo, si tenemos un registro de personas donde almacenamos datos como: nombre, RFC, puesto, salario base, dirección, teléfono, fecha de nacimiento, gustos musicales, aficiones, nombre, fecha de nacimiento y ocupación del cónyugue, nombres y fechas de nacimiento de sus hijos, nombres de sus mascotas, autos que posee (con todas las características), etc; y frecuentemente sólo utilizaremos nombre, RFC, puesto y salario base para generar una nómina, esto implica que en cada corrida, recuperaremos información que no necesitamos, con el agravante de que tenemos que traer registros inmensos para utilizar sólo cuatro campos. El problema no es el almacenaje en disco, sino el tiempo desperdiciado en recuperar registros de tal magnitud.

Un diseño un poco más inteligente tendría dos tablas, una con los datos más frecuentemente empleados de cada persona y la otra con el resto de la información que se emplea quizá sólo para fines estadísticos o para enviar tarjetas de felicitación. Por supuesto, ambas tablas estarán relacionadas por una llave primaria común.

Si tenemos un sistema donde por un lado hacemos un abono y por otro un cargo en una operación que está relacionada, es de esperar que no ocurra el cargo si no puede ocurrir el abono, o viceversa. Es decir, debemos de tener transacciones atómicas. En este caso, la pareja de transacciones debe de ocurrir por completo o no debe de ocurrir en lo absoluto.

Finalmente, un terrible problema es la exposición de los datos. En muchos casos nos interesa que ciertas gentes tengan acceso sólo a parte de la información. Es de mal gusto que todos los empleados sepan cuál es el salario del director.

Qué es un Manejador de Bases de Datos.

De la lectura previa podemos deducir que el Manejador de Bases de Datos (DBM por sus siglas en inglés) facilita las funciones de:

- almacenar físicamente,
- garantizar consistencia,
- garantizar integridad,
- atomicidad transaccional,
- y manejar vistas a la información.

Sistemas de archivos:

Algunos MBD implementan sus propios sistemas de archivos, manejando directamente particiones o discos completos. Esto se ha hecho principalmente para facilitar la portabilidad y hacer más eficiente esta parte. Sin embargo, PostgreSQL utiliza el sistema de archivos del sistema operativo huésped y en el caso de los derivados de Unix, esto ha mostrado ser eficiente a la vez que mantiene la portabilidad.

Índices:

El empleo adecuado de índices en una relación acelera el acceso a la información, pero consume espacio considerable, es por esto que vale la pena hacer un análisis cuidadoso de cuáles atributos requieren ser indexados.

Niveles:

Interno: cómo se almacenan y recuperan los datos (único)

Externo: cómo perciben los datos los usuarios (muchos)

Conceptual: enlace entre los anteriores

Estructuras de datos

Las estructuras de datos que se manejan en el modelo relacional corresponden a los conceptos de relación, entidad, atributo y dominio, los cuales de introducen aquí intencionalmente:

- **Relación:** Por una relación se entiende una colección o grupo de objetos que tienen en común un conjunto de características o atributos.

- **Entidad:** Es una unidad de datos en una relación con un conjunto finito de atributos. Es también conocido como n-ada, a raíz de que consiste de n-valores, uno por cada atributo.

- **Atributo:** También llamado característica, cada atributo de una relación tiene asociado un dominio en el cual toma sus valores.

- **Dominio:** Es un conjunto de valores que puede tomar un atributo en una relación.

Reglas de integridad

Los conceptos básicos de integridad en el modelo relacional son el de llave primaria, llave foránea, valores nulos y un par de reglas de integridad.

Una llave primaria es uno o un conjunto de atributos que permiten identificar a las n-adas de manera única en cualquier momento.

Una llave foránea de una relación es un atributo que hace referencia a una llave primaria de otra relación; esto da pie a que una relación pueda tener varias llaves foráneas.

Un valor nulo es un valor que está fuera de la definición de cualquier dominio el cual permite dejar el valor del atributo ``latente", su uso es frecuente en las siguientes situaciones:

i) Cuando se crea una n-ada y no se conocen todos los valores de cada uno de los atributos.
ii) Cuando se agrega un atributo a una relación ya existente.
iii) Para no tomarse en cuenta al hacer cálculos numéricos.

Las dos reglas de integridad tienen que ver precisamente con los conceptos antes mencionados y son:

- **Integridad de Relaciones:** Ningún atributo que forme parte de una llave primaria puede aceptar valores nulos.
- **Integridad Referencial:** Implica que en todo momento dichos datos sean correctos, sin repeticiones innecesarias, datos perdidos y relaciones mal resueltas.

Comandos DDL

El lenguaje SQL incluye un conjunto de comandos para definición de estructura de datos.

Create Table

El comando fundamental para definir datos es el que crea una nueva relación (una nueva tabla). La sintaxis del comando CREATE TABLE es:

```
CREATE TABLE table_name (
      name_of_attr_1 type_of_attr_1
      [, name_of_attr_2 type_of_attr_2 [, ...]]
);
```

Ejemplo de Creación de una tabla:

Para crear las tablas definidas en La Base de Datos de Proveedores y Artículos se utilizaron las siguientes instrucciones de SQL:

```
CREATE TABLE SUPPLIER (
      SNO INTEGER,
      SNAME VARCHAR(20),
      CITY VARCHAR(20)
);

CREATE TABLE PART (
      PNO INTEGER,
      PNAME VARCHAR(20),
      PRICE DECIMAL(4 , 2)
);

CREATE TABLE SELLS (
      SNO INTEGER,
      PNO INTEGER
);
```

Tipos de Datos en SQL

A continuación sigue una lista de algunos tipos de datos soportados por SQL:

- INTEGER: entero binario con signo de palabra completa (31 bits de precisión).
- SMALLINT: entero binario con signo de media palabra (15 bits de precisión).
- DECIMAL (p[,q]): número decimal con signo de p dígitos de precisión, asumiendo q a la derecha para el punto decimal. (15 _ p _ qq _ 0). Si q se omite, se asume que vale 0.
- FLOAT: numérico con signo de doble palabra y coma flotante.
- CHAR(n): cadena de caracteres de longitud fija, de longitud n.
- VARCHAR(n): cadena de caracteres de longitud variable, de longitud máxima n.

Create Index

Se utilizan los índices para acelerar el acceso a una relación. Si una relación R tiene un índice en el atributo A podremos recuperar todas la tuplas t que tienen t(A) = a en un tiempo aproximadamente proporcional al número de tales tuplas t más que en un tiempo proporcional al tamaño de R.

Para crear un índice en SQL se utiliza el comando CREATE INDEX. La sintaxis es:

```
CREATE INDEX index_name
ON table_name ( name_of_attribute );
```

Ejemplo de la creación de un indice:

Para crear un índice llamado I sobre el atributo SNAME de la relación SUPPLIER, utilizaremos la siguiente instrucción:

```
CREATE INDEX I
ON SUPPLIER (SNAME);
```

El índice creado se mantiene automáticamente. es decir, cada vez que una nueva tupla se inserte en la relación SUPPLIER, se adaptará el índice I. Nótese que el único cambio que un usuario puede percibir cuando se crea un índice es un incremento en la velocidad.

Create View

Se puede ver una vista como una tabla virtual, es decir, una tabla que no existe físicamente en la base de datos, pero aparece al usuario como si existiese. Por contra, cuando hablamos de una tabla base, hay realmente un equivalente almacenado para cada fila en la tabla en algún sitio del almacenamiento físico.

Las vistas no tienen datos almacenados propios, distinguibles y físicamente almacenados. En su lugar, el sistema almacena la definición de la vista (es decir, las reglas para acceder a las tablas base físicamente almacenadas para materializar la vista) en algún lugar de los catálogos del sistema.

En SQL se utiliza el comando CREATE VIEW para definir una vista. La sintaxis es:

```
CREATE VIEW view_name
AS select_stmt
```

donde **select_stmt** es una instrucción select válida, como se definió en Select. Nótese que **select_stmt** no se ejecuta cuando se crea la vista. Simplemente se almacena en los catálogos del sistema y se ejecuta cada vez que se realiza una consulta contra la vista.

Sea la siguiente definición de una vista (utilizamos de nuevo las tablas de La Base de Datos de Proveedores y Artículos):

```
CREATE VIEW London_Suppliers
      AS SELECT S.SNAME, P.PNAME
            FROM SUPPLIER S, PART P, SELLS SE
                  WHERE S.SNO = SE.SNO AND
                        P.PNO = SE.PNO AND
                        S.CITY = 'London';
```

Ahora podemos utilizar esta relación virtual London_Suppliers como si se tratase de otra tabla base:

```
SELECT *
FROM London_Suppliers
WHERE P.PNAME = 'Tornillos';
```

Lo cual nos devolverá la siguiente tabla:

```
SNAME | PNAME
------+-------
Smith | Tornillos
```

Para calcular este resultado, el sistema de base de datos ha realizado previamente un acceso oculto a las tablas de la base SUPPLIER, SELLS y PART. Hace esto ejecutando la consulta dada en la definición de la vista contra aquellas tablas base. Tras eso, las cualificaciones adicionales (dadas en la consulta contra la vista) se podrán aplicar para obtener la tabla resultante.

Drop Table, Drop Index, Drop View

Se utiliza el comando DROP TABLE para eliminar una tabla (incluyendo todas las tuplas almacenadas en ella):

```
DROP TABLE table_name;
```

Para eliminar la tabla SUPPLIER, utilizaremos la instrucción:

```
DROP TABLE SUPPLIER;
```

Se utiliza el comando DROP INDEX para eliminar un índice:

```
DROP INDEX index_name;
```

Finalmente, eliminaremos una vista dada utilizando el comando DROP VIEW:

```
DROP VIEW view_name;
```

Comandos DML

El lenguaje SQL incluye un conjunto de comandos para la manipulación y consulta de datos.

Select

El comando más usado en SQL es la instrucción SELECT, que se utiliza para recuperar datos. La sintaxis es:

```
SELECT [ALL|DISTINCT]
{ * | expr_1 [AS c_alias_1] [, ...]}
      FROM table_name_1 [t_alias_1]
            [, ... [, table_name_n [t_alias_n]]]
      [WHERE condition]
      [GROUP BY name_of_attr_i
          [,... [, name_of_attr_j]] [HAVING condition]]
[{UNION [ALL] | INTERSECT | EXCEPT}
SELECT ...]
      [ORDER BY name_of_attr_i [ASC|DESC]
            [, ... [, name_of_attr_j [ASC|DESC]]]];
```

Ilustraremos ahora la compleja sintaxis de la instrucción SELECT con varios ejemplos. Las tablas utilizadas para los ejemplos se definen en: *La Base de Datos de Ejemplos.*

Consultas sencillas

Aquí tenemos algunos ejemplos sencillos utilizando la instrucción SELECT:

Query sencilla con cualificación

Para recuperar todas las tuplas de la tabla PART donde el atributo PRICE es mayor que 10, formularemos la siguiente consulta:

```
SELECT * FROM PART
WHERE PRICE > 10;
```

y obtenemos la siguiente tabla:

```
PNO |  PNAME   | PRICE
----+---------+-----
  3 | Cerrojos| 15
  4 | Levas   | 25
```

Utilizando "*" en la instrucción SELECT solicitaremos todos los atributos de la tabla. Si queremos recuperar sólo los atributos PNAME y PRICE de la tabla PART utilizaremos la instrucción:

```
SELECT PNAME, PRICE
FROM PART
WHERE PRICE > 10;
```

En este caso el resultado es:

```
PNAME   | PRICE
--------+-----
Cerrojos| 15
Levas   | 25
```

Nótese que la SELECT SQL corresponde a la "proyección" en álgebra relaciona, no a la "selección".

