4. ESTUDIO TEÓRICO DE PostgreSQL

4.1. INTRODUCIÓN

PostgreSQL es un sistema gestor de bases de datos objeto-relacional, que emplea el modelo cliente/servidor, con un proceso por cada usuario. Los sistemas gestores de Bases de Datos relacionales tradicionales (DBMS,s) soportan un modelo de datos que consisten en una colección de relaciones con nombre, que contienen atributos de un tipo específico. En los actuales sistemas, los tipos posibles incluyen numéricos de punto flotante, enteros, cadenas de caracteres, cantidades monetarias y fechas. Este modelo de datos resulta en algunas ocasiones inadecuado para determinadas aplicaciones de procesado de datos. PostgreSQL ofrece una potencia adicional sustancial al incorporar los siguientes conceptos adicionales:

- Clases.
- Herencia.
- Sobrecarga de funciones.

Aparte de las anteriormente mencionadas, PostgreSQL proporciona otras características aportan potencia y flexibilidad adicional:

- Restricciones (*Constraints*).
- Disparadores (*triggers*).
- Reglas (*rules*).
- Integridad transaccional.

Estas características colocan a PostgreSQL en la categoría de las Bases de Datos identificadas como objeto-relacionales. Nótese que éstas son diferentes de las referidas como orientadas a objetos, que en general no son bien aprovechables para soportar lenguajes de Bases de Datos relacionales tradicionales. PostgreSQL tiene algunas características que son propias del mundo de las bases de datos orientadas a objetos.

La implementación inicial de PostgreSQL se empezó en 1986. Su predecesor fue Ingres, una base de datos desarrollada por la universidad de Berkeley. PostgreSQL es considerado según varios autores como el gestor de bases de datos de código abierto más avanzado hoy en día, ofreciendo control de concurrencia multi-versión, soportando casi toda la sintaxis SQL (incluyendo subconsultas, transacciones, y tipos y funciones definidas por el usuario), contando también con un amplio conjunto de enlaces con lenguajes de programación (incluyendo C, C++, Java, Perl, Tcl y Python). Algunas de las características fundamentales de PostgreSQL son las siguientes:

- Los bloqueos de tabla han sido sustituidos por el control de concurrencia multi-versión, el cual permite a los accesos de sólo lectura continuar leyendo datos consistentes durante la actualización de registros, y permite copias de seguridad en caliente mientras la base de datos permanece disponible para consultas.
- El motor de datos soporta importantes características, incluyendo subconsultas, valores por defecto, restricciones a valores en los campos (*constraints*) y disparadores (*triggers*).

- Se cumple casi al 100% el estándar SQL92, incluyendo claves primarias, identificadores entrecomillados, forzado de tipos de cadena literales, conversión de tipos y entrada de enteros binarios y hexadecimales.
- Los tipos internos son muy amplios, incluyendo nuevos tipos de fecha/hora de rango amplio y soporte para tipos geométricos adicionales.
- La velocidad del código del motor de datos es elevada, y su tiempo de arranque ha bajado el 80% desde que la versión 6.0 fue lanzada.

PostgreSQL se distribuye bajo los términos de la licencia Berkeley. La licencia Berkeley (BSD) es muy similar a la licencia GPL, y por esa razón PostgreSQL puede ser usado, copiado y modificado libremente.

Como se ha mencionado antes, PostgreSQL es un sistema gestor de bases de datos objeto-relacional. Es compatible con el estándar ANSI SQL casi al 100%, esto hace que PostgreSQL sea un sistema gestor de bases de datos bastante portable. El hecho de que sea un sistema gestor objeto-relacional, al igual que Oracle, proporciona mayor flexibilidad en el desarrollo de aplicaciones.

PostgreSQL es bastante modular, que permite crear funciones propias fácilmente, soporta una gran variedad de tipos de datos, incluso definidos por el usuario. Proporciona interfaces de programación para diversos lenguajes tales como C, Perl, Pithon, Tcl, Java y PHP. Pero PostgreSQL no solo proporciona interfaces de programación externos, sino que dispone de un poderoso lenguaje llamado PL/pgSQL (similar al PL/SQL de Oracle) con el que se pueden hacer programas directamente integrados en la base de datos.

