Tutorial de PostgreSQL

Publicación 9.1.0

Luis Felipe López Acevedo

21 de July de 2016

Índice general

Las siguientes páginas tienen como propósito dar una introducción sencilla sobre PostgreSQL, conceptos de bases de datos relacionales y el lenguaje SQL a aquellas personas nuevas en cualquiera de estos temas. Se requiere experiencia básica en Sistemas operativos libres. Esta documentación le dará cierta experiencia práctica con aspectos importantes del sistema PostgreSQL. No se pretende dar un tratamiento exhaustivo de los temas que se tratan.

Después de haber trabajado este tutorial, tal vez quiera leer la Parte II del manual de PostgreSQL para obtener un conocimiento más formal del lenguaje SQL, o la Parte IV para obtener información sobre cómo desarrollar aplicaciones que usan PostgreSQL. Aquellos que configuran y administran su propio servidor, deberían leer también la Parte III.

Nota: Este tutorial está dirigido a usuarios de Sistemas operativos libres basados en Ubuntu, algunas de las instrucciones especificadas aquí podrían no funcionar en otras distribuciones de GNU.

Visite el sitio Web del tutorial para obtener la última versión.

CAPÍTULO 1

Contenido

1.1 Parte I: Primero lo primero

1.1.1 Introducción

PostgreSQL es un sistema de administración de bases de datos relacionales orientadas a objetos (ORDBMS, object-relational database management system) basado en POSTGRES, Version 4.2, desarrollado en el Departamento de Ciencias Computacionales de la Universidad de California, Berkeley. POSTGRES fue pionero en muchos conceptos que solo llegaron a aparecer en algunos sistemas de bases de datos comerciales mucho tiempo después. PostgreSQL es un descendiente libre del código original de Berkeley.

Características

- Bases de datos de nivel empresarial.
- Multiplataforma: corre en los sistemas operativos más populares, incluyendo GNU/Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64) y Windows.
- Altamente escalable tanto en la cantidad de datos que puede administrar como en el número de usuarios concurrentes que puede manejar. Existen sistemas PostgreSQL en ambientes de producción que manejan más de 4 terabytes de datos [1].
- Cumplimiento completo de ACID (atomicity, consistency, isolation, durability).
- Claves foráneas (foreign keys).
- Uniones (joins).
- Vistas (views).
- Disparadores (triggers).
- Procedimientos almacenados (en diferentes lenguajes).
- Incluye la mayoría de tipos de datos de SQL:2008, como INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL y TIMESTAMP.
- Almacenamiento de objetos binarios grandes, incluyendo imágenes, sonido y video.
- Disponibilidad de interfaces de programación nativas para C/C++, Java, .Net, Perl, Python, Ruby, Tcl, ODBC, entre otros.
- Documentación excepcional.

[1] Usuarios destacados. PostgreSQL.org.. The PostgreSQL Global Development Group, 2011.

- Multi-Version Concurrency Control (MVCC).
- Point in Time Recovery (PITR).
- Tablespaces.
- Replicación asincrónica.
- Transacciones anidadas (savepoints).
- Copias de respaldo en línea o en caliente (Online/hot backups).
- Optimizador/Planificador de consultas sofisticado.
- Conjuntos de caracteres internacionales.
- Unicode.

Y por su licencia libre, cualquier persona puede usar, modificar y distribuir PostgreSQL de manera libre y gratuita para cualquier propósito, sea privado, comercial o académico.

Límites generales de PostgreSQL

Límite	Valor
Tamaño máximo de base de datos	Ilimitado
Tamaño máximo de tabla	32 TB
Tamaño máximo de fila	1,6 TB
Tamaño máximo de campo	1 GB
Máximo de filas por tabla	Ilimitado
Máximo de columnas por tabla	250 - 1600 (dependiendo del tipo de columnas)
Máximo de índices por tabla	Ilimitado

Arquitectura

Antes de empezar, es necesario comprender la arquitectura básica del sistema PostgreSQL. Entender cómo interactúan las partes de PostgreSQL hará que las siguientes páginas sean más fáciles de entender.

En la jerga de bases de datos, PostgreSQL usa un modelo *cliente/servidor*. Una sesión de PostgreSQL se compone de los siguientes procesos cooperativos (programas):

- **El servidor**, que administra los archivos de las bases de datos, acepta conexiones a las bases de datos de parte de aplicaciones clientes y ejecuta acciones sobre las bases de datos en representación de ellos. El programa servidor de bases de datos se llama *postgres*.
- **La aplicación cliente (frontend)** que desea ejecutar operaciones en las bases de datos. Las aplicaciones cliente pueden ser muy diversas por naturaleza: podría ser una herramienta con interfaz de texto, una aplicación gráfica, un servidor Web que accede a las bases de datos para mostrar páginas Web, o una herramienta especializada para el mantenimiento de bases de datos. La distribución de PostgreSQL viene con algunas aplicaciones cliente; la mayoría es desarrollada por usuarios.

Como es típico en las aplicaciones cliente/servidor, el cliente y el servidor pueden estar en diferentes máquinas. En tal caso, ambos se comunican por medio de una conexión de red TCP/IP. Esto debe tenerse presente porque los archivos a los que se puede acceder desde una máquina cliente podrían no ser accesibles para la máquina con el servidor de bases de datos.

El servidor de PostgreSQL puede manejar muchas conexiones concurrentes de diferentes clientes. Para lograrlo, inicia ("ramifica") un proceso nuevo para cada conexión. A partir de ese punto, el cliente y el nuevo proceso del servidor se

comunican sin la intervención del proceso original de *postgres*. De esta manera, el proceso maestro del servidor siempre está corriendo, esperando conexiones de clientes, mientras que los clientes y sus procesos de servidor asociados vienen y van. (Todo esto, claro está, es invisible para el usuario. Solo se menciona aquí para propósitos ilustrativos).

Notas

1.1.2 Instalación

Abra un terminal y ejecute la siguiente orden:

```
$ sudo apt-get install postgresql
```

Esta orden instalará los paquetes necesarios para tener un sistema de administración de bases de datos completo, que incluye el servidor **postgres**, el cliente de línea de órdenes **psql** y otras herramientas importantes que se describirán más adelante.

Configuración del usuario postgres

Al instalar PostgreSQL, se crea un usuario del sistema operativo llamado **postgres**. Adicionalmente, se crea un rol y una base de datos con el mismo nombre (postgres) en el sistema de bases de datos. En PostgreSQL, el concepto de rol puede pensarse como un usuario de bases de datos o como un grupo de usuarios de bases de datos. El rol **postgres** tiene cualidades de superusuario [1].