Las cualificaciones en la clausula WHERE pueden también conectarse lógicamente utilizando las palabras claves OR, AND, y NOT:

```
SELECT PNAME, PRICE
FROM PART
WHERE PNAME = 'Cerrojos' AND
(PRICE = 0 OR PRICE < 15);
```

dará como resultado:

```
PNAME   | PRICE
--------+-----
Cerrojos| 15
```

Las operaciones aritméticas se pueden utilizar en la lista de objetivos y en la clausula WHERE. Por ejemplo, si queremos conocer cuanto cuestan si tomamos dos piezas de un artículo, podríamos utilizar la siguiente consulta:

```
SELECT PNAME, PRICE * 2 AS DOUBLE
FROM PART
WHERE PRICE * 2 < 50;
```

y obtenemos:

```
   PNAME  | DOUBLE
----------+------
Tornillos | 20
Tuercas   | 16
Cerrojos  | 30
```

Nótese que la palabra DOBLE tras la palabra clave AS es el nuevo título de la segunda columna. Esta técnica puede utilizarse para cada elemento de la lista objetivo para asignar un nuevo título a la columna resultante. Este nuevo título recibe el calificativo de "un alias". El alias no puede utilizarse en todo el resto de la consulta.

Joins (Cruces)

El siguiente ejemplo muestra como las *joins (cruces)* se realizan en SQL.

Para cruzar tres tablas SUPPLIER, PART y SELLS a través de sus atributos comunes, formularemos la siguiente instrucción:

```
SELECT S.SNAME, P.PNAME
FROM SUPPLIER S, PART P, SELLS SE
WHERE S.SNO = SE.SNO AND
P.PNO = SE.PNO;
```

y obtendremos la siguiente tabla como resultado:

```
SNAME | PNAME
-----+-----
Smith | Tornillos
Smith | Tuercas
Jones | Levas
Adams | Tornillos
Adams | Cerrojos
Blake | Tuercas
Blake | Cerrojos
Blake | Levas
```

En la clausula FROM hemos introducido un alias al nombre para cada relación porque hay atributos con nombre común (SNO y PNO) en las relaciones. Ahora podemos distinguir entre los atributos con nombre común simplificando la adicción de un prefijo al nombre del atributo con el nombre del alias seguido de un punto. La join se calcula de la misma forma, tal como se muestra en *Una Inner Join (Una Join Interna)*. Primero el producto cartesiano: SUPPLIER X PART X SELLS Ahora seleccionamos únicamente aquellas tuplas que satisfagan las condiciones dadas en la clausula WHERE es decir, los atributos con nombre común deben ser iguales). Finalmente eliminamos las columnas repetidas (S.SNAME, P.PNAME).

Operadores Agregados

SQL proporciona operadores agregados (como son AVG, COUNT, SUM, MIN, MAX) que toman el nombre de un atributo como argumento. El valor del operador agregado se calcula sobre todos los valores de la columna especificada en la tabla completa. Si se especifican grupos en la consulta, el cálculo se hace sólo sobre los valores de cada grupo.

Ejemplo de Operadores Agregados

Si queremos conocer el coste promedio de todos los artículos de la tabla PART, utilizaremos la siguiente consulta:

```
SELECT AVG(PRICE) AS AVG_PRICE
FROM PART;
```

El resultado es:

```
AVG_PRICE
-------
14.5
```

Si queremos conocer cuantos artículos se recogen en la tabla PART, utilizaremos la instrucción:

```
SELECT COUNT(PNO)
FROM PART;
```

y obtendremos:

```
COUNT
-----
4
```

Agregación por Grupos

SQL nos permite particionar las tuplas de una tabla en grupos. En estas condiciones, los operadores agregados descritos antes pueden aplicarse a los grupos (es decir, el valor del operador agregado no se calculan sobre todos los valores de la columna especificada, sino sobre todos los valores de un grupo. El operador agregado se calcula individualmente para cada grupo).

El particionamiento de las tuplas en grupos se hace utilizando las palabras clave **GROUP BY** seguidas de una lista de atributos que definen los grupos. Si tenemos **GROUP BY** $A_1, \ldots, A_k$ habremos particionado la relación en grupos, de tal modo que dos tuplas son del mismo grupo si y sólo si tienen el mismo valor en sus atributos $A_1, \ldots, A_k$.

Ejemplo de Grupos y Agregados

Si queremos conocer cuántos artículos han sido vendidos por cada proveedor formularemos la consulta:

```
SELECT S.SNO, S.SNAME, COUNT(SE.PNO)
FROM SUPPLIER S, SELLS SE
WHERE S.SNO = SE.SNO
GROUP BY S.SNO, S.SNAME;
```

y obtendremos:

```
 SNO| SNAME | COUNT
----+-------+-----
  1 | Smith | 2
  2 | Jones | 1
  3 | Adams | 2
  4 | Blake | 3
```

Demos ahora una mirada a lo que está ocurriendo aquí. Primero, la join de las tablas SUPPLIER y SELLS:

```
S.SNO | S.SNAME | SE.PNO
------+---------+-----
    1 |  Smith  | 1
    1 |  Smith  | 2
    2 |  Jones  | 4
    3 |  Adams  | 1
    3 |  Adams  | 3
    4 |  Blake  | 2
    4 |  Blake  | 3
    4 |  Blake  | 4
```

Ahora particionamos las tuplas en grupos reuniendo todas las tuplas que tiene el mismo atributo en S.SNO y S.SNAME:

```
S.SNO |  S.SNAME  | SE.PNO
------+-----------+-----
1     |  Smith     | 1
                   | 2
------------------------
2     |  Jones     | 4
------------------------
3     |  Adams     | 1
                   | 3
------------------------
4     |  Blake     | 2
                   | 3
                   | 4
```

En nuestro ejemplo, obtenemos cuatro grupos y ahora podemos aplicar el operador agregado COUNT para cada grupo, obteniendo el resultado total de la consulta dada anteriormente.

Nótese que para el resultado de una consulta utilizando GROUP BY y operadores agregados para dar sentido a los atributos agrupados, debemos primero obtener la lista objetivo. Los demás atributos que no aparecen en la clausula GROUP BY se seleccionarán utilizando una función agregada. Por otro lado, no se pueden utilizar funciones agregadas en atributos que aparecen en la clausula GROUP BY.

Having

La clausula HAVING trabaja de forma muy parecida a la clausula WHERE, y se utiliza para considerar sólo aquellos grupos que satisfagan la cualificación dada en la misma. Las expresiones permitidas en la clausula HAVING deben involucrar funcionen agregadas. Cada expresión que utilice sólo atributos planos deberá recogerse en la clausula WHERE. Por otro lado, toda expresión que involucre funciones agregadas debe aparecer en la clausula HAVING.

Ejemplo de Having

Si queremos solamente los proveedores que venden más de un artículo, utilizaremos la consulta:

```
SELECT S.SNO, S.SNAME, COUNT(SE.PNO)
FROM SUPPLIER S, SELLS SE
WHERE S.SNO = SE.SNO
GROUP BY S.SNO, S.SNAME
HAVING COUNT(SE.PNO) > 1;
```

y obtendremos:

```
SNO | SNAME | COUNT
----+-------+-----
1   | Smith | 2
3   | Adams | 2
4   | Blake | 3
```

Sub - Consultas

En las clausulas WHERE y HAVING se permite el uso de subconsultas (subselects) en cualquier lugar donde se espere un valor. En este caso, el valor debe derivar de la evaluación previa de la subconsulta. El uso de subconsultas amplía el poder expresivo de SQL.

Ejemplo de Sub Consultas

Si queremos conocer los artículos que tienen mayor precio que el artículo llamado 'Tornillos', utilizaremos la consulta:

```
SELECT *
FROM PART
WHERE PRICE > (SELECT PRICE FROM PART
WHERE PNAME='Tornillos');
```

El resultado será:

```
PNO |   PNAME   | PRICE
----+----------+-----
3   | Cerrojos | 15
4   | Levas    | 25
```

Cuando revisamos la consulta anterior, podemos ver la palabra clave SELECT dos veces. La primera al principio de la consulta – a la que nos referiremos como la SELECT externa y la segunda en la clausula WHERE, donde empieza una consulta anidada – nos referiremos a ella como la SELECT interna. Para cada tupla de la SELECT externa, la SELECT interna deberá ser evaluada. Tras cada evaluación, conoceremos el precio de la tupla llamada 'Tornillos', y podremos chequear si el precio de la tupla actual es mayor.

Si queremos conocer todos los proveedores que no venden ningún artículo (por ejemplo, para poderlos eliminar de la base de datos), utilizaremos:

```
SELECT *
FROM SUPPLIER S
WHERE NOT EXISTS
(SELECT * FROM SELLS SE
WHERE SE.SNO = S.SNO);
```

En nuestro ejemplo, obtendremos un resultado vacío, porque cada proveedor vende al menos un artículo. Nótese que utilizamos S.SNO de la SELECT externa en la clausula WHERE de la SELECT interna. Como hemos descrito antes, la subconsulta se evalúa para cada tupla de la consulta externa, es decir, el valor de S.SNO se toma siempre de la tupla actual de la SELECT externa.

Unión, Intersección, Excepción

Estas operaciones calculan la unión, la intersección y la diferencia de la teoría de conjuntos de las tuplas derivadas de dos subconsultas.