PostgreSQL se ajusta perfectamente en entornos Linux, pero también puede ser utilizado en otros sistemas tales como Windows o SunOS.

Algunas de las limitaciones que posee este sistema gestor de base de datos son las siguientes:

- El tamaño de una tabla esta limitado a 32 TB en todos los sistemas operativos, y los campos pueden tener un tamaño máximo de 1 GB. No obstante, si el tamaño de una tabla llega a ser muy grande, PostgreSQL la divide en varios ficheros.
- El número de filas en una tabla es ilimitado, pero el número de columnas de una tabla no puede ser superior a 1600
- El tamaño máximo de una fila es de 1.6 TB
- Posee pocos clientes gráficos para administrar la base de datos, de los cuales prácticamente la totalidad están desarrollados por terceras partes y con una interfaz Web.

4.2. INSTALACIÓN Y PUESTA EN MARCHA

A diferencia de otros sistemas gestores de bases de datos, tales como Oracle, PostgreSQL está diseñado para funcionar en casi todo tipo de máquinas, incluso en máquinas lentas con pocos megabytes de memoria. El tamaño mínimo de memoria disponible en el sistema no se puede evaluar porque esto depende del compilador, es decir, diferentes compiladores generarán código diferente en diferentes máquinas. Tampoco hay ninguna frecuencia mínima de procesador.

PostgreSQL está escrito en C, de forma que el código fuente es bastante portable. Está disponible para varios procesadores y gran cantidad de sistemas operativos, tales como Linux, Solares o Windows. A continuación se muestra una lista de las plataformas soportadas por la última versión de PostgreSQL (versión 8.1):

Sistema Operativo	Procesador	Sistema Operativo	Procesador
AIX	PowerPC	NetBSD	arm32
AIX	RS6000	NetBSD	m68k
BSD/OS	x86	NetBSD	Sparc
Debian GNU/Linux	Alpha	NetBSD	x86
Debian GNU/Linux	AMD64	OpenBSD	Sparc
Debian GNU/Linux	ARM	OpenBSD	Sparc64
Debian GNU/Linux	Athlon XP	OpenBSD	x86
Debian GNU/Linux	IA64	Red Hat Linux	AMD64
Debian GNU/Linux	m68k	Red Hat Linux	IA64
Debian GNU/Linux	MIPS	Red Hat Linux	PowerPC
Debian GNU/Linux	MIPSEL	Red Hat Linux	PowerPC 64
Debian GNU/Linux	PA-RISC	Red Hat Linux	S/390
Debian GNU/Linux	PowerPC	Red Hat Linux	S/390x
Debian GNU/Linux	S/390	Red Hat Linux	x86
Debian GNU/Linux	Sparc	Slackware Linux	x86
Debian GNU/Linux	x86	Solaris	Sparc
Fedora	AMD64	Solaris	x86
Fedora	x86	SUSE Linux	AMD64
FreeBSD	Alpha	SUSE Linux	IA64
FreeBSD	AMD64	SUSE Linux	PowerPC
FreeBSD	x86	SUSE Linux	PowerPC 64
Gentoo Linux	AMD64	SUSE Linux	S/390
Gentoo Linux	IA64	SUSE Linux	S/390x
Gentoo Linux	PowerPC 64	SUSE Linux	x86
Gentoo Linux	x86	Tru64 UNIX	Alpha
HP-UX	IA64	UnixWare	x86
HP-UX	PA-RISC	Windows	x86
IRIX	MIPS	Windows with Cygwin	x86
Mac OS X	PowerPC	Yellow Dog Linux	PowerPC
Mandrake Linux	x86		

Tabla 4.1. *Plataformas soportadas por PostgreSQL.*

Los sistemas mostrados en la tabla anterior son aquellos en los cuales PostgreSQL ha sido probado. Ello no implica que este sistema gestor de bases de datos no pueda funcionar en otro sistema diferente que no haya sido objeto de los test.

4.2.1. Instalación de PostgreSQL.

Para instalar PostgreSQL previamente se deberá descargar una versión del mismo. Desde el propio sitio Web de PostgreSQL (http://www.postgresql.org) se pueden descargar libremente tanto el código fuente como la aplicación. No obstante, en la mayoría de las distribuciones de Linux viene incluida alguna versión de PostgreSQL, teniendo únicamente que seleccionar el paquete adecuado para su instalación. Si este no es el caso, la instalación dependerá de la versión de Linux adoptada. El fabricante proporciona amplia información sobre el proceso de instalación en cada sistema.