Antes de poder hacer algo productivo, es necesario darle una contraseña al rol **postgres**. Para hacerlo, ejecute lo siguiente en un terminal:

```
$ sudo -u postgres psql postgres
```

La orden anterior permite al usuario **postgres** conectarse a la base de datos del mismo nombre por medio del cliente **psql** (conocido como intérprete interactivo de PostgreSQL). Debería ver algo similar a esto:

```
$ sudo -u postgres psql postgres
[sudo] password for USUARIO:
psql (9.1.8)
Digite «help» para obtener ayuda.

postgres=#
```

Ya dentro de **psql**, puede ponerle una contraseña a **postgres** ejecutando:

```
postgres=# \password postgres
```

Escriba la contraseña y cierre **psql** presionando `Ctrl+D` (también puede escribir `\q` y luego presionar `Enter`). Debió ver algo así:

```
postgres=# \password postgres
Ingrese la nueva contraseña:
Ingrésela nuevamente:
postgres=# \q
```

Configuración de su propio usuario

Los pasos de arriba son suficientes para empezar a trabajar con PostgreSQL, pero existe una configuración adicional que ahorra mucho tiempo y le hace las cosas más fáciles y agradables a los desarrolladores que tienen instalado el

[1] PostgreSQL 9.1.8 Documentation. Chapter 20. Database Roles and Privileges. The PostgreSQL Global Development Group, 2009.

servidor localmente, en su entorno de desarrollo.

Lo que se va a hacer a continuación es crear un rol con el mismo nombre de su nombre de usuario del sistema operativo y darle privilegios de superusuario sobre el sistema de PostgreSQL. Esto le permitirá usar los programas cliente sin necesidad de proporcionar un rol y una contraseña a cada momento.

Ejecute lo siguiente en un terminal:

```
$ sudo -u postgres createuser --superuser $USER
```

La orden anterior, ejecuta la aplicación **createuser** con el usuario **postgres** y crea un superusuario con su nombre de usuario (la variable `$USER` se reemplaza automáticamente por su nombre de usuario). Si todo sale bien, no debería ver nada especial.

Ahora, asígnele una contraseña al usuario que acabó de crear, ejecutando lo siguiente en un terminal:

```
$ sudo -u postgres psql
```

En **psql** ejecute lo siguiente, reemplazando la palabra **USUARIO** por su nombre de usuario actual. (Si no conoce su nombre de usuario, escriba en otro terminal la orden `echo $USER`):

```
postgres=# \password USUARIO
```

Escriba una contraseña nueva cuando se la pidan y, finalmente, presione `Ctrl+D` para salir de **psql**.

Hecho esto, ahora puede empezar a crear bases de datos, tablas, registros y hacer todo tipo de consultas con SQL. Esto es lo que aprenderá a hacer en las siguientes páginas.

Notas

1.1.3 Creación de bases de datos

El servidor de PostgreSQL puede administrar muchas bases de datos. Típicamente, puede crear una base de datos para cada uno de sus proyectos.

Para crear una base de datos nueva, en este ejemplo llamada **misdatos**, ejecute la siguiente orden en un terminal:

```
$ createdb misdatos
```

El proceso puede durar unos segundos y, si todo sale bien, no debería ver nada especial.

La orden de arriba es una de las ganancias que trae haber creado un superusuario con su mismo nombre de usuario de sistema operativo. Si solo existiera el usuario predeterminado **postgres**, tendría que ejecutar una orden como la siguiente:

```
$ sudo -u postgres createdb misdatos
```

Y de manera similar, con otras órdenes, tendría que especificar siempre el usuario **postgres** e ingresar la contraseña.

Puede crear bases de datos con nombres diferentes. PostgreSQL le permite crear cualquier cantidad de bases de datos. Los nombres de las bases de datos tienen que empezar con una letra del alfabeto y están limitados a 63 bytes de longitud.

Una opción conveniente es crear una base de datos con el mismo nombre de su usuario. Muchas herramientas buscan predeterminadamente una base de datos con su mismo nombre de usuario cuando no se da un nombre de base de datos específico, lo que puede ahorrarle algo de escritura. Cree una base de datos con su mismo nombre de usuario, simplemente ejecute la siguiente orden en un terminal:

```
$ createdb
```

Eliminar una base de datos

Si ya no desea usar alguna de sus bases de datos, puede eliminarla. Por ejemplo, como usted es el dueño (creador) de la base de datos misdatos, puede destruirla usando la siguiente orden en un terminal:

```
$ dropdb misdatos
```

Nota: Lea los manuales.

psql, **createuser**, **createdb** y **dropdb**, son algunas de las aplicaciones cliente que vienen con el sistema PostgreSQL. Como cualquier aplicación del sistema operativo, estas también tienen sus propios manuales de uso. Para leerlos, simplemente escriba en un terminal `man app`. Por ejemplo:

```
$ man createdb
```

(Para salir del manual, presione la tecla `Q`).

1.1.4 Acceso a bases de datos

Después de haber creado una base de datos, puede acceder a ella de las siguientes formas:

- Ejecutando el terminal interactivo de PostgreSQL, llamado **psql**, que permite escribir, editar y ejecutar órdenes de SQL de manera interactiva.
- Usando una herramienta gráfica como pgAdmin o un paquete de ofimática compatible con ODBC o JDBC para crear y manipular bases de datos. Estas posibilidades no se cubren en este tutorial.
- Escribiendo una aplicación a la medida, usando cualquiera de los muchos *"bindings"* disponibles para varios lenguajes de programación. Esta posibilidad se discute más detalladamente en la Parte IV de la documentación de PostgreSQL.

Antes de continuar, cree una base de datos nueva llamada **mibd**:

```
$ createdb mibd
```

Ahora inicie **psql** para probar los ejemplos de este tutorial. Para indicarle a **psql** que quiere trabajar en la base de datos **mibd**, ejecute la siguiente orden en un terminal:

```
$ psql midb
```

Si no proporciona el nombre de la base de datos, **psql** usará la base de datos que tiene por nombre su mismo nombre de usuario, como se indicó en *Creación de bases de datos*.