Ejemplo de Union, Intersect, Except

La siguiente consulta es un ejemplo de UNION:

```
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNAME = 'Jones'
UNION
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNAME = 'Adams';
```

Dará el resultado:

```
SNO | SNAME | CITY
 ---+-------+-----
   2 | Jones | Paris
   3 | Adams | Vienna
```

Aquí tenemos un ejemplo para INTERSECT:

```
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNO > 1
INTERSECT
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNO > 2;
```

que dará como resultado:

```
SNO | SNAME | CITY
 ---+------+-----
  2 | Jones | Paris
```

```
La única tupla devuelta por ambas partes de la consulta es la única que
tiene $SNO=2$.
```

Finalmente, un ejemplo de EXCEPT:

```
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNO > 1
EXCEPT
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNO > 3;
```

que dará como resultado:

```
SNO | SNAME | CITY
 ---+------+-----
  2 | Jones | Paris
  3 | Adams | Vienna
```

Insert Into

Una vez que se crea una tabla (vea *Create Table*), puede ser llenada con tuplas mediante el comando **INSERT INTO**. La sintaxis es:

```
INSERT INTO table_name (name_of_attr_1
   [, name_of_attr_2 [,...]])
VALUES (val_attr_1
   [, val_attr_2 [, ...]]);
```

Para insertar la primera tupla en la relación SUPPLIER (de *La Base de Datos de Proveedores y Art´culos*) utilizamos la siguiente instrucción:

```
INSERT INTO SUPPLIER (SNO, SNAME, CITY)
VALUES (1, 'Smith', 'London');
```

Para insertar la primera tupla en la relación SELLS, utilizamos:

```
INSERT INTO SELLS (SNO, PNO)
VALUES (1, 1);
```

Update

Para cambiar uno o más valores de atributos de tuplas en una relación, se utiliza el comando UPDATE. La sintaxis es:

```
UPDATE table_name
SET name_of_attr_1 = value_1
    [, ... [, name_of_attr_k = value_k]]
WHERE condition;
```

Para cambiar el valor del atributo PRICE en el artículo 'Tornillos' de la relación PART, utilizamos:

```
UPDATE PART
SET PRICE = 15
WHERE PNAME = 'Tornillos';
```

El nuevo valor del atributo PRICE de la tupla cuyo nombre es 'Tornillos' es ahora 15.

Delete

Para borrar una tupla de una tabla particular, utilizamos el comando DELETE FROM.

La sintaxis es:

```
DELETE FROM table_name
WHERE condition;
```

Para borrar el proveedor llamado 'Smith' de la tabla SUPPLIER, utilizamos la siguiente instrucción:

```
DELETE FROM SUPPLIER
WHERE SNAME = 'Smith';
```

System Catalogs

En todo sistema de base de datos SQL se emplean *catálogos de sistema* para mantener el control de qué tablas, vistas, índices, etc están definidas en la base de datos. Estos catálogos del sistema se pueden investigar como si de cualquier otra relación normal se tratase. Por ejemplo, hay un catálogo utilizado para la definición de vistas. Este catálogo almacena la consulta de la definición de la vista. Siempre que se hace una consulta contra la vista, el sistema toma primero la *consulta de definición de la vista* del catálogo y materializa la vista antes de proceder con la consulta del usuario.

SQL Embebido

En esta sección revisaremos como se puede embeber SQL en un lenguaje de host (por ejemplo C). Hay dos razones principales por las que podríamos querer utilizar SQL desde un lenguaje de host:

- Hay consultas que no se pueden formular utilizando SQL puro (por ejemplo, las consultas recursivas). Para ser capaz de realizar esas consultas necesitamos un lenguaje de host de mayor poder expresivo que SQL.

- Simplemente queremos acceder a una base de datos desde una aplicación que está escrita en el lenguaje del host (p.e. un sistema de reserva de billetes con una interface gráfica escrita en C, y la información sobre los billetes está almacenada en una base de datos que puede accederse utilizando SQL embebido).

Un programa que utiliza SQL embebido en un lenguaje de host consiste en instrucciones del lenguaje del host e instrucciones de *SQL embebido* (ESQL). Cada instrucción de ESQL empieza con las palabras claves **EXEC SQL**. Las instrucciones ESQL se transforman en instrucciones del lenguaje del host mediante un *precompilador* (que habitualmente inserta llamadas a rutinas de librerías que ejecutan los variados comandos de SQL).

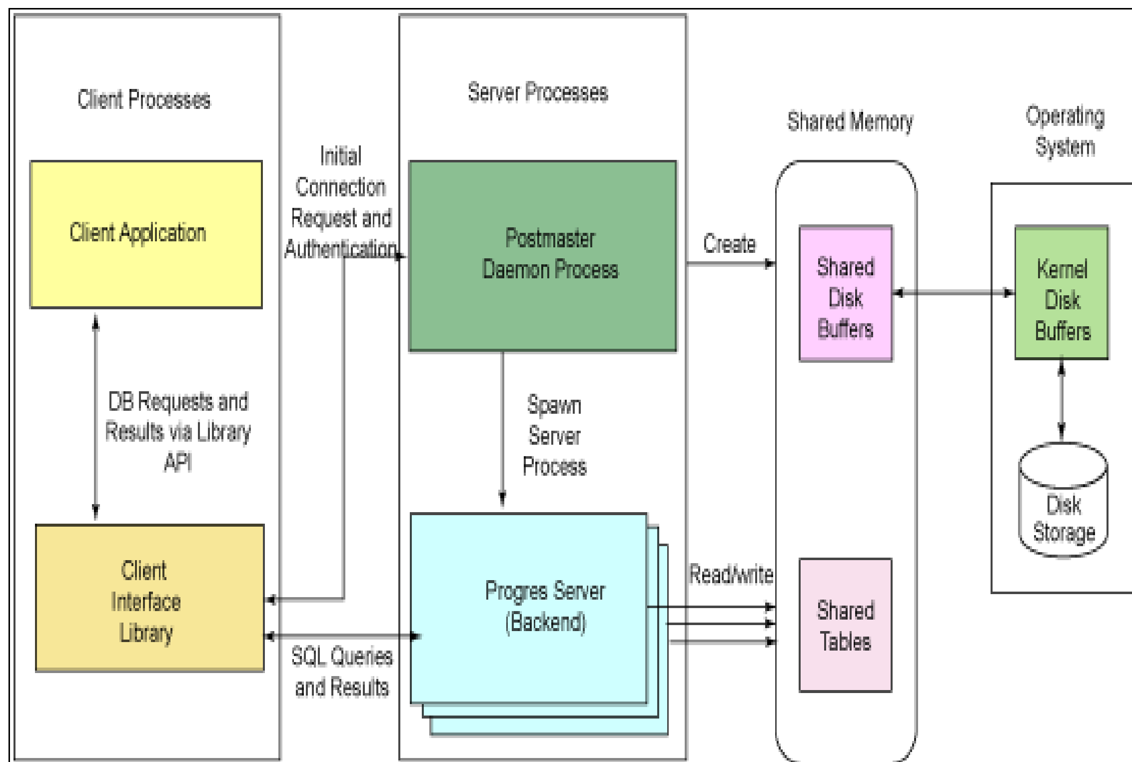
Cuando vemos los ejemplos de *Select* observamos que el resultado de las consultas es algo muy próximo a un conjunto de tuplas. La mayoría de los lenguajes de host no están diseñados para operar con conjuntos, de modo que necesitamos un mecanismo para acceder a cada tupla única del conjunto de tuplas devueltas por una instrucción SELECT. Este mecanismo puede ser proporcionado declarando un *cursor*. Tras ello, podemos utilizar el comando FETCH para recuperar una tupla y apuntar el cursor hacia la siguiente tupla.

Postgres: Conceptos de arquitectura

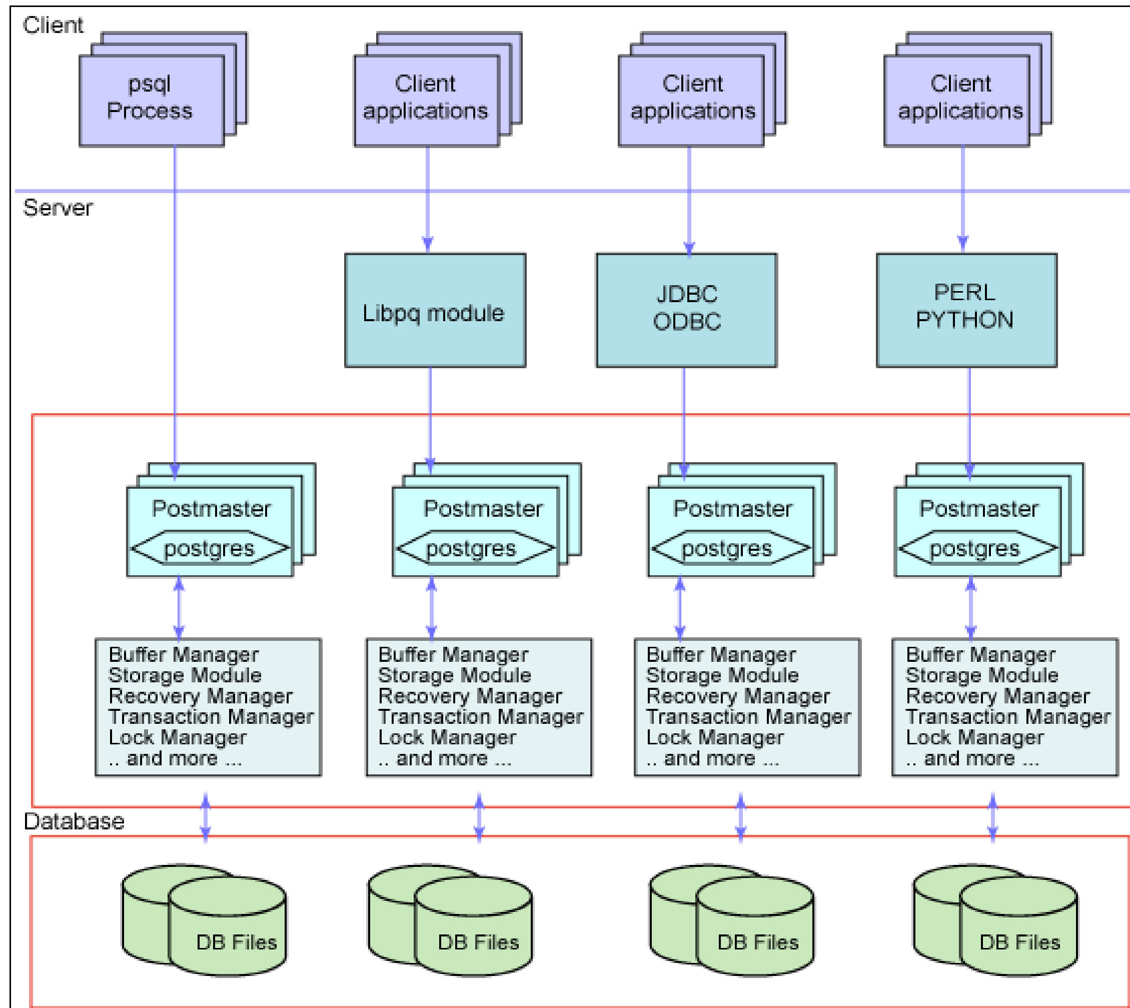
Antes de comenzar, debería comprender las bases de la arquitectura del sistema Postgres . Entendiendo como las partes de Postgres interactuan le hará el siguiente capítulo mucho más sencillo. En la jerga de bases de datos, Postgres usa un modelo cliente/sevidor conocido como "proceso por usuario". Una sesión Postgres consiste en los siguientes procesos cooperativos de Unix (programas):

- Un proceso demonio supervisor (postmaster),
- La aplicación sobre la que trabaja el usuario (frontend) (e.g., el programa psql), y
- Uno o más servidores de bases dedatos en segundo plano (el mismo proceso postgres).