En un sistema Linux, los pasos que se deben seguir son los siguientes:

1) Acceder al sistema como usuario *root* para no tener ningún problema.
2) Añadir el grupo *postgres* y crear el usuario *postgres* dentro del mismo.
3) Descomprimir las fuentes. Las fuentes descargadas estarán en formato .tar.gz (*tarbal*).
4) Se preparan las fuentes para compilarlas con el comando ./configure.
5) Se compilan las fuentes de PostgreSQL.
6) Configuración Post-Instalación. En esta parte, se va a inicializar la base de datos. Para ello, se empleará el siguiente comando:

initdb [opciones] *directorio_de_datos*

En el comando anterior, *directorio_de_datos* hace referencia al directorio en el que se van a almacenar las bases de datos. Las opciones soportadas por este comando se muestran en la tabla 4.2.

Opción	**Descripción**
-D, --pgdata *directorio*	Especifica localización para los datos.
-W, --pwprompt	Establece el *password* para un nuevo superusuario.
-E, --encoding *encoding*	Establece la codificación multi-byte por defecto para las nuevas bases de datos.
-i, --sysid *id*	Sysid de la base de datos para el superusuario.
-L *directorio*	Especifica de dónde se cogen los ficheros de entrada.
-d, --debug	Genera información para depuración.

Tabla 4.2. *Opciones de post-instalación en PostgreSQL.*

Una vez que se han completado todos los pasos anteriores, PostgreSQL esta listo para ser ejecutado.

4.2.2. Inicio y detención del servidor.

Para iniciar o detener el servidor PostgreSQL (también llamado *postmaster*), se deberá teclear el comando correspondiente desde el *shell* de Linux. Este comando es *postgresql*. El uso de este comando se ve en la tabla 4.3.

Comando	Descripción
postgresql start	Inicia el servidor PostgreSQL.
postgresql stop	Detiene el servidor PostgreSQL.
postgresql status	Muestra el estado del servidor.

Tabla 4.3. *Inicio y detención de PostgreSQL.*

Existe otro comando para iniciar y detener el servidor. Este comando es *pgctl*. Con este comando se pueden realizar algunas tareas más que con el anterior. Su sintaxis es la siguiente:

pgctl start [-w] [-D *directorio_datos*] [-s] [-l *nombre_fichero*] [-o *opciones*]
pgctl stop [-w] [-D *directorio_datos*] [-s] [-m *shutdoun_mode*]
pgctl restart [-w][-D *directorio_datos*] [-s] [-m *shutdoun_mode*] [-o *opciones*]
pgctl status [-D *directorio_datos*]

Esta utilidad permite iniciar, detener, reiniciar y obtener el estado del servidor PostgreSQL.

Las opciones que se pueden emplear con este comando son las siguientes:

Opción	Descripción
-D directorio_datos	Localización del área de almacenamiento de las bases de datos. Si no se especifica, se usa el directorio almacenado en la variable del sistema PGDATA.
-s	Sólo imprime errores durante la ejecución del comando.
-w	Espera a que se complete la operación. Se usa por defecto con shutdown, pero no con start o restart.
-W	No espera a que se complete la operación.
-l nombre_fichero	Guarda en un registro la información sobre el proceso de inicio o reinicio del servidor.
-o *opciones*	Opciones de la línea de comandos para pasar al *postmaster*. Sólo se usa con start o restart.
-m shutdown_mode	Modo de apagado o reinicio del servidor. Los valores posibles son 'smart' (apaga después de que todos los usuarios se han desconectado), 'fast' (apaga directamente y de forma segura el servidor) o 'inmediate' (apaga directamente y de forma no segura el servidor).

Tabla 4.4. *Opciones del comando pgctl.*

Otro modo de iniciar el servidor PostgreSQL es a través del comando *postmaster*. Cabe comentar que si se desea iniciar la ejecución de PostgreSQL como superusuario de Linux se obtendrá un error. Esto es por motivos de seguridad. Otros sistemas gestores de bases de datos, como por ejemplo MySQL si que lo permiten.