En **psql** verá un mensaje de bienvenida como este:

```
$ psql mibd
psql (9.1.8)
Digite «help» para obtener ayuda.

mibd=#
```

La última línea que imprime **psql** es el *"prompt"* (`mibd=#`), que indica que **psql** está listo para escucharle y que puede empezar a escribir consultas con SQL. En la siguiente parte de este tutorial empezará a escribir consultas con SQL. El *"prompt"* también podría ser `mibd=>`, que indicaría que usted **no es superusuario**.

psql tiene un conjunto de órdenes internas, también conocidas como metaórdenes, que no son órdenes SQL. Todas ellas empiezan con una barra inversa: *"\"*. Por ejemplo, puede obtener ayuda sobre la sintaxis de varias órdenes SQL de PostgreSQL escribiendo:

```
mibd=# \h
```

(Presione la tecla `Q` para salir de la ayuda que se abre).

Puede ver todas las órdenes internas de **psql** escribiendo:

```
mibd=# \?
```

(Presione la tecla `Q` para salir de la ayuda que se abre).

Para salir de **psql** escriba lo siguiente y presione `Enter`:

```
mibd=# \q
```

1.2 Parte II: El lenguaje SQL

1.2.1 Conceptos de SQL

Esta parte del tutorial proporciona un vistazo al uso de SQL con PostgreSQL para ejecutar operaciones sencillas. Los temas que se tratan son solamente introductorios y de ninguna manera representan un tutorial completo sobre SQL. Tenga en cuenta que algunas de las características del lenguaje de PostgreSQL son extensiones hechas al estándar.

En los ejemplos que siguen, se asume que ya existe una base de datos llamada **midb**, como se describió en *Acceso a bases de datos*.

PostgreSQL es un sistema de administración de bases de datos relacionales (RDBMS). Significa que es un sistema para administrar datos guardados en relaciones. Una relación es esencialmente un término matemático para referirse a una tabla. La noción de guardar datos en tablas es tan común hoy en día que puede parecer inherentemente obvia, pero existen otras maneras de organizar las bases de datos. Los archivos y directorios de los sistemas operativos tipo Unix son un ejemplo de bases de datos jerárquicas. Un desarrollo más moderno son las bases de datos orientadas a objetos.

Cada tabla es una colección de filas. Cada fila de una tabla dada tiene el mismo número de columnas, cada una de ellas con un nombre, y cada columna es de un tipo de dato específico. Aunque las columnas tienen un orden fijo en cada fila, es importante recordar que SQL no garantiza el orden de las filas dentro de una tabla (aunque sí se pueden organizar explícitamente al mostrarlas).

Las tablas están agrupadas en bases de datos y una colección de bases de datos administrada por una sola instancia del servidor de PostgreSQL constituye un "cluster" de bases de datos.

1.2.2 Creación de tablas

Primero que todo, abra **psql** especificando la base de datos en la que quiere trabajar:

```
$ psql midb
```

Puede crear una tabla nueva especificando el nombre de la tabla junto con los nombres de las columnas y sus tipos:

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

La orden de arriba crea una tabla de climas registrados en diferentes ciudades, en diferentes fechas.

Puede escribir lo mismo de arriba en **psql** con los saltos de línea e indentación. **psql** solo ejecutará la orden después de escribir la línea que termina en punto y coma.

Los espacios en blanco (o sea, espacios, tabulaciones y saltos de línea) se pueden usar libremente en las órdenes SQL. Quiere decir que puede escribir la orden alineada de manera diferente a la de arriba, o incluso todo en una línea. Cree la tabla `weather` escribiendo la orden como aparece arriba o como aparece a continuación:

```
midb=# CREATE TABLE weather (city varchar(80), temp_lo int, temp_hi int, prcp real, date date);
```

Dos guiones (`--`) introducen comentarios. Lo que sea que haya después de estos se ignora hasta al final de la línea. SQL no diferencia entre mayúsculas y minúsculas en las palabras clave e identificadores, excepto cuando los identificadores están entre comillas dobles para preservar esa diferenciación (en el ejemplo de arriba no se hace).

`varchar(80)` especifica un tipo de dato que puede guardar cadenas de 80 caracteres arbitrarios de largas. `int` es el tipo entero común y corriente. `real` es un tipo para guardar números de coma flotante de precisión simple. `date` es un tipo de dato para almacenar fechas. (En el ejemplo de arriba la columna también se llama "date". Esto puede ser conveniente o confuso; usted elige.)

PostgreSQL admite los tipos estándar de SQL `int`, `smallint`, `real`, `double precision`, `char(N)`, `varchar(N)`, `date`, `time`, `timestamp` e `interval`, además de otros tipos de uso general y un conjunto especial de tipos geométricos. PostgreSQL se puede modificar con un número arbitrario de tipos de datos definidos por el usuario. Consecuentemente, los nombres de los tipos de datos no son palabras clave en la sistaxis, excepto en casos especiales donde se requiera por compatibilidad con el estándar SQL.

Cree una segunda tabla que guardará ciudades con sus respectivas ubicaciones geográficas:

```
midb=# CREATE TABLE cities (name varchar(80), location point);
```

El tipo de dato `point` es un ejemplo de un tipo de dato específico de PostgreSQL.

Finalmente, debería saber que si ya no necesita una tabla o quiere volverla a crear de una manera diferente, puede eliminarla usando la siguiente orden:

```
midb=# DROP TABLE nombre_de_la_tabla;
```

1.2.3 Poblar tablas

La declaración `INSERT` se usa para poblar la tabla con filas (también llamadas registros o tuplas).

Inserte una fila nueva en la tabla **weather**:

```
midb=# INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Note que todos los tipos de datos usan formatos bastante obvios. Las constantes que no son valores numéricos corrientes normalmente deben ir entre comillas simples (‘), como se ve arriba. El tipo de dato `date` es muy flexible en lo que acepta, pero en este tutorial se usará siempre el formato de fecha usado arriba.

Inserte ahora una fila nueva en la tabla `cities`. El tipo de dato `point` requiere pares ordenados como valor:

```
midb=# INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

La sintaxis usada hasta ahora requiere que uno recuerde el orden de las columnas definidas para cada tabla. Una sintaxis alternativa permite listar las columnas explícitamente.