Un único postmaster controla una colección de bases de datos dadas en un único host. Debido a esto una colección de bases de datos se suele llamar una instalación o un sitio. Las aplicaciones de frontend que quieren acceder a una determinada base de datos dentro de una instalación hacen llamadas a la libreria La libreria envía peticiones de usuario a través de la red al postmaster (*Como se establece una conexión*), el cual en respuesta inicia un nevo proceso en el servidor (backend)



Arquitectura resumida de postgresql 8.x



Arquitectura detalladade postgresql 8.x

y conecta el proceso de frontend al nuevo servidor. A partir de este punto, el proceso de frontend y el servidor en backend se comunican sin la intervención del postmaster.

Aunque, el postmaster siempre se está ejecutando, esperando peticiones, tanto los procesos de frontend como los de backend vienen y se van.

La libreria `libpq` permite a un único proceso en frontend realizar multiples conexiones a procesos en backend. Aunque, la aplicación frontend todavía es un proceso en un único thread. Conexiones multithread entre el frontend y el backend no están soportadas de momento en `libpq`. Una implicación de esta arquitectura es que el postmaster y el proceso backend siempre se ejecutan en la misma máquina (el servidor de base de datos), mientras que la aplicación en frontend puede ejecutarse desde cualquier sitio. Debe tener esto en mente, porque los archivos que pueden ser accedidos en la máquina el cliente pueden no ser accesibles (o sólo pueden ser accedidos usando un nombre de archivo diferente) el máquina del servidor de base de datos.

Tenga en cuenta que los servicios postmaster y postgres se ejecutan con el identificador de usuario del "superusuario" Postgres. Note que el superusuario Postgres no necesita ser un usuario especial (ej. un usuario llamado "postgres"). De todas formas, el super-usuario Postgres definitivamente no tiene que ser el superusuario de Unix ("root")! En cualquier caso, todos los archivos relacionados con la base de datos deben pertenecer a este super-usuario Postgres. Postgres.

Conceptos de Orientación a Objetos de Postgres

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

Creación de una nueva clase

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

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,          - temperatura mínima
    temp_hi         int,          - temperatura máxima
    prcp            real,         - precipitación
    date            date
);
```

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

Llenando una clase con instancias

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

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

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

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

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

Consutar a una clase

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

```
SELECT * FROM weather;
```

Y la salida sera:

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

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

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

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

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

da como resultado:

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

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

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

Redireccionamiento de consultas SELECT

Cualquier consulta select puede ser redireccionada a una nueva clase:

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

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

Joins (uniones) entre clases

Hasta ahora, nuestras consultas sólo accedían a una clase a la vez. Las consultas pueden acceder a múltiples clases a la vez, o acceder a la misma clase de tal modo que múltiples instancias de la clase sean procesadas al mismo tiempo. Una consulta que acceda a múltiples instancias de las mismas o diferentes clases a la vez se conoce como una consulta join. Como ejemplo, digamos que queremos encontrar todos los registros que están en el rango de temperaturas de otros registros. En efecto, necesitamos comparar los atributos temp_lo y temp_hi de cada instancia EMP con los atributos temp_lo y temp_hi de todas las demás instancias EMP.

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

Podemos hacer esto con la siguiente consulta:

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

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

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

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

Actualizaciones

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

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

Borrados

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

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

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

```
DELETE FROM classname;
```

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

Uso de funciones de conjunto

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

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

Como ejemplo podemos buscar la mínima temperatura en cualquier parte con

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

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

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

pero no funcionará debido a que la función max() no puede ser usada en **where**. Sin embargo, podemos replantar la consulta para llevar a cabo lo que buscamos. En este caso usando una *subseleccion*:

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

Lo que ya es correcto, ya que la subselección es una operación independiente que calcula su propia función de grupo sin importar lo que pase en el select exterior. Las funciones de grupo son también muy útiles combinándolas con cláusulas *group by.* Por ejemplo, podemos obtener la temperatura mínima tomada en cada ciudad con:

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

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

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

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

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

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

Características Avanzadas de SQL en Postgres

Habiendo cubierto los aspectos básicos de Postgre SQLpara acceder a los datos, discutiremos ahora aquellas características de Postgres que los distinguen de los gestores de bases de datos convecionales. Estas características incluyen herencia, y valores no-atómicos de datos (atributos basados en vectores y conjuntos).

Herencia

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

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

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

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

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

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

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

Que retornara:

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

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

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

Que retornara:

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

Aquí la clausula FROM ONLY obliga a que traiga los datos unica y exclusivamente de la tabla padre y no los datos de las tablas que heredan.

Valores No-Atómicos

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

Vectores

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

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

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

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

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

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

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

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

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

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


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

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

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

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

Empezando

¿Cómo empezar a trabajar con Postgres?

Algunos de los pasos necesarios para usar Postgres pueden ser realizados por cualquier usuario, y algunos los deberá realizar el administrador de la base de datos. Este administrador es la persona que instaló el software, creó los directorios de las bases de datos e inició el proceso postmaster. Esta persona no tiene que ser el superusuario Unís (“root”) o el administrador del sistema. Una persona puede instalar y usar Postgres sin tener una cuenta especial o privilegiada

Si está instalando Postgres, consulte las instrucciones de instalación en la Guía de Administración y regrese a esta guía cuando haya concluido la instalación. Mientras lee este manual, cualquier ejemplo que vea que comience con el carácter “%” o “$” son órdenes que se escribirán en el la línea de órdenes de Unix. Los ejemplos que comienzan con el caracter “*” son órdenes en el lenguaje de consulta Postgres (Postgres SQL).

Configurando el entorno

Esta sección expone la manera de configurar el entorno, para las aplicaciones. Asumimos que Postgres ha sido instalado e iniciado correctamente; consulte la Guía del Administrador y las notas de instalación si desea instalar Postgres.

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

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

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

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

en el archivo `.profile` de su directorio personal. Desde ahora, asumiremos que ha añadido el directorio bin de Postgres a su path. Además, haremos referencia frecuentemente a "configurar una variable de shell" o "configurar una variable de entorno" a lo largo de este documento. Si no entiende completamente el último párrafo al respecto de la modificación de su path, antes de continuar debería consultar los manuales de Unix que describen el shell que utiliza.

Si el administrador del sistema no tiene la configuración en el modo por defecto, tendrá que realizar trabajo extra. Por ejemplo, si la máquina servidor de bases de datos es una máquina remota, necesitará configurar la variable de entorno PGHOST con el nombre de la máquina servidor de bases de datos. También deberá especificar la variable de entorno PGPORT. Si trata de iniciar un programa de aplicación y éste notifica que no puede conectarse al postmaster, deberá consultar al administrador para asegurarse de que su entorno está configurado adecuadamente.

Ejecución del Monitor Interactivo (psql)

Asumiendo que su administrador haya ejecutado adecuadamente el proceso postmaster y le haya autorizado a utilizar la base de datos, puede comenzar a ejecutar aplicaciones como usuario. Como mencionamos previamente, debería añadir **`/usr/local/pgsql/bin`** al "path" de búsqueda de su intérprete de órdenes. En la mayoría de los casos, es lo único que tendrá que hacer en términos de preparación.

Desde Postgres v6.3, se soportan dos tipos diferentes de conexión. El administrador puede haber elegido permitir conexiones por red TCP/IP, o restringir los accesos a la base de datos a través de conexiones locales (en la misma máquina). Esta elección puede ser significativa si encuentra problemas a la hora de conectar a la base de datos.

Si obtiene los siguientes mensajes de error de una orden Postgres (tal como psql o createdb):

```
% psql template1

Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting connections
at 'UNIX Socket' on port '5432'?
```

o

```
% psql -h localhost template1

Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting TCP/IP
(with -i) connections at 'localhost' on port '5432'?
```

normalmente es debido a que (l) el postmaster no está en funcionamiento, o (2) está intentando conectar al servidor equivocado. Si obtiene el siguiente mensaje de error:

```
FATAL 1:Feb 17 23:19:55:process userid (2360) != database owner (268)
```

Significa que el administrador ejecutó el postmaster mediante el usuario equivocado. Dígale que lo reinicie utilizando el superusuario de Postgres.

Administrando una Base de datos

Ahora que Postgres está ejecutándose podemos crear alguna base de datos para experimentar con ella. Aquí describimos las órdenes básicas para administrar una base de datos.

La mayoría de las aplicaciones Postgres asumen que el nombre de la base de datos, si no se especifica, es el mismo que el de su cuenta en el sistema.

Si el administrador de bases de datos ha configurado su cuenta sin privilegios de creación de bases de datos, entonces deberán decirle el nombre de sus bases datos. Si este es el caso, entonces puede omitir la lectura de esta sección sobre creación y destrucción de bases de datos.

Creación de una base de datos

Digamos que quiere crear una base de datos llamada mydb. Puede hacerlo con la siguiente orden:

```
% createdb mydb
```

Si no cuenta con los privilegios requeridos para crear bases de datos, verá lo siguiente:

```
% createdb mydb

NOTICE:user "su nombre de usuario" is not allowed to create/destroy
databases
createdb: database creation failed on mydb.
```

Postgres le permite crear cualquier número de bases de datos en un sistema dado y automáticamente será el administrador de la base de datos que creó. Los nombres de las bases de datos deben comenzar por un carácter alfabético y están limitados a una longitud de 32 caracteres en las versiones anteriores a la v7.0. No todos los usuarios están autorizados para ser administrador de una base de datos. Si Postgres le niega la creación de bases de datos, seguramente es debido a que el administrador del sistema ha de otorgarle permisos para hacerlo. En ese caso, consulte al administrador del sistema.

Acceder a una base de datos

Una vez que ha construido una base de datos, puede acceder a ella:

- Ejecutando los programas de monitorización de Postgres (por ejemplo psql) los cuales le permiten introducir, editar y ejecutar órdenes SQL interactivamente
- Escribiendo un programa en C usando la librería de subrutinas LIBPQ, la cual le permite enviar órdenes SQL desde C y obtener mensajes de respuesta en su programa. Esta interfaz es discutida más a fondo en la *Gu´a de Programadores de PostgreSQL.*

Puede que desee ejecutar psql, para probar los ejemplos en este manual. Lo puede activar para la base de datos mydb escribiendo la orden:

```
% psql mydb
```

Se le dará la bienvenida con el siguiente mensaje:

```
Welcome to the POSTGRESQL interactive sql monitor:
Please read the file COPYRIGHT for copyright terms of POSTGRESQL

type \? for help on slash commands
type \q to quit
type \g or terminate with semicolon to execute query

You are currently connected to the database: template1
mydb=>
```

Este prompt indica que el monitor está listo y puede escribir sus consultas SQL dentro de un espacio de trabajo mantenido por el monitor. El programa psql responde a los códigos de escape que empiezan por el carácter “\”. Por ejemplo, puede obtener la ayuda acerca de la sintaxis de varias órdenes SQL Postgres escribiendo:

```
mydb=> \h
```

Una vez que haya terminado de introducir consultas, puede pasar el contenido del espacio de trabajo al servidor Postgres escribiendo:

```
mydb=> \g
```

Esto le dice al servidor que procese la consulta. Si termina su consulta con un punto y coma, la “\g” no es necesaria. psql procesará automáticamente las consultas terminadas con punto y coma. Para leer consultas desde un archivo, digamos myFile, en lugar de introducirlas interactivamente, escriba:

```
mydb=> \i nombreDelFichero
```

Para salir de psql y regresar a Unix escriba:

```
mydb=> \q
```

y psql terminará y volverá a la línea de órdenes. (Para conocer más códigos de escape, escriba **\h** en el prompt del monitor). Se pueden utilizar espacios en blanco (por ejemplo espacios, tabulador y el carácter de nueva línea) en las consultas SQL. Las líneas simples comentadas comienzan por “–”. Lo que haya después de los guiones hasta el final de línea será ignorado. Los comentarios múltiples y los que ocupan más de una línea se señalan con “/* ... */”.

Eliminando bases de datos

Si es el administrador de la base de datos mydb, puede eliminarla utilizando la siguiente orden Unix:

```
% dropdb mydb
```

Esta acción elimina físicamente todos los archivos Unix asociados a la base de datos y no pueden recuperarse, así que deberá hacerse con precaución.

Gestión de Usuarios y Grupos

Al igual que en la mayoría de los sistemas de bases de datos, la gestión de los usuarios y grupos juega un importante papel dentro de PostgreSQL. Utilizado correctamente, los usuarios y grupos tendrán un sencillo acceso a los objetos de sus bases de datos.

PostgreSQL almacena tanto los datos de usuarios como los de grupos dentro de sus propios catálogos de sistema. Estos son distintos a los definidos dentro del sistema operativo sobre el cual está instalado el software. Cualquier conexión a PostgreSQL debe ser realizada con un usuario específico, y cualquier usuario puede pertenecer a uno o más grupos definidos.

La tabla de usuarios controla los permisos de acceso y quién está autorizado a realizar acciones en el sistema (y qué acciones puede realizar). Los grupos existen como un mecanismo para simplificar la ubicación de estos permisos. Tanto las tablas de usuarios como de grupos existen como objetos globales de base de datos, lo que significa que no están adscritas a ninguna base de datos en particular.

Este capítulo le muestra la gestión y aplicación práctica de los usuarios y grupos de PostgreSQL.

Gestionando Usuarios

En orden a establecer una conexión a PostgreSQL, usted debe proporcionar una forma básica de identificación. Esta es denominada un nombre de usuario, que identifica al usuario al cual el sistema reconocerá como conectado a una base de datos. Los usuarios dentro de PostgreSQL no están necesariamente relacionados con los usuarios del sistema operativo (los cuales son también denominados cuentas del sistema), aunque usted puede escoger nominar a sus usuarios de PostgreSQL como a las cuentas de sistema a través de las que ellos accederán.

Cada usuario tiene un ID de sistema interno en PostgreSQL (llamado sysid), así como una contraseña, aunque la contraseña no es necesariamente imprescindible para conectar (dependerá de la configuración del archivo pg_hba.conf; (vea el Capítulo 8, para más detalles). El ID de sistema del usuario es utilizado para asociar objetos en una base de datos con su propietario (el usuario que está autorizado para dar/quitar privilegios sobre un objeto).

Así como son usados para asociar objetos de base de datos con su propietario, los usuarios también pueden tener permisos globales asignados a ellos cuando estos son creados. Estos permisos o privilegios determinan si un usuario podrá o no crear y eliminar bases de datos, o si el usuario es o no un superusuario (un usuario que tiene todos los permisos, en todas las bases de datos, incluyendo la capacidad de crear a otros usuarios). La asignación de estos permisos puede ser modificada en cualquier momento por cualquier superusuario.

PostgreSQL crea por defecto a un superusuario llamado postgres. Todos los demás superusuarios pueden ser creados por éste, o por cualquier otro superusurio creado posteriormente.

Viendo a los Usuarios

Toda la información relativ a los usuarios es elamacenada en una tabla de sistema de PostgreSQL llamada pg_shadow, que se muestra en la Tabla 10-1. Esta tabla es sólo seleccionable por los superusuarios, aunque una vista limitada de esta tabla, llamada pg_user, es accesible por los usuarios normales.

La tabla pg_shadow	
Columna	**Tipo**
usename	*name*
usesysid	*integer*
usecreatedb	*boolean*
usetrace	*boolean*
usesuper	*boolean*
usecatupd	*boolean*
passwd	*text*
valuntil	*abstime*

La principal diferencia entre los datos seleccionables en pg_user y pg_shadow es que el actual valor de la columna passwd no es mostrado (éste es reemplazado por asteriscos). Esta es una característica de seguridad para garantizar que los usuarios normales no están capacitados para determinar las contraseñas de los demás.

La columna usename almacena el nombre del usuario del sistema, el cual es una cadena de caracteres única (no pueden haber dos usuarios con mismo nombre, ya que los usuarios son objetos globales de base de datos). Similarmente, la columna usesysid almacena un valor entero único asociado con el usuario. Las columnas usecreatedb y usesuper respectivamente corresponden al par de privilegios que pueden ser establecidos tras la creación del usuario, tal como se documenta en la sección denominada ``Creando Usuarios''.

Creando Usuarios

PostgreSQL proporciona dos métodos para la creación de usuarios de bases de datos. Cada uno de ellos requiere autenticación como superusuario, ya que sólo los superusuarios pueden crear nuevos usuarios.

El primer método es a través del uso del comando SQL **CREATE USER**, el cual puede ser ejecutado a través de cualquier cliente PostgreSQL client (p.ej., psql). El segundo es un programa de línea de comandos llamado createuser, el cual puede ser más conveniente para un administrador de sistemas, ya que puede ser ejecutado como un simple comando sin necesidad de interactuar a través de un cliente PostgreSQL.

Las siguientes secciones documentan cada uno de estos métodos.

Creando un usuario con el comando SQL CREATE USER

El comando **CREATE USER** requiere sólo un parámetro: el nombre del nuevo usuario. También hay una variedad de opciones que pueden ser establecidas, incluyendo un contraseña, un ID de sistema explícito, grupo, y un juego de permisos que pueden ser específicamente definidos. Aquí está la sintaxis completa para **CREATE USER**:

```
CREATE USER nombre_usuario
[ WITH [ SYSID uid ]
       [ PASSWORD 'password' ] ]
[ CREATEDB | NOCREATEDB ]
[ CREATEUSER | NOCREATEUSER ]
[ IN GROUP groupname [, ...] ]
[ VALID UNTIL 'abstime' ]
```

En ésta sintaxis, nombre_usuario es el nombre del nuevo usuario que va a ser creado. Usted no puede tener dos usuarios con el mismo nombre. Mediante el uso de la palabra clave **WITH**, pueden aplicarse las palabras clave **SYSID** y **PASSWORD**.

Cada una de las otras palabras clave opcionales pueden seguir en el orden displayado (no se requiere el uso de **WITH**). Lo siguiente es una explicación detallada de cada palabra clave opcional y su significado:

- **SYSID uid**: Especifica que el ID de sistema que va a definirse debe establecerse al valor de uid. Si se omite, un razonable y único valor numérico por defecto es escogido.
- **PASSWORD 'password':** Establece la nueva contraseña del usuario a password. Si no se especifica, la contraseña por defecto es **NULL.**
- **CREATEDB | NOCREATEDB:** Usando la palabra clave **CREATEDB** se le garantiza al nuevo usuario el privilegio de crear nuevas bases de datos, así como el de destruir las de su propiedad. Usando **NOCREATEDB** se deniega este permiso (que es lo que ocurre por defecto).
- **CREATEUSER | NOCREATEUSER:** Garantiza el privilegio de crear nuevos usuarios, lo cual implícitamente crea a un superusuario. Advierta que un usuario con los privilegios de crear a otros usuarios tendrá todos los privilegios, en todas las bases de datos (incluyendo los permisos para crear una base dedatos, aunque se haya especificado **NOCREATEDB**). **NOCREATEUSER** explícitamente fuerza a la situación por defecto, que deniega el privilegio.
- **IN GROUP nombre_grupo [, ...]:** Añade al nuevo usuario al grupo llamado nombre_grupo. Pueden ser especificados múltiples nombres de grupo, separándolos mediante comas. El/los grupos deben existir para que funcione el estamento.
- **VALID UNTIL 'abstime':** Establece que la contraseña del usuario expirará el abstime, el cual debe ser un formato reconocible de fecha/hora (timestamp). Tras esa fecha, la contraseña se resetea, y la expiración se hace efectiva.
- **VALID UNTIL 'infinity':** Establece validez permanente para la contraseña del usuario.

Si no se especifica CREATEDB o CREATEUSER, los usuarios son implícitamente ``normales'', sin privilegios especiales. No pueden crear bases de datos u otros usuarios, ni pueden eliminar bases de datos o usuarios. Estos usuarios pueden conectar a bases de datos en PostgreSQL, pero sólo pueden ejecutar estamentos para los que han sido autorizados (vea la sección nominada ``Otorgando Privilegios'' para más información).

El Siguiente Ejemplo crea a un usuario normal llamado salesuser. También establece una contraseña de N0rm4! mediante el uso de la cláusulaby the use of the WITH PASSWORD. Omitiendo la cláusula VALID UNTIL, esta contraseña nunca expirará.

Ejemplo de Creacion de un usuario normal.

```
template1=# CREATE USER salesuser
template1-# WITH PASSWORD 'N0rm4!';
CREATE USER
```

El mensaje del servidor CREATE USER retornado en el Ejemplo anterior indica que el usuario fue creado correctamente. Otros mensajes que usted puede recibir son los siguientes:

```
ERROR: CREATE USER: permission denied
```

Este mensaje es retornado si el usuario que utilizó el comando CREATE USER no es un superusuario. Sólo los superusuarios pueden crear nuevos usuarios.