4.3. USO BASICO DE PostgreSQL

Como ya se dijo antes, PostgreSQL es un sistema gestor de bases de datos objeto-relacional basado en el modelo cliente/servidor. Este modelo cliente/servidor es exactamente el mismo que para MySQL. Como es sabido, se trata de un servidor ejecutándose en segundo plano en el sistema, que atiende consultas de los usuarios y devuelve los correspondientes resultados. Los usuarios se conectan al la base de datos a través de alguna interfaz cliente, como por ejemplo una página Web. El número máximo de clientes simultáneos que se pueden conectar al servidor es de 1024.
Existe una aplicación cliente especial que viene incluida en la distribución de PostgreSQL. Este cliente es una interfaz de usuario para administrar y gestionar las bases de datos de PostgreSQL, al estilo del monitor MySQL. Esta interfaz de usuario se denomina *psql*.

4.3.1. La interfaz *psql*.

El programa *psql* es un interprete que permite introducir, editar y ejecutar de manera interactiva comandos SQL sobre una base de datos de PostgreSQL. Por tanto, una vez que se haya accedido se presentará un *prompt* y se quedará a la espera de que se introduzcan comandos. Los comandos que se pueden introducir son de dos tipos:

- Comandos SQL, que deben terminarse por ";" o por "\g".
- Comandos propios del programa *psql*. Estos comandos no realizarán acción alguna sobre la base de datos a que se este conectado. Se escriben en una sola

línea y se distinguen de los anteriores en que empiezan por el símbolo “\”. Uno de estos comandos es “\q”, que sirve para abandonar el programa cliente.

Para invocar a la interfaz de usuario, se deberá ejecutar el comando *psql* desde el *shell* de Linux. La sintaxis de este comando es la siguiente:

psql [opciones] [*nombre_bd* [*nombre_usuario*]]

Este comando se puede utilizar para conectar a bases de datos locales o remotas. Los parámetros *nombre_bd* y *nombre_usuario* se emplean para conectar con una base de datos determinada como usuario *nombre_usuario*. El resto de opciones se detallan a continuación.

Opción	Descripción
-a	Redirecciona toda la entrada por pantalla.
-A	Muestra las tablas de salida de las consultas desalineadas.
-c <*consulta*>	Ejecuta una consulta simple y sale.
-d <*nombre_bd*>	Especifica la base de datos a la que ha de conectarse. Por defecto, se conecta a *postgres*.
-e	Muestra por pantalla las consultas enviadas al servidor.
-E	Muestra las consultas que generan los comandos internos.
-f <*nombre_fichero*>	Ejecuta las consultas desde un fichero y después se sale.
-F	Establece el separador para los campos. Por defecto se usa “|”.
-h <*host*>	Especifica el host del servidor de la base de datos.
-H	Establece el formato de tablas de salida a HTML.
-l	Muestra las bases de datos disponibles y sale
-n	Deshabilita la línea de lectura.
-o <*nombre_fichero*>	Redirecciona la salida de la consulta a un fichero.
-p <*port*>	Especifica el número de puerto del servidor de la base de datos.
-q	No muestra mensajes de salida, sólo los resultados de las consultas.
-P *var*[=*arg*]	Establece la opción *var* al valor *arg*.
-R <*string*>	Establece el separador de filas. Por defecto nueva línea.
-s	Se debe confirmar cada consulta.
-S	Una nueva línea termina la consulta.
-t	Imprime las filas únicamente.
-T *text*	Establece las opciones de la tabla HTML.
-U <*nombre_usuario*>	Especifica el usuario de la base de datos. Por defecto *postgres*.
-v *nombre_var=val*	Establece la variable ‘*nombre_var*’ al valor ‘*val*’.
-V	Muestra información de la versión y sale.
-W	Solicita el *password* antes de conectar a la base de datos.
-x	Activa la salida expandida de tabla.
-X	No lee el fichero de configuración (~/.psqlrc)

Tabla 4.5. *Opciones de psql.*

Queda patente que *psql* es una herramienta bastante útil para gestionar y operar con una base de datos, aunque con las limitaciones y la incomodidad que supone el hecho de tener una interfaz basada en línea de comandos.