Inserte una fila nueva en la tabla `weather` usando la sintaxis alternativa:

```
midb=# INSERT INTO weather (city, temp_lo, temp_hi, prcp, date) VALUES ('San Francisco', 43, 57, 0.0,
```

Puede listar las columnas en un orden diferente o incluso omitir algunas de ellas, si no conoce el valor para una columna específica. Por ejemplo, inserte una fila nueva en la tabla `weather`, donde no se conoce la precipitación:

```
midb=# INSERT INTO weather (date, city, temp_hi, temp_lo) VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Muchos desarrolladores consideran que listar las columnas es mejor estilo que depender del orden implícito.

1.2.4 Consultar tablas

Para recuperar datos de una tabla, se hacen consultas. Para esto, se usa la declaración de SQL `SELECT`. Esta declaración se divide en una lista de selecciones (la parte que lista las columnas que se van a devolver), una lista de tablas (la parte que lista las tablas a partir de las cuales se van a recuperar los datos) y una cualidad opcional (la parte que especifica cualquier restricción). Por ejemplo, para recuperar todas las filas de la tabla `weather`, escriba:

```
midb=# SELECT * FROM weather;
```

Arriba, * significa "todas las columnas". Así que lo siguiente daría el mismo resultado:

```
midb=# SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

Advertencia: Evite usar * en producción.
Aunque `SELECT *` sirve para hacer consultas rápidamente, su uso en código de producción se considera mal estilo, ya que si se agrega una columna nueva a la tabla el resultado cambiaría.

Al ejecutar cualquiera de las órdenes de arriba debería ver una tabla con 3 filas correspondientes a los datos que generó en la página anterior:

```
midb=# SELECT * FROM weather;

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

También puede escribir expresiones en la lista de selecciones, no solamente referencias a las columnas existentes. Escriba lo siguiente:

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

En este caso, se genera una columna nueva en el resultado, con el nombre `temp_avg`, cuyo valor corresponde al promedio de temperatura de cada fila:

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

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

Se puede agregar "restricciones" a una consulta usando la cláusula `WHERE` que especifica qué tipo de filas se desea obtener. La cláusula `WHERE` contiene una expresión Booleana y solamente se devuelven las filas para las cuales dicha expresión sea verdadera. Se permiten los operadores Booleanos usuales (AND, OR y NOT) en la "restricción". Por ejemplo, escriba lo siguiente para obtener climas registrados de San Francisco en días lluviosos:

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

Resultado:

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

Puede pedir también que los resultados de una consulta estén ordenados:

```
midb=# SELECT * FROM weather ORDER BY city;

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

En el ejemplo anterior, el orden no está bien específicado del todo, y por eso se obtienen las filas de San Francisco en cualquier orden. Para organizar no solo por el nombre de la ciudad sino también por la temperatura más baja:

```
midb=# SELECT * FROM weather ORDER BY city, temp_lo;

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

Además, puede pedir que no se muestren filas duplicadas en el resultado:

```
midb=# SELECT DISTINCT city FROM weather;

     city
---------------
 Hayward
 San Francisco
(2 filas)
```

1.2.5 Uniones entre tablas

Hasta ahora, las consultas que se han visto solamente acceden a una tabla a la vez. Sin embargo, las consultas pueden acceder a varias tablas al mismo tiempo, o acceder a la misma tabla de tal forma que se procesen varias filas al mismo tiempo. A las consultas que acceden a varias filas de la misma o de diferentes tablas a la vez se les llama consultas combinadas. Como ejemplo, digamos que se desea listar todos los registros de climas junto con la ubicación de las ciudades asociadas a ellos. Para hacerlo, es necesario comparar la columna `city` de cada una de las filas de la tabla `weather` con la columna `name` de todas las filas en la tabla `cities` y seleccionar los pares de filas donde estos valores concuerden.

Lo anterior se logra de la siguiente manera:

```
midb=# SELECT * FROM weather, cities WHERE city = name;

     city      | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+-----------
```

```
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(2 filas)
```

Observe dos cosas sobre el resultado:

- No hay resultado para la ciudad "Hayward". Esto es porque en la tabla `cities` no hay ninguna ciudad llamada de esa manera. Por esta razón, la unión ignora las filas sin pareja de la tabla `weather`. Más adelante verá cómo resolver esto.
- Hay dos columnas que contienen el nombre de la ciudad. Esto es correcto porque la lista de columnas de las tablas `weather` y `cities` se concatenan. Sin embargo, en la práctica, esto no es deseable; así que tal vez sería mejor listar explícitamente las columnas en vez de usar `*`, como se muestra a continuación:

```
    midb=# SELECT city, temp_lo, temp_hi, prcp, date, location FROM weather, cities WHERE city = nam
```

En el ejemplo de arriba, como todas las columnas en las dos tablas tienen nombres diferentes, el analizador sintáctico (parser) encuentra automáticamente a qué tabla pertenece cada columna. Si hubiera nombres de columnas duplicados en las dos tablas, sería necesario especificar la tabla a la que pertenece cada columna:

```
midb=# SELECT weather.city, weather.temp_lo, weather.temp_hi, weather.prcp,
            weather.date, cities.location
       FROM weather, cities WHERE cities.name = weather.city;
```

Generalmente se considera buen estilo especificar siempre en las uniones la tabla a la que pertenece cada columna, así la consulta no fallará en caso de agregar más adelante un nombre de columna duplicado a una de las tablas.

Las consultas combinadas vistas hasta ahora también se pueden escribir de esta manera alternativa:

```
midb=# SELECT * FROM weather INNER JOIN cities ON (weather.city = cities.name);
```

Esta sintaxis no se usa tanto como la primera, pero se muestra aquí para ayudarle a entender los temas que siguen.

Ahora vamos a ver cómo incluir los registros relacionados con la ciudad "Hayward". Lo que se desea de la consulta es que escanee la tabla `weather` y, para cada fila, que encuentre las filas que concuerdan con la tabla cities. Si no se encuentra concordancia, queremos que se agreguen "valores vacíos" en las columnas correspondientes a la tabla cities. Este tipo de consulta se llama combinación externa (outer join). (Las combinaciones que se han visto hasta ahora son internas o "inner joins".) La orden sería como esta:

```
midb=# SELECT * FROM weather LEFT OUTER JOIN cities ON (weather.city = cities.name);

     city      | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+-----------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
 Hayward       |      37 |      54 |      | 1994-11-29 |               |
(3 filas)
```

Más específicamente, esta consulta se conoce como combinación externa izquierda (left outer join) porque la tabla mencionada a la izquierda del operador de unión tendrá sus filas en el resultado por lo menos una vez, mientras que la tabla de la derecha solo tendrá aquellas filas que concuerden con alguna fila de la tabla de la izquierda. Cuando se muestra una fila de una tabla "izquierda" para la cual no hay pareja en la tabla "derecha", se sustituyen valores vacíos (null) para las columnas de la tabla "derecha".

También existen combinaciones externas derechas (right outer joins) y combinaciones externas completas (full outer joins). Intente averiguar para qué sirven estas.

También es posible unir una tabla a sí misma. Esto se conoce como autocombinación (self join). Como ejemplo, suponga que se desea encontrar registros de clima que estén en el rango de temperatura de otros registros de clima.

Así que se necesita comparar las columnas `temp_lo` y `temp_hi` de cada fila de la tabla `weather` con las columnas `temp_lo` y `temp_hi` de todas las demás filas de la tabla `weather`. Esto se puede hacer con la siguiente consulta:

```
midb=# SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high, W2.city, W2.temp_lo AS low, W2.temp_hi

     city      | low | high |     city      | low | high
---------------+-----+------+---------------+-----+------
 San Francisco |  43 |   57 | San Francisco |  46 |   50
 Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 filas)
```

En el ejemplo anterior se han usado los alias `W1` y `W2` para la tabla, de tal forma que se puedan distinguir el lado izquierdo y derecho de la unión. También puede usar este tipo de alias en otras consultas para ahorrarse letras. Por ejemplo:

```
midb=# SELECT * FROM weather w, cities c WHERE w.city = c.name;
```

Encontrará que este estilo abreviado se usa con mucha frecuencia.

1.2.6 Funciones de agregados

Como la mayoría de los demás productos de bases de datos relacionales, PostgreSQL cuenta con funciones de agregados (aggregate functions). Una función de agregado calcula un resultado único a partir de varias filas. Por ejemplo, hay agregados que calculan conteo (count), sumatorias (sum), promedios (avg), máximos (max), mínimos (min) a partir de un conjunto de filas.

Como ejemplo, se puede encontrar la temperatura baja más alta de cualquier ciudad escribiendo:

```
midb=# SELECT max(temp_lo) FROM weather;

max
-----
  46
(1 fila)
```

Si uno quisiera saber a qué ciudad o ciudades pertenece esa temperatura, uno pensaría en algo como lo siguiente (que es incorrecto):

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

La línea de arriba no funciona porque el agregado `max` no se puede usar en la cláusula `WHERE`. (Esta restricción existe porque la cláusula `WHERE` determina qué filas se van a incluir en el cálculo del agregado; por lo cual debe evaluarse antes de computar cualquier función de agregado). Sin embargo, la consulta puede reestructurarse para lograr el resultado deseado, en este caso usando una subconsulta:

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

     city
---------------
 San Francisco
(1 fila)
```

Esto funciona bien porque la subconsulta es un cálculo independiente que calcula su agregado de manera separada de lo que sucede en la consulta externa.

Los agregados también son muy útiles en combinación con las cláusulas `GROUP BY`. Por ejemplo, se puede obtener la temperatura baja mayor observada en cada ciudad con:

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

     city      | max
---------------+-----
 Hayward       |  37
 San Francisco |  46
(2 filas)
```

Lo anterior da como resultado una fila por ciudad. Cada resultado agregado se calcula sobre las filas de la tabla que concuerdan con esa ciudad. Estas filas agrupadas se pueden filtrar usando `HAVING`:

```
midb=# SELECT city, max(temp_lo) FROM weather GROUP BY city HAVING max(temp_lo) < 40;

  city   | max
---------+-----
 Hayward |  37
(1 fila)
```

1.2.7 Actualizar registros

Puede actualizar filas existentes usando la orden `UPDATE`. Suponga que descubre que todas las lecturas de temperaturas después del 28 de noviembre están excedidas en 2 grados. Puede corregir los datos así:

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

El nuevo estado de los datos sería:

```
midb=# SELECT * FROM weather;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
 Hayward       |      35 |      52 |      | 1994-11-29
(3 filas)
```

1.2.8 Borrar registros

Puede eliminar las filas de una tabla usando la orden `DELETE`. Suponga que ya no está interesado en los datos climáticos de "Hayward". Puede hacer lo siguiente para eliminarlos de la tabla:

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

Todos los registros de clima que pertenecen a "Hayward" se eliminan:

```
midb=# SELECT * FROM weather;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
(2 filas)
```

Debe tener cuidado con declaraciones de esta forma:

```
DELETE FROM nombre_de_la_tabla;
```

Si no usa condiciones (con `WHERE`), `DELETE` eliminará todas las filas de la tabla especificada, dejándola totalmente vacía. ¡El sistema no le va a pedir ninguna confirmación para ejecutar una orden como esta!

1.3 Parte III: Características avanzadas

En la parte anterior del tutorial se tocaron algunos de los conceptos básicos de SQL para almacenar y acceder a los datos en PostgreSQL. Ahora se discutirán algunas de las características más avanzadas de SQL que simplifican la administración y que previenen la pérdida o daño de los datos.

En esta parte se hará referencia en ocasiones a los ejemplos de la *Parte II: El lenguaje SQL* para cambiarlos o mejorarlos, así que será necesario que haya leído esa parte.

1.3.1 Vistas

Recuerde lo que se hizo en *Uniones entre tablas*. Suponga que la lista combinada de registros de climas y ubicaciones de las ciudades es de interés especial para su aplicación, pero que usted no desea escribir siempre la consulta cada vez que la necesite. Para estos casos, puede crear una vista a partir de la consulta, lo que le da a la consulta un nombre al cual usted se puede referir como si fuera una tabla común y corriente:

```
midb=# CREATE VIEW mivista AS SELECT city, temp_lo, temp_hi, prcp, date, location FROM weather, citie
```

Ahora puede hacer esto:

```
midb=# SELECT * FROM mivista;

     city      | temp_lo | temp_hi | prcp |    date    | location
---------------+---------+---------+------+------------+-----------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | (-194,53)
 San Francisco |      41 |      55 |    0 | 1994-11-29 | (-194,53)
(2 filas)
```

Hacer uso libre de las vistas es un aspecto clave del buen diseño de bases de datos SQL. Las vistas permiten encapsular los detalles de la estructura de las tablas, estructura que puede cambiar a medida que su aplicación evoluciona, detrás de interfaces consistentes.

Las vistas se pueden usar casi en cualquier parte donde se pueda usar una tabla. Construir vistas a partir de otras vistas también es una práctica común.

1.3.2 Claves primarias y foráneas

Siguiendo con las tablas `weather` y `cities` de la *Parte II: El lenguaje SQL*, considere el siguiente problema: suponga que quiere asegurarse de que nadie pueda insertar filas en la tabla `weather` si estas filas no tienen una ciudad que ya esté en la tabla `cities`. A esto se le conoce como mantener la integridad referencial de los datos. PostgreSQL le da las herramientas para hacer esto por usted.

Cree una base de datos nueva llamada `clima`:

```
$ createdb clima
```

Ahora acceda a la base de datos con **psql** y cree las tablas `weather` y `cities`, pero esta vez especificando las claves primarias y foráneas para mantener la integridad referencial de los datos:

```
clima=# CREATE TABLE cities (city varchar(80) primary key, location point);
```

```
clima=# CREATE TABLE weather (city varchar(80) references cities(city), temp_lo int, temp_hi int, pr
```

En la tabla `cities`, se especificó la columna `city` como clave primaria (primary key) y en la tabla `weather` la columna `city` como clave foránea (foreign key). Este es el cambio que permitirá resolver el problema descrito arriba.

Pueble ahora la tabla `cities` con las ciudades que quiera. Por ejemplo:

```
clima=# INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

Pueble también la tabla `weather` con registros referentes a cualquiera de las ciudades que agregó en la tabla `cities`. Por ejemplo:

```
clima=# INSERT INTO weather (city, temp_lo, temp_hi, prcp, date) VALUES ('San Francisco', 43, 57, 0.0
```

Ahora intente agregar un registro incorrecto en la tabla `weather` (usando una ciudad que no está registrada en la tabla `cities`):

```
clima=# INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');
```

Debería ver un error como este:

```
ERROR: inserción o actualización en la tabla «weather» viola la llave foránea «weather_city_fkey»
DETALLE: La llave (city)=(Berkeley) no está presente en la tabla «cities».
```

El comportamiento de las claves foráneas puede ajustarse específicamente para las necesidades de cada aplicación. Aquí no se va a tratar nada más allá de este ejemplo sencillo, pero puede leer el capítulo 5 del manual de PostgreSQL para obtener más información. El uso correcto de las claves foráneas mejora la calidad de las aplicaciones de bases de datos, así que preocúpese por aprender lo necesario sobre el tema.

1.3.3 Transacciones

Las transacciones son un concepto fundamental de todos los sistemas de bases de datos. El punto esencial de una transacción es su capacidad para empaquetar varios pasos en una sola operación "todo o nada". Los estados intermedios entre los pasos no son visibles para otras transacciones concurrentes, y si ocurre alguna falla que impida que se complete la transacción, entonces ninguno de los pasos se ejecuta y no se afecta la base de datos en absoluto.

Por ejemplo, considere una base de datos bancaria que contiene balances de varias cuentas de clientes y balances totales de depósito de sucursales. Suponga que queremos registrar un pago de $100 de la cuenta de Alicia a la de Roberto. Simplificando la operación exageradamente, las órdenes SQL para hacerlo se verían así:

```
UPDATE cuentas SET balance = balance - 100.00 WHERE nombre = 'Alicia';
UPDATE sucursales SET balance = balance - 100.00 WHERE nombre = (SELECT sucursal FROM cuentas WHERE n
UPDATE cuentas SET balance = balance + 100.00 WHERE nombre = 'Roberto';
UPDATE sucursales SET balance = balance + 100.00 WHERE nombre = (SELECT sucursal FROM cuentas WHERE n
```

Los detalles de estas órdenes no son importantes en este momento; lo que importa es que hay varias actualizaciones separadas involucradas para lograr esta operación más o menos sencilla. Los operadores bancarios van a querer estar seguros de que o todos estos pasos se ejecutan o no se ejecuta ninguno. Definitivamente no sería aceptable si una falla del sistema resulta en que Roberto recibe $100 que no fueron debitados de la cuenta de Alicia. Tampoco si a Alicia le debitaran y a Roberto no le abonaran. Se necesita una garantía de que si algo sale mal en el transcurso de la operación, ninguno de los pasos ejecutados hasta el momento tendrán efecto. Para el ejemplo anterior, agrupar las actualizaciones en una transacción proporciona esa garantía. De las transacciones se dice que son atómicas: desde el punto de vista de otras transacciones, la transacción ocurre completamente o no ocurre en absoluto.

También es necesario garantizar que, después que se complete una transacción y que el sistema de bases de datos tenga completo conocimiento de ella, realmente el registro haya sido permanente y que este no se perderá, incluso si llega a

suceder una falla poco tiempo después. Por ejemplo, si se estuviera registrando un retiro de Roberto, no sería aceptable que el débito de su cuenta desapareciera en una falla del sistema justo después de que él sale del banco. Una base de datos transaccional garantiza que todas las actualizaciones realizadas por una transacción se grabarán en un medio de almacenamiento permanente (en disco, por ejemplo) antes de que la transacción se reporte completamente.

Otra propiedad importante de las bases de datos transaccionales se relaciona con la noción de las actualizaciones atómicas: cuando hay muchas transacciones concurrentes, ninguna de ellas debería conocer los cambios incompletos hechos por las demás. Por ejemplo, si alguna transacción está ocupada totalizando todos los balances de una sucursal, no serviría que incluyera el débito de la sucursal de Alicia pero no el crédito a la sucursal de Roberto, ni viceversa. Así que las transacciones deben ser todo o nada, no solamente en términos de su efecto permanente en la base de datos, sino también en términos de su visibilidad a medida que suceden. Las actualizaciones hechas hasta cierto momento por una transacción abierta son invisibles para las demás transacciones hasta que la transacción se complete. A partir de su finalización, todas las actualizaciones se hacen visibles simultáneamente.

En PostgreSQL, una transacción se indica encerrando las órdenes SQL de la transacción entre las órdenes `BEGIN` y `COMMIT`. Entonces la transacción bancaria del ejemplo de arriba se vería así:

```
BEGIN;
UPDATE cuentas SET balance = balance - 100.00 WHERE nombre = 'Alicia';
-- etc etc
COMMIT;
```

Si en medio de una transacción se decide que ya no se quiere registrar los cambios (tal vez el balance de Alicia se volvió negativo en algún momento, por ejemplo), se puede recurrir a la orden `ROLLBACK` en lugar de `COMMIT` y todas las actualizaciones hasta ese punto quedarían canceladas.