```
ERROR: CREATE USER: user name "salesuser" already exists
```

Este mensaje indica que un usuario con el nombre salesuser ya existe.

Si desea crear un usuario con la capacidad de crear bases de datos en PostgreSQL pero que no pueda crear o eliminar a usuarios PostgreSQL, puede especificar la palabra clave CREATEDB en vez de CREATEUSER. Esto permite que el usuario pueda crear bases de datos arbitrariamente, así como eliminar cualquier base de datos de la que él sea propietario. Para más información sobre la creación y eliminación de bases de datos, consulte los topicos anteriores de este manual.

El proximo Ejemplo ilustra la creación de un usuario llamado dbuser quien tiene permisos para crear nuevas bases de datos. Esto se hace mediante la especificación de la palabra clave CREATEDB tras el nombre de usuario. Advierta también el uso de las palabras clave WITH PASSWORD y VALID UNTIL. Estas establecen la contraseña para dbuser a DbuS3r, la cual será válida hasta el 11 de Noviembre de 2002.

Ejemplo de la creacion un usuario con permisos CREATEDB

```
template1=# CREATE USER dbuser CREATEDB
template1-# WITH PASSWORD 'DbuS3r'
template1-# VALID UNTIL '2002-11-11';
CREATE USER
```

El hecho de resetear una contraseña de usuario expirada no modifica el valor de VALID UNTIL. En orden a reactivar el acceso de un usuario cuya contraseña ha expirado, tanto la palabra clave WITH PASSWORD y VALID UNTIL deben ser proporcionadas al comando ALTER USER. Vea la sección denominada "Alterando Usuarios" para más información.

Advertencia: Los valores en VALID UNTIL sólo son relevantes en sistemas que no son de confianza; los sitios de confianza no requieren contraseñas. Vea el Capítulo 8 para más información sobre autenticación basada en máquinas.

Puede desear crear un superusuario alternativo al usuario postgres, aunque debería tener cuidado a la hora de crear superusuarios. Estos usuarios tienen garantizados todos los privilegios dentro de PostgreSQL, incluyendo la creación de usuarios, eliminación de usuarios, y eliminación de bases de datos. El Ejemplo 10-3 demuestra la creación de un superusuario PostgreSQL llamado manager desde el prompt de psql.

Ejemplo de la Creación de un superusuario

```
template1=# CREATE USER manager CREATEUSER;
CREATE USER
```

Creando un usuario con el script createuser

El script createuser es ejecutado directamente desde la línea de comandos, y puede operar de dos formas. Si se utiliza sin argumentos, él interactivamente le pedirá el nombre de usuario y cada uno de los privilegios que se le van a asignar, e intentará realizar una conexión local a PostgreSQL. Alternativamente, puede optar por especificar las opciones y el nombre del usuario a ser creado en la misma línea de comandos.

Al igual que con otras aplicaciones de línea de comandos para PostgreSQL, los argumentos pueden ser suministrados en formato corto (con un único guión, y un carácter), o en su formato largo (con dos guiones, y el nombre completo del argumento).

Aquí tiene la sintaxis de createuser:

```
createuser [ opciones ] [ nombre_usuario ]
```

El nombre_usuario en la sintaxis representa el nombre del usuario que va a crear. Reemplace opciones con una o más de las siguientes:

- **-d, -createdb:** Equivalente a la palabra clave CREATEDB del comando SQL CREATE USER. Permite al nuevo usuario crear bases de datos.
- **-D, -no-createdb:** Equivalente a la palabra clave NOCREATEDB del comando SQL CREATE USER. Explícitamente indica que el nuevo usuario no puede crear bases de datos. Esta es la situación por defecto.
- **-a, -adduser:** Equivalente a la palabra clave CREATEUSER del comando SQL CREATE USER. Perimte al nuevo usuario la creación de otros usuarios, y asigna el status de superusurario al usuario (activando todos los privilegios dentro de PostgreSQL).
- **-A, -no-adduser:** Equivalente a la palabra clave NOCREATEUSER del comando SQL CREATE USER. Explícitamente indica que el nuevo usuario no es superusuario. Esta es la situación por defecto.
- **-i SYSID, -sysid=SYSID:** Establece el nuevo ID de sistema del usuario a SYSID.
- **-P, -pwprompt:** Resulta en una petición de introducción de contraseña, permitiéndole establecer la contraseña del nuevo usuario.
- **-h NOMBRE_MAQUINA, -host=NOMBRE_MAQUINA:** Especifica desde qué NOMBRE_MAQUINA se conectará, además de la local (localhost), o la máquina definida por la variable de entorno PGHOST.
- **-p PUERTO, -port=PUERTO:** Especifica que la conexión de base de datos se realizará por el puerto PUERTO, en vez de por el puerto por defecto (usualmente el 5432).

- **-U NOMBRE_USUARIO, -username=NOMBRE_USUARIO:** Especifica que NOMBRE_USUARIO será el usuario que conecte a PostgreSQL (por defecto se conecta usando el nombre de usuario del sistema).

- **-W, -password:** Resulta en una petición de contraseña para el usuario que conecta, lo cual ocurre automáticamente si el archivo pg_hba.conf está configurado para no confiar en la máquina solicitante.

- **-e, -echo:** Causa que el comando CREATE USER envíe a PostgreSQL para ser displayado todo lo que se ejecute con createuser.

- **-q, -quiet:** Previene que la salida sea enviada a stdout (aunque los errores serán enviados a stderr).

Si alguno de los argumentos -d, -D, -a, -A, o nombre_usuario son omitidos, createuser le pedirá la introducción de cada uno de ellos. Esto se debe a que PostgreSQL no realizará ninguna asunción sobre los privilegios que se deben asignar al nuevo usuario, ni sobre el nombre del mismo. El Ejemplo 10-4 crea a un usuario llamado newuser, el cual no tiene permisos ni para crear bases de datos ni para crear nuevos usuarios.

Ejemplo de la creacion de un usuario con createuser

```
[jworsley@booktown ~]$ createuser -U manager -D -A newuser
CREATE USER
```

Advierta el flag "-U manager" pasado al script createuser. Este indica que el usuario que el nombre de usuario con el que se realizará la conexión a PostgreSQL es manager, y no jworsley como el script podría haber asumido, basándose en el nombre de la cuenta de sistema que ha invokado al script.

Si usted prefiere que de forma interactiva se le vaya preguntando por cada parámetro de configuración (en vez de tener que recordar el significado de cada flag), puede simplemente omitirlos. El script createuser le preguntará por las opciones básicas de createuser. Estas opciones incluyen el nombre de usuario PostgreSQL, si el usuario podrá crear bases de datos, y si el usuario podrá o no crear nuevos usuarios para PostgreSQL.

El Ejemplo siguiente demuestra el uso del script createuser en modo interactivo. El efecto de este ejemplo es el mismo que el ejectuado en una sóla línea en el Ejemplo 10-4.

Ejemplo de la creación interactiva con createuser

```
[jworsley@booktown ~]$ createuser
Enter name of user to add: newuser
Shall the new user be allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
```

Alterando (Modificando) Usuarios

Los usuarios existentes sólo pueden ser modificados por superusuarios PostgreSQL. Las posibles modificaciones incluyen cada una de las opciones disponibles a la hora de la creación del usuario (p.ej., contraseña, fecha de expiración de la contraseña, permisos globales), excepto el ID de sistema para un usuario existente, que no puede ser modificado. La modificación de usuarios existentes se realiza mediente el uso del estamento SQL ALTER USER.

Aquí tiene la sintaxis de ALTER USER:

```
ALTER USER nombre_usuario
      [ WITH PASSWORD 'password' ]
      [ CREATEDB | NOCREATEDB ]
      [ CREATEUSER | NOCREATEUSER ]
      [ VALID UNTIL 'abstime' ]
```

El argumento requerido nombre_usuario especifica el usuario a ser modificado. Cualquiera de los siguientes parámetros puede ser especificado adicionalmente:

- **WITH PASSWORD 'password':** Establece la contraseña del usuario a password.
- **CREATEDB | NOCREATEDB:** Garantiza o revoka al usuario el privilegio de crear bases de datos.
- **CREATEUSER | NOCREATEUSER:** Garantiza o revoak al usuario el status de superusuario, el cual le activa todos los privilagios dentro de PostgreSQL (el más notable, la capacidad de crear y destruir usuarios y superusuarios).
- **VALID UNTIL 'abstime':** Establece que la contraseña del usuario expirará el abstime, el cual debe ser algún formato válido timestamp. Este valor sólo es relevante para sistemas que requieran autentificación mediante contraseña, ya que de lo contrario es ignorado (p.ej., para sitios de confianza).

Una función común de ALTER USER es resetear la contraseña (y potencialmente la fecha de expiración) de un usuario. Si un usuario PostgreSQL tiene una fecha de expiración establecida y la fecha ha pasado, y el usuario necesita autentificación basada en contraseña, un superusuario tendrá que resetear tanto la contraseña como la fecha de expiración para reactivar la capacidad de conexión del usuario. Si desea que nunca expire una contraseña de usuario, esteblezca la fecha especial infinity.

El Ejemplo siguiente modifica a un usuario llamado salesuser. La contraseña del usuario está configurada a n3Wp4s4 por la cláusula WITH PASSWORD, y expira el 1 de Enero del 2003, por la cláusula VALID UNTIL.

Ejemplo de re-establecer una contraseña

```
template1=# ALTER USER salesuser
template1-# WITH PASSWORD 'n3WP4s4'
template1-# VALID UNTIL '2003-01-01';
ALTER USER
```

En alguna ocasión puede desear o necesitar otorgar a un usuario privilegios adicionales que originalmente no tenía. El uso de CREATEUSER en el proximo ejemplo modifica los del usuario salesuser y le otorga todos los privilegios en PostgreSQL, convirtiendo al usuario en superusuario. Advierta que esto activa al CREATEDB, ya que los superusuarios pueden crear bases de datos implícitamente.

Ejemplo de añadir privilegios de superusuario

```
template1=# ALTER USER salesuser
template1-# CREATEUSER;
ALTER USER
```

De forma contraria, pueden existir ocasiones en las que un usuario no debe ostentar por más tiempo determinados privilegios. Estos pueden ser fácilmente eliminados por un superusuario con las palabras clave NOCREATEDB y NOCREATEUSER.

Ejemplo de eliminar privilegios de superusuario

```
template1=# ALTER USER salesuser
template1-# NOCREATEDB NOCREATEUSER;
ALTER USER
```

Advertencia: Como cualquier superusuario puede revocar privilegios a cualquier superusuario, o incluso eliminarlo, debe ser extremadamente cuidadoso cuando asigne el privilegio CREATEUSER.

Eliminando Usuarios

Los usuarios de PostgreSQL pueden ser eliminados en cualquier momento del sistema por los superusuarios. La única restricción es que un usuario no puede ser eliminado si existen bases de datos de su propiedad. Si un usuario es propietario de una base de datos, esa base de datos debe ser eliminada antes de poder eliminar al usuario.

Al igual que con la creación de usuarios en PostgreSQL, hay dos métodos mediante los cuales los usuarios pueden ser eliminados. Estos son el comando SQL DROP USER, y el ejecutable en línea de comandos dropuser

Eliminando usuarios mediante el comando SQL DROP USER

Un superusuario puede eliminar a un usuario usando el comando DROP USER desde cualquier cliente PostgreSQL válido. El programa psql es el más comúnmente utilizado para este tipo de tareas.

Aquí tiene la sintaxios para DROP USER:

```
DROP USER nombre_usuario
```

En ésta sintaxis, nombre_usuario es el nombre del usuario que va a ser permanentemente eliminado del sistema. El Ejemplo 10-9 muestra el uso del cliente psql para conectar a PostgreSQL como el usuario manager en orde a eliminar al usuario salesuser.

Ejemplo de eliminar a un usuario con DROP USER

```
[jworsley@booktown ~]$ psql -U manager template1
Welcome to psql, the PostgreSQL interactive terminal.