4.3.2. Tareas básicas de administración.

En este apartado se describen las órdenes básicas para administrar una base de datos. La mayoría de las aplicaciones PostgreSQL asumen que el nombre de la base de datos, si no se especifica, es el mismo que el de la cuenta en el sistema.

Creación de una base de datos.
Para crear una nueva base de datos se puede emplear la siguiente orden desde el *shell* de Linux:

createdb *nombre_bd*

Si no se cuenta con los privilegios necesarios para crear bases de datos, se verá un mensaje de error. PostgreSQL permite crear cualquier número de bases de datos en un sistema dado y el usuario que la crea automáticamente se convierte en el administrador de dicha base de datos. Los nombres de las bases de datos deben comenzar por un carácter alfabético y están limitados a una longitud de 32 caracteres. No todos los usuarios están autorizados para ser administrador de una base de datos.

Acceder a una base de datos.
Para acceder a una base de datos determinada, hay varias opciones:

- Ejecutar programas de monitorización de PostgreSQL, como por ejemplo *psql*, ya comentado anteriormente, los cuales le permiten introducir, editar y ejecutar órdenes SQL interactivamente.
- Escribiendo un programa que emplee una de las interfaces que ofrece PostgreSQL.

Eliminar bases de datos.
Para eliminar una base de datos, se emplea la siguiente instrucción (hay que tener en cuenta que sólo el administrador de la base de datos podrá borrarla):

dropdb *nombre_bd*

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

4.3.3. El lenguaje de consultas

PostgreSQL implementa un subconjunto extendido de los lenguajes SQL92/98 y SQL3. Tiene muchas extensiones, tales como tipos de sistema extensibles, herencia, reglas de producción y funciones. Estas son características tomadas del lenguaje de consultas original de Postgres (PostQuel). En este apartado se verá la versión de SQL implementada por PostgreSQL. La noción fundamental en PostgreSQL es la de clase, que es una colección de instancias de un objeto. Cada instancia tiene la misma colección de atributos y cada atributo es de un tipo específico. Más aún, cada instancia tiene un identificador de objeto (OID) permanente, que es único en todo el sitio. Ya que la sintaxis SQL hace referencia a tablas, se pueden asociar los términos tabla y clase. Asimismo una fila SQL es una instancia y las columnas SQL son atributos. Las clases se agrupan en bases de datos y una colección de bases de datos gestionada por un único proceso *postmaster* (servidor de PostgreSQL) constituye una instalación o sitio.

4.3.3.1.Comandos básicos.

A continuación se van a detallar algunos de los comandos básicos que se pueden emplear desde el monitor *psql* u otra aplicación para manipular los elementos de una base de datos.

Creación de una nueva clase.
Se puede crear una nueva clase especificando el nombre de la clase, además de todos los nombres de atributo y sus tipos:

```
CREATE TABLE nombre_tabla (
        Nombre_campo1       TIPO_DATO OPCIONES,
        Nombre_campo2       TIPO_DATO OPCIONES,
        …
        Nombre_campoN       TIPO_DATO OPCIONES
);
```

Hay que tener en cuenta que las palabras clave y los identificadores son sensibles a las mayúsculas y minúsculas. Los identificadores pueden llegar a ser insensibles a mayúsculas o minúsculas si se les pone entre dobles comillas, tal como lo permite SQL92. PostgreSQL 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 tipo geométrico. Tal como se verá más adelante, PostgreSQL puede ser configurado con un número arbitrario de tipos de datos definidos por el usuario. Consecuentemente, los nombres de tipo no son sintácticamente palabras clave, excepto donde se requiera para soportar casos especiales en el estándar SQL92. Yendo más lejos, el comando CREATE es idéntico al comando usado para crear una tabla en el sistema relacional de siempre. Sin embargo, se verá que las clases tienen propiedades que son extensiones del modelo relacional.

Al crear una tabla, se le puede asignar un valor por defecto a sus columnas. Para hacer esto sólo hay que añadir DEFAULT *valor* después del nombre de la columna que se ha creado. También se pueden añadir sentencias para limitar el conjunto de valores que se pueden almacenar en una columna. Para hacer esto, hay varios métodos, que tendrán diferentes efectos sobre la columna en la que se aplican. En la tabla 4.6 se muestran estos limitadores. Para usarlos, simplemente habrá que añadirlos después de la columna en cuestión en la definición de la tabla.