De hecho, PostgreSQL trata cada declaración de SQL como si se estuviera ejecutando dentro de una transacción. Si uno no especifica una orden `BEGIN`, entonces cada declaración individual tiene un `BEGIN` y, si es exitosa, un `COMMIT` alrededor de ella. Algunas veces, a un grupo de declaraciones encerradas entre `BEGIN` y `COMMIT` se les llama un bloque de transacción.

Nota: `BEGIN` y `COMMIT` automáticos.

Algunas bibliotecas cliente usan las órdenes `BEGIN` y `COMMIT` automáticamente, de tal forma que uno obtiene el efecto de bloques de transacción sin pedirlos. Revise la documentación de la interfaz que esté usando.

Es posible controlar las declaraciones en una transacción de una manera más granular por medio de puntos de recuperación (savepoints). Los puntos de recuperación permiten descartar selectivamente algunas partes de la transacción mientras las demás sí se ejecutan. Después de definir un punto de recuperación con `SAVEPOINT`, se puede volver a él si es necesario por medio de `ROLLBACK TO`. Todos los cambios de la base de datos hechos por la transacción entre el punto de recuperación y el rollback se descartan, pero los cambios hechos antes del punto de recuperación se mantienen.

Después de volver a un punto de recuperación, este último sigue definido, o sea que se puede volver a él varias veces. Y al contrario, si uno está seguro de que no necesita volver a un punto de recuperación particular otra vez, entonces puede liberarlo para que el sistema ahorre algunos recursos. Tenga en cuenta que tanto liberar un punto de recuperación como volver a él liberará automáticamente todos los puntos de recuperación definidos después de él.

Todo esto sucede dentro del bloque de transacción, por lo tanto nada es visible para otras sesiones de la base de datos. Cuando se ejecuta el bloque de transacción, las acciones ejecutadas se hacen visibles como una unidad para otras sesiones, mientras que las acciones de rollback nunca se hacen visibles.

Retomando el ejemplo de la base de datos bancaria, suponga que se debitan $100 de la cuenta de Alicia y se abonan a la cuenta de Roberto, pero que después resulta que se debió abonar a la cuenta de Walter. Esto se podría hacer usando un punto de recuperación:

```
BEGIN;
UPDATE cuentas SET balance = balance - 100.00 WHERE nombre = 'Alicia';
SAVEPOINT mi_savepoint;
UPDATE cuentas SET balance = balance + 100.00 WHERE nombre = 'Roberto';
-- Uy ... no era la cuenta de Roberto sino la de Walter
ROLLBACK TO mi_savepoint;
UPDATE cuentas SET balance = balance + 100.00 WHERE nombre = 'Walter';
COMMIT;
```

Este ejemplo, claro, está sobresimplificado, pero existe mucha posibilidad de control en un bloque de transacción por medio de los puntos de recuperación. Es más, `ROLLBACK TO` es la única manera de retomar el control de un bloque de transacción puesto en estado de aborto por el sistema debido a un error, devolverlo completamente y reiniciarlo.

1.3.4 Funciones ventana

Una función ventana realiza una operación sobre un conjunto de filas de una tabla que de alguna manera están relacionadas con la fila actual. Esto es similar al tipo de cálculo que se puede hacer con *Funciones de agregados*. Pero a diferencia de estas, el uso de las funciones ventana no hace que las filas se agrupen en una sola fila como resultado (las filas mantienen sus identidades por separado). Entre bastidores, la función ventana puede acceder a más que solo la fila del resultado de la consulta.

Este es un ejemplo que muestra cómo comparar el salario de cada empleado con el salario promedio del departamento al que pertenecen:

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;
```

```
  depname  | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

Las tres primeras columnas del resultado vienen directamente de la tabla `empsalary`, y por cada fila en la tabla hay una fila como resultado. La cuarta columna representa un promedio tomado de todas las filas de la tabla que tienen el mismo valor en `depname` que la fila actual. (De hecho, esta es la misma función que desempeña la función de agregado `avg`, pero la cláusula `OVER` hace que sea tratada como una función ventana y computada sobre un conjunto apropiado de filas).

La llamada a una función ventana siempre contiene la cláusula `OVER` después del nombre de la función y sus argumentos. Esto es lo que la distingue sintácticamente de una función común y corriente o de una función de agregado. La cláusula `OVER` determina exactamente cómo se deben partir las filas de la consulta para que sean procesadas por la función ventana. La lista `PARTITION BY` dentro de `OVER` especifica la división de las filas en grupos, o particiones, que comparten los mismos valores de la expresión (o expresiones) `PARTITION BY`. Para cada fila, la función ventana se computa sobre las filas que están dentro de la misma partición que la fila actual.

Aunque `avg` produce el mismo resultado sin importar en qué orden procese las filas de la partición, no pasa lo mismo con las funciones ventana. En estas, se puede controlar el orden usando `ORDER BY` dentro de `OVER`. Como en este ejemplo:

```
SELECT depname, empno, salary, rank() OVER (PARTITION BY depname ORDER BY salary DESC) FROM empsalary
```

```
  depname  | empno | salary | rank
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    10 |   5200 |    2
 develop   |    11 |   5200 |    2
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
 personnel |     2 |   3900 |    1
 personnel |     5 |   3500 |    2
 sales     |     1 |   5000 |    1
 sales     |     4 |   4800 |    2
 sales     |     3 |   4800 |    2
(10 rows)
```

Como se muestra aquí, la función `rank` produce una gradación numérica dentro de la partición de la fila actual para cada valor diferente de `ORDER BY`, en el orden definido por la cláusula `ORDER BY`. `rank` no necesita parámetros explícitos porque su comportamiento lo determina la cláusula `OVER` en su totalidad.

Las filas consideradas por una función ventana son aquellas de la "tabla virtual" producida por la cláusula `FROM` de la consulta, filtrada por sus cláusulas `WHERE`, `GROUP BY` y `HAVING`. Por ejemplo, una fila removida porque no cumple la condición `WHERE` es invisible para cualquier función ventana. Una consulta puede tener muchas funciones ventana que recortan los datos de diferentes formas por medio de diferentes cláusulas `OVER`, pero todas ellas actúan sobre la misma colección de filas definidas por la tabla virtual.

Ya vimos que `ORDER BY` se puede omitir si el orden de las filas no importa. También es posible omitir `PARTITION BY`, en cuyo caso habría solamente una partición que contiene todas las filas.