Type: \copyright for distribution terms
      \h for help with SQL commands
      \? for help on internal slash commands
      \g or terminate with semicolon to execute query
      \q to quit

template1=# DROP USER salesuser;
DROP USER
```

El mensaje del servidor DROP USER indica que el usuario fue eliminado con éxito del sistema. Otros mensajes que usted puede recibir incluyen:

```
ERROR: DROP USER: permission denied
```

Indica que el usuario que inició el comando no tenía privilegios para eliminar a un usuario. Sólo los superusuarios pueden eliminar usuarios de bases de datos existentes.

```
ERROR: DROP USER: user "salesuser" does not exist
```

Indica que no existe un usuario con el nombre salesuser.

Eliminando usuarios mediante el comando del sistema dropuser

El comando dropuser opera igual que el script createuser. Ofrece las mismas opciones de conexión, garantizando que puede ser usado remotamente tanto como localmente, y requiere sólo el nombre de usuario del usaurio que va a ser eliminado del sistema.

Aquí tiene la sintaxis de dropuser:

```
dropuser [ options ] [ username ]
```

Cada una de las opciones de conectividad son idénticas a las de createuser, descritas en la sección denominada "Creando un usuario con el script createuser", que ya vimos anteriormente. El Ejemplo 10-10 demuestra el mismo efecto que el estamento SQL del Ejemplo 10-9 conectando al motor PostgreSQL como el usuario manager, y eliminando al usuario denominado salesuser.

Ejemplo de Eliminar a un usuario con dropuser

```
[jworsley@booktown ~]$ dropuser -U manager salesuser
DROP USER
```

La salida de dropuser es la misma que la del comando SQL DROP USER. Si usted omite el nombre de usuario a eliminar, será interrogado interactivamente por el nombre del usuario a ser eliminado del sistema.

Manejando Grupos

Los grupos sirven para simplificar la asignación de privilegios. Los privilegios ordinarios debern ser asignados a un único usuario, uno cada vez. Esto puede resultar tedioso si los usuarios a los que hay que asignar los mismos accesos a una gran variedad de objetos de base de datos.

Los grupos son creados para evitar este problema. Un grupo requiere un nombre, y puede ser creado vacío (sin usuarios). Una vez creado, los usuarios que se pretende compartan permisos comunes son añadidos todos al grupo, y quedan asociados al grupo por su número de miembro. Los permisos en los objetos de base de datos son entonces asignados al grupo, en vez de a cada uno de los miembros del grupo. Para un sistema con muchos usuarios y bases de datos, los grupos haccen que la asignación de permisos sea más cómoda para el administrador.

Nota: Los usuarios pueden pertenecer a cualquier número de grupos, o a ninguno.

Creando y Eliminando Grupos

Antes de que se inicie con la gestión de grupos, debería antes aprender a crearlos y eliminarlos del sistema. Cada uno de estos procedimientos requiere privilegios de superusuario. Vea la sección llamada "*Manejando Usuarios*", que ya vimos anteriormente, para aprender más sobre superusuarios.

Creando un Grupo

Cualquier superusuario puede crear un nuevo grupo en PostgreSQL con el comando CREATE GROUP. Aquí tiene la sintaxis de CREATE GROUP:

```
CREATE GROUP nombre_grupo
       [ WITH [ SYSID groupid ]
       [ USER username [, ...] ] ]
```

En ésta sintaxis, nombre_grupo es el nombre del grupo a ser creado. Un nombre de grupo debe comenzar por una carácter alfabético, y no puede superar los 31 caracteres de longitud. El proporcionar la palabra clave WITH palabra clave le permite especificar cualquiera de los atributos opcionales. Si desea especificar el ID de sistema a usar con el nuevo grupo, utilice la palabra clave SYSID para especificar el valor de groupid. Use la palabra clave USER para incluir a uno o más usuarios al grupo en tiempo de creación. Separe los distintos usuarios mediante comas.

Adicionalmente, el usuario PostgreSQL y las tablas de grupos operan separadamente las unas de las otras. Esta separación que los ID de usuarios y grupos puedan ser idénticos dentro del sistema PostgreSQL.

Como ejemplo, el siguiente crea el grupo sales, y añade dos usuarios al grupo en tiempo de creación. Estos usuarios son allen y vincent.

Ejemplo de crear un grupo

```
booktown=# CREATE GROUP sales
booktown-# WITH USER allen, vincent;
CREATE GROUP
```

El mensaje del servidor CREATE GROUP indica que el grupo fue creado con éxito. Usted puede verificar la creación del grupo, así como ver todos los grupos existentes, con una consulta a la tabla de sistema pg_group. El Ejemplo siguiente ejecuta dicha consulta.

Ejemplo de verificar un grupo

```
booktown=# SELECT * FROM pg_group;
    groname | grosysid | grolist
------------+----------+-------------
      sales | 1        | {7017,7016}
 accounting | 2        |
  marketing | 3        |
(3 rows)
```

Advierta que la columna grolist es un array, que contiene el ID de usuario de PostgreSQL de cada usuario en el grupo. Estos son los mismos IDs de usuario que vimos en la vista de pg_user. Por ejemplo:

```
booktown=# SELECT usename FROM pg_user
booktown-# WHERE usesysid = 7017 OR usesysid = 7016;

usename
---------
allen
vincent
(2 rows)
```

Eliminando un grupo

Cualquier superusuario puede también eliminar un grupo con el comando SQL DROP GROUP. Debería ejecutar con cuidado éste comando, ya que es irreversible, y no será interrogado para que verifique la orden de eliminación del grupo (aunque todavía haya usuarios en el grupo). A diferencia de DROP DATABASE, DROP GROUP puede ser realizado dentro de un bloque de transacciones.

Aquí tiene la sintaxis de DROP GROUP:

```
DROP GROUP nombre_grupo
```

El nombre_grupo es el nombre del grupo que va a ser eliminado permanentemente. El Ejemplo siguiente elimina el grupo marketing de la base de datos Book Town.

Ejemplo de la eliminacion un grupo

```
booktown=# DROP GROUP marketing;
DROP GROUP
```

El mensaje del servidor DROP GROUP retornado en el Ejemplo anterior indica que el grupo fue eliminado con éxito. Advierta que la eliminación de un grupo no elimina los permisos asignados al mismo, sino que le libera de los mismos. Cualesquier permisos asignados a un objeto de base de datos que tiene los permisos asignados a un grupo eliminado aparecerán asignados a un ID de grupo, en vez de a un grupo.

Nota: Los grupos eliminados por error pueden ser restaurados creando un nuevo grupo con el mismo ID de sistema que el eliminado. Esto implica a la palabra clave SYSID, tal como la documentamos en la sección llamada ``Creando un grupo''. Si usted asigna permisos de grupo a una tabla y luego elimina el grupo, los permisos del grupo en la tabla serán mantenidos. Sin embargo, necesitará añadir a los adecuados usuarios al recientemente creado grupo para que los permisos de tabla sean efectivos para los miembros de ese grupo.

Asociando usuarios con grupos

Los usuarios pueden ser añadidos y eliminados de los grupos en PostgreSQL a través del comando SQL ALTER GROUP. Aquí tiene la sintaxis del comando ALTER GROUP:

```
ALTER GROUP nombre_grupo { ADD | DROP } USER username [, ... ]
```

El nombre_grupo es el nombre del grupo a ser modificado, mientras que nombre_usuario es el nombre del usuario a ser añadido o eliminado, dependiendo del uso de las palabras clave ADD o DROP.

Añadiendo un usuario a un grupo

Suponga que Booktown contrata a dos nuevos asociados de ventas, David y Ben, y les asigna los nombres de usuario david y ben, respectivamente. El Ejemplo 10-14 que usa el comando ALTER GROUP añade estos nuevos usuarios al grupo sales.

Ejemplo de añadir un usuario a un grupo

```
booktown=# ALTER GROUP sales ADD USER david, ben;
ALTER GROUP
```

El mensaje del servidor ALTER GROUP indica que los usuarios david y ben fueron añadidos con éxito al grupo sales. El Ejemplo anterior demuestra otra consulta a la tabla pg_group para verificar la adición de esos nuevos usuarios al grupo. Advierta que ahora hay cuatro IDs de sistema en la columna grolist para el grupo sales .

Ejemplo de Verificar la adición de usuarios

```
booktown=# SELECT * FROM pg_group WHERE groname = 'sales';
 groname | grosysid | grolist
---------+----------+-----------------------
  sales  | 1        | {7019,7018,7017,7016}
(1 row)
```

Eliminando a un Usuario de un Grupo

Suponga que durante cierto tiempo el usuario David es transferido desde el departamento sales al departamento accounting. En orden a mantener la correcta asociación de grupo, y de asegurarse de que David no tiene privilegios exclusivos en el grupo sales, su nombre de usuario (david) debería ser eliminado del grupo; el Ejemplo siguiente lo muestra.

Ejemplo de eliminar a un usuario de un grupo

```
booktown=# ALTER GROUP sales DROP USER david;
ALTER GROUP
```

El mensaje ALTER GROUP retornado en el Ejemplo anterior indica que el usuario david fue eliminado con éxito del grupo sales.

Para completar su transición al departamento accounting, David debe tener su nombre de usuario añadido al grupo accounting. Los siguientes estamentos usan una sintaxis similar a la de los estamentos de los dos ejemplos del topico. El efecto es que el usuario david es añadido al grupo accounting. Esto significa que cualesquiera privilegios especiales otorgados a este grupo serán implícitamente asignados a david durante el tiempo que pertenezca al grupo.