Hay otro concepto importante asociado con las funciones ventana: para cada fila, hay un conjunto de filas dentro de su partición que se conoce como su "marco de ventana". Muchas funciones ventana (pero no todas) actúan solamente sobre las filas del marco, en vez de actuar sobre toda la partición. Predeterminadamente, si se usa `ORDER BY`, entonces el marco consta de todas las filas desde el inicio de la partición hasta la fila actual, más cualquier otra fila siguiente que sea igual a la fila actual de acuerdo con la cláusula `ORDER BY`. Cuando se omite `ORDER BY`, el marco predeterminado consta de todas las filas de la partición [1]. Aquí hay un ejemplo que usa `sum`:

```
SELECT salary, sum(salary) OVER () FROM empsalary;
```

```
 salary |  sum
--------+-------
   5200 | 47100
   5000 | 47100
   3500 | 47100
   4800 | 47100
   3900 | 47100
   4200 | 47100
   4500 | 47100
   4800 | 47100
   6000 | 47100
   5200 | 47100
(10 rows)
```

Arriba, como no se usa `ORDER BY` en la cláusula `OVER`, el marco de la ventana es lo mismo que la partición, que por la omisión de `PARTITION BY` es toda la tabla; en otras palabras, cada suma se hace sobre toda la tabla y por eso se obtiene el mismo resultado para cada fila resultante. Pero si se agrega una cláusula `ORDER BY`, se obtienen resultados muy diferentes:

[1] Existen opciones para definir el marco de otras formas, pero este tutorial no las cubre. Vea la Sección 4.2.8 de la documentación de PostgreSQL para más detalles.

```
SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;
```

```
 salary |  sum
--------+-------
   3500 |  3500
   3900 |  7400
   4200 | 11600
   4500 | 16100
   4800 | 25700
   4800 | 25700
   5000 | 30700
   5200 | 41100
   5200 | 41100
   6000 | 47100
(10 rows)
```

Aquí la suma se hace desde el primer salario (el más bajo) hasta el actual, incluyendo cualquiera igual al actual (note los resultados para ver los salarios repetidos).

Las funciones ventana se permiten únicamente en la lista `SELECT` y la cláusula `ORDER BY` de la consulta. En cualquier otro lugar están prohibidas, por ejemplo en las cláusulas `GROUP BY`, `HAVING` y `WHERE`. Esto se debe a que las funciones se ejecutan lógicamente después de estas cláusulas. También se ejecutan después de las funciones de agregados. Quiere decir que es válido incluir llamadas a funciones de agregados en los argumentos de una función ventana, pero no al contrario.

Si se necesita filtrar o agrupar las filas después de terminar el cálculo de la ventana, se puede usar una subselección. Por ejemplo:

```
SELECT depname, empno, salary, enroll_date
FROM
  (SELECT depname, empno, salary, enroll_date,
          rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos
     FROM empsalary
  ) AS ss
WHERE pos < 3;
```

La consulta de arriba solamente muestra las filas de la consulta interna que tengan un valor de `rank` menor que 3.

Cuando una consulta involucra muchas funciones ventana, es posible escribir cada una de ellas con una cláusula `OVER` separadamente, pero esto es redundante y propenso a errores si se desea el mismo comportamiento de ventana para varias funciones. En lugar de esto, a cada comportamiento de ventana se le puede dar un nombre en una cláusula `WINDOW` y luego hacer referencia al mismo en `OVER`. Por ejemplo:

```
SELECT sum(salary) OVER w, avg(salary) OVER w
  FROM empsalary
  WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);
```

Puede encontrar más detalles sobre funciones ventana en la Sección 4.2.8, Sección 7.2.4 y la página de referencia de SELECT en la documentación de PostgreSQL.

Notas

1.3.5 Herencia

La herencia es un concepto de bases de datos orientadas a objetos que abre nuevas posibilidades interesantes de diseño de bases de datos.

Creemos dos tablas: una tabla de ciudades (`cities`) y otra tabla de capitales (`capitals`). Naturalmente, las capitales también son ciudades, así que uno quisiera tener cierta forma de mostrar las capitales de manera implícita cuando se listan las ciudades. Si uno es realmente inteligente inventaría un esquema como este:

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

CREATE TABLE non_capitals (
  name       text,
  population real,
  altitude   int      -- (en pies)
);

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

Esto funciona bien para las consultas, pero se va poniendo feo cuando se necesita actualizar varias filas.

Una mejor solución es esta:

```
CREATE TABLE cities (
  name       text,
  population real,
  altitude   int      -- (en pies)
);

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

En este caso, una fila de `capitals` hereda todas las columnas de su tabla madre, `cities` (`name`, `population` y `altitude`). El tipo de dato de la columna `name` es `text`, que es un tipo de dato nativo de PostgreSQL para cadenas de letras de longitud variable. Las capitales de estado tienen una columna adicional, `state`, que muestra su estado. En PostgreSQL, una tabla puede heredar de cero o más tablas.

Por ejemplo, la siguiente consulta encuentra el nombre de todas las ciudades, incluyendo las capitales, que están ubicadas a una altitud superior a los 500 pies:

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

Resultado:

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

Por otro lado, la siguiente consulta encuentra todas las ciudades que no son capitales de estado y que están situadas a una altitud igual o superior a 500 pies:

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

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

Aquí, el `ONLY` antes de `cities` indica que la consulta debe ejecutarse solamente sobre la tabla de ciudades y no sobre las tablas que están debajo de ella en la jerarquía de herencia. Muchas de las órdenes que ya se han mencionado (`SELECT`, `UPDATE` y `DELETE`) admiten la notación `ONLY`.

Nota: Aunque la herencia es útil con frecuencia, no ha sido integrada con restricciones únicas ni con claves foráneas, lo que limita su utilidad. Vea la Sección 5.8 de la documentación de PostgreSQL para más detalles.

Hasta aquí llega este tutorial. PostgreSQL tiene muchas más características que no se tocaron en este tutorial introductorio. Estas características se discuten con más detalle en la Documentación de PostgreSQL.

1.4 Créditos

- Luis Felipe López Acevedo - *Autor, traductor*
- The PostgreSQL Global Development Group - *Autor original*

1.5 Licencia

Este tutorial es una traducción al español del tutorial original de PostgreSQL con algunas modificaciones adicionales. Este trabajo se considera una obra derivada y a esta se aplica la misma licencia de uso del tutorial original, PostgreSQL License, una licencia libre. El cuerpo de la licencia empieza al terminar este párrafo. Cualquier error o inconsistencia introducida en esta obra derivada es responsabilidad de Luis Felipe López Acevedo y no de los autores originales.

VERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.