```
booktown=# ALTER GROUP accounting ADD USER david;
ALTER GROUP


booktown=# SELECT * FROM pg_group;
groname     | grosysid | grolist
------------+----------+------------------
sales       | 1        | {7016,7017,7019}
accounting  | 2        | {7018}
(2 rows)
```

Otorgando Privilegios

PostgreSQL mantiene un altamente controlado juego de listas de control de acceso, o ACLs. Esta información describe qué usuarios están autorizados para realizar consultas, actualizaciones, etc. o modificar objetos dentro de una base de datos. Existe un juego de privilegios de acceso y de restricciones aplicable para cada objeto de base de datos en PostgreSQL (p.ej., tablas, vistas y secuencias). Los superusuarios y los propietarios de los objetos de bases de datos mantienen estos ACLs a través de un par de comandos SQL: GRANT y REVOKE.

Como se vió en el Capítulo 9, cuando un usuario crea por primera vez una base de datos, se vuelve implícitamente el propietario de esa base de datos. Similarmente, cada vez que alguien crea u objeto de base de datos, este es poseído por el individuo que ejecutó el correspondiente comando SQL CREATE.

Además de los superusuarios de PostgreSQL (que pueden manipular cualquier objeto de base de datos de cualquier forma), sólo los propietarios de los objetos de base de datos pueden dar y/o revocar privilegios sobre los objetos de su propiedad. Aunque cualquier usuario pueda conectar a una base de datos, si desea acceso a objetos dentro de la base de datos debe tener los correspondientes privilegios explícitamente otorgados.

Comprendiendo el Control de Acceso

Como ya mencionamos antes en esta sección, las listas de control de acceso se aplican a tres tipos de objetos de base de datos: tablas, listas y secuencias. Para estos objetos, hay cuatro privilegios generales que pueden ser otorgados, o revocados, a un usuario o grupo. Los usuarios y/o grupos no poseen privilegios por defecto.

Desde el cliente psql, usted puede ver el sumario de permisos de las ACL usando el comando rápido \z. Este comando displaya todos los permisos de acceso sobre la base de datos actualmente accedida. Para ver los permisos de un objeto específico, indique el nombre del objeto como parámetro del comando \z. Puede usar una expresión regular en lugar de un nombre para ver los privilegios de un grupo de objetos.

La tabla siguiente lista cada uno de los privilegios de control de acceso disponibles en PostgreSQL. Cada privilegio también tiene un símbolo asociado, el cual aparece como un carácter alfabético único. Estos símbolos son la abreviatura del privilegio descrito, y son usados por el comando \z de psql cuando se displayan sumarios o permisos de acceso.

Privilegios ACL en PostgreSQL		
P.Clave	Símbolo	Descripción
SELECT	*r*	Permite a un usuario obtener datos de una tabla, vista o secuencia (aunque la función netxval() no puede ser llamada sólo con privilegios SELECT). También conocido como permisos de “lectura”.
INSERT	*a*	Permite a un usuario insertar nuevas filas en una tabla. También conocido como permisos de “adición”.
UPDATE, DELETE	*w*	Permite a un usuario modificar o eliminar filas de datos de una tabla. Si sólo se asigna uno de los privilegios, el otro es implícitamente otorgado. También conocidos como permisos de “escritura”.
RULE	*R*	Permite a un usuario crear una regla de reescritura sobre una tabla o vista.
ALL	*arwR*	Representa una forma corta de garantizar o revocar todos los permisos de una sola vez. ALL no es un privilegio en sí mismo. Agisnar ALL implica garantizar los permisos SELECT, INSERT, UPDATE, DELETE, y RULE.

Garantizando Privilegios con GRANT

Para asignar un privilegio a un usuario o a un grupo, use el comando SQL GRANT. Aquí tiene la sintaxis de GRANT:

```
GRANT privilegio [, ...] ON objecto [, ...]
      TO { PUBLIC | nombre_usuario | GROUP nombre_grupo }
```

En esta sintaxis, privilegio es cualquiera de los privilegios listados en la tabla anterior, objecto es el nombre del objeto de base de datos (tabla, vista o secuencia) para el que es asignado el privilegio, y el elemento que sigue a la palabra clave TO describe a quién es garantizado el privilegio. Se pueden indicar múltiples privilegios y objetos, separados los unos de los otros mediante comas.

Sólo uno de los términos a continuación de TO pueden ser usados en un único estamento GRANT. El otorgamiento de privilegios con la palabra clave PUBLIC indiscriminadamente garantiza el privilegio al objetivo especial "public". Los privilegios PUBLIC son compartidos por todos los usuarios. Especificar un nombre de usuario garantiza el privilegio al usuario especificado. Por el contrario, especificar un nombre de grupo garantiza el privilegio al grupo especificado.

Supongamos, por ejemplo, que el usuario manager necesita todos los permisos para las tablas customers, books, editions y publishers. El Ejemplo 10-17 da al usuario manager dichos privilegios, con un único estamento GRANT.

Ejemplo de otorgar privilegios de usuario.

```
booktown=# GRANT ALL ON customers, books, editions, publishers
booktown-# TO manager;
CHANGE
```

El uso de la palabra clave ALL en el Ejemplo 10-17 otorga todos los posibles permisos ACL (SELECT, UPDATE, etc.) para los objetos especificados al usuario manager. El mensaje CHANGE del servidor indica que los privilegios fueron modificados con éxito. Recuerde que puede usar el comando rápido \z en psql para verificar los permisos establecidos sobre un objeto de base de datos.

```
booktown=# \z publishers
Access permissions for database "booktown"
Relation    | Access permissions
------------+----------------------
publishers  | {"=","manager=arwR"}
(1 row)
```

Como otro ejemplo, veamos el uso de la palabra clave GROUP para otorgar privilegios a los miembros de un grupo. Por ejemplo, todo el departamento de ventas (sales) Book Town debería tener permisos para ver la tabla customers, pero no para modificarla. El Ejemplo 10-18 otorga privilegios SELECT sobre la tabla customers a cualquier miembro del grupo sales.

Ejemplo de otorgar privilegios de grupo.

```
booktown=# GRANT SELECT ON customers TO GROUP sales;
CHANGE

booktown=# \z customers
Access permissions for database "booktown"
Relation   | Access permissions
-----------+----------------------------------
customers  | {"=","manager=arwR","group sales=r"}
(1 row)
```

Restringiendo Permisos con REVOKE

Por defecto, un usuario normal no tiene ningún privilegio sobre ningún objeto de la base de datos de la cual no es propietario. Para revocar explícitamente un privilegio que actualmente está otorgado, el propietario del objeto (o un superusuario) puede usar el comando REVOKE. Este comando es muy similar en formato al comando GRANT.

Aquí tiene su sintaxis:

```
REVOKE privilege [, ...] ON object [, ...]
       FROM { PUBLIC | username | GROUP groupname }
```

La sintaxis de la estructura del comando REVOKE es idéntica a la del comando GRANT, con la excepción de que el comando SQL es REVOKE en lugar de GRANT, y la palabra clave FROM es usada, en vez de la palabra clave TO.

Nota: La revocación de privilegios a PUBLIC sólo afecta al grupo especial ``public'', el cual incluye a todos los usuarios. La revocación de privilegios para PUBLIC no afectará a ningún usuario a los que explícitamente se les hayan asignado dichos privilegios.

Supongamos que los privilegios UPDATE sobre la tabla books han sido otorgados al usuario david. Cuando David es transferido a otro departamento, y no necesita más la capacidad de modificar información en la tabla book, usted debería revocar el privilegio UPDATE de David sobre la tabla de libros.

El Ejemplo siguiente usa el comando rápido \z de psql para comprobar los permisos sobre la tabla de libros, revelando que david tiene privilegios de escritura a dicha tabla. Un estamento REVOKE explícitamente revoca entonces los privilegios UPDATE y DELETE sobre la tabla de libros para el usuario david. Finalmente, otra ejecución de \z es ejecutada para verificar la revocación del privilegio.

Ejemplo de revocar privilegios

```
booktown=# \z books Access permissions for database "booktown"
Relation  | Access permissions
----------+--------------------------------
books     | {"=","manager=arwR","david=w"}
(1 row)

booktown=# REVOKE UPDATE, DELETE ON books
booktown-# FROM david;
CHANGE

booktown=# \z books
Access permissions for database "booktown"
Relation  | Access permissions
----------+----------------------
books     | {"=","manager=arwR"}
(1 row)
```

Usando Vistas para el Control de Acceso

Aunque usted no puede controlar el acceso de lectura a específicas columnas o filas de uan tabla, puede hacerlo indirectamente a través del uso adecuado de vistas. Mediante la creación de una vista sobre una tabla, y forzando a los usuarios a acceder a la tabla a través de dicha vista, usted puede permitir el acceso sólo a las deseadas filas o columnas seleccionadas.

Usted limita las columnas especificando una lista de columnas en el estamento SELECT de la vista cuando usted la crea. La vista sólo retornará las columnas que usted especifique. Usted limita las filas escribiendo una cláusula WHERE en el estamento SELECT de la vista. La vista entonces retornará sólo aquéllas filas que coincidan con la cláusula WHERE (vea el Capítulo 4, para más información sobre crear vistas).

Como los privilegios ACL pueden ser aplicados tanto a las vistas como a las tablas, usted puede entonces garantizar privilegios SELECT para la vista limitada, pero no para la tabla misma. Los usuarios podrán seleccionar desde la vista aunque no tendrán acceso a la tabla.

Por ejemplo, La tienda Book Town tiene una tabla de stocks correlacionando un número ISBN de libro con su precio de coste, precio de venta y stock actual. La estructura de la tabla se muestra en la tabla siguiente.

La tabla de stocks		
Columna	Tipo	Modificador
isbn	text	NOT NULL
cost	numeric(5,2)	
retail	numeric(5,2)	
stock	integer	

Suponga que el gerente de Book Town no desea que el personal del ventas tenga acceso al coste de cada libro. Esta información puede ser restringida generando una vista que retorna sólo los datos relativos al isbn, precio de venta y stock existente. El Ejemplo 10-20 crea dicha vista, otorga permisos al grupo de ventas (sales), y verifica los privilegios con el comando rápido \z de psql.

Ejemplo de Controlar privilegios SELECT con una vista.

```
booktown=# CREATE VIEW stock_view
booktown-# AS SELECT isbn, retail, stock
booktown-# FROM stock;
CREATE

booktown=# GRANT SELECT ON stock_view TO GROUP sales;
CHANGE

booktown=# \z stock
Access permissions for database "booktown"
Relation      | Access permissions
--------------+-------------------------------------
stock         |
stock_backup  |
stock_view    | {"=","manager=arwR","group sales=r"}
(3 rows)
```

El ejemplo siguiente demuestra la adición de un nuevo usuario, barbara. Esta garantiza privilegios SELECT sobre la vista stock_view. Como la usuaria barbara no tiene ningún privilegio implícito sobre la tabla de stocks, ésta es inaccesible; este es el caso, incluso aunque la vista sobre la tabla es accesible como resultado del estamento GRANT.

Ejemplo de contolar SELECT

```
booktown=# CREATE USER barbara;
CREATE USER

booktown=# GRANT USER barbara SELECT ON stock_view;
booktown=# \c - barbara
You are now connected as new user barbara.

booktown=> SELECT * FROM stock;
ERROR: stock: Permission denied.

booktown=> SELECT * FROM stock_view;
isbn        | retail | stock
------------+--------+-------
0385121679  | 36.95  | 65
039480001X  | 32.95  | 31
0394900014  | 23.95  | 0
044100590X  | 45.95  | 89
0441172717  | 21.95  | 77
0451160916  | 28.95  | 22
0451198492  | 46.95  | 0
0451457994  | 22.95  | 0
0590445065  | 23.95  | 10
0679803335  | 24.95  | 18
0694003611  | 28.95  | 50
0760720002  | 23.95  | 28
0823015505  | 28.95  | 16
0929605942  | 21.95  | 25
1885418035  | 24.95  | 77
0394800753  | 16.95  | 4
(16 rows)
```

Advierta que cuando se conecte como el usuario barbara, el estamento SELECT para la vista stock_view tiene éxito, mientras que la tabla stock presenta un error de Permiso Denegado.