Modificador	Descripción
CHECK (*condición*)	El valor introducido deberá cumplir la condición especificada.
NOT NULL	El valor introducido no puede ser NULL.
UNIQUE	Asegura que el valor introducido sea único en la columna.
PRIMARY KEY	Es una combinación de los limitadores UNIQUE y NOT NULL.

Tabla 4.6. *Modificadores de columna en PostgreSQL.*

Eliminación de una clase.
Para eliminar una tabla se dispone del comando DROP TABLE. La sintaxis es la siguiente:

```
DROP TABLE nombre_tabla;
```

Llenar una clase con instancias.
La declaración INSERT se usa para llenar una clase con instancias:

```
INSERT INTO nombre_tabla(col1, col2, …, colN)
VALUES (valor1, valor2, ..., valorN);
```

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. Así, la forma de proceder sería la siguiente:

```
COPY nombre_tabla FROM 'ruta_al_fichero/nombre_fich.txt'
USING DELIMITERS 'caracter_delimitador';
```

Donde el *nombre_tabl*a es el nombre de la tabla que se quiere poblar de datos, *carácter_delimitador* es el carácter que separa los datos en el fichero *nombre_fich*.txt. La ruta del fichero origen debe ser accesible al servidor, no al cliente, ya que el servidor lee el fichero directamente.

Consutar a una clase.
Las clases pueden ser consultadas con una selección relacional normal y consultas de proyección. Para ello se usa la declaración SQL SELECT. 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). La sintaxis completa es la siguiente:

```
SELECT [ ALL | DISTINCT [ ON ( expresion [, ...] ) ] ]
   expresion [ AS nombre ] [, ...]
   [ INTO [ TEMPORARY | TEMP ] [ TABLE ] nueva_tabla ]
   [ FROM tabla [ alias ] [, ...] ]
   [ WHERE condicion ]
   [ GROUP BY columna [, ...] ]
   [ HAVING condicion [, ...] ]
   [ { UNION [ ALL ] | INTERSECT | EXCEPT } select ]
   [ ORDER BY columna [ ASC | DESC | USING operador ] [, ...] ]
   [ FOR UPDATE [ OF nombre_clase [, ...] ] ]
   LIMIT { cont | ALL } [ { OFFSET | , } comienzo ];
```

Una de las características más importantes de PostgreSQL, que lo diferencia de otros sistemas gestores de bases de datos como MySQL es la posibilidad de emplear subconsultas. Mediante el uso de subconsultas se puede seleccionar un conjunto de resultados que han sido generados a partir de otra consulta, todo ello con una sola instrucción SQL.

Uniones (*Joins*) entre clases.
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*. La forma de hacer la unión sería la siguiente:

SELECT *nombre_rango.campo* [AS *nombre*] [,*nombre_rango.campo* [AS *nombre*]][,...]
FROM *nombre_tabla nombre_rango* [, *nombre_tabla2 nombre_rango*][,…]
WHERE *condicion*;

En la sentencia SQL anterior, *condición* puede emplear campos de cualquiera de las tablas definidas en FROM.

Los matices de las consultas *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 del producto cartesiano cuya cualificación sea verdadera, PostgreSQL calcula y devuelve los valores especificados en la lista de destino. PostgreSQL no da ningún significado a los valores duplicados en este tipo de expresiones. Esto significa que PostgreSQL en ocasiones recalcula la misma lista de destino varias veces. Esto ocurre frecuentemente cuando las expresiones booleanas se conectan con un "*or*". Para eliminar estos duplicados, debe usarse la declaración SELECT DISTINCT. Una consulta puede contener un número arbitrario de nombres de clases y sustituciones.

Actualizaciones.
Se pueden actualizar instancias existentes usando el comando UPDATE de la siguiente manera:

UPDATE *nombre_tabla*
SET *campo1 = valor1, campo2 = valor2, …*
WHERE *condicion*;

Borrado de elementos.
Los borrados se hacen usando el comando DELETE:

DELETE FROM *nombre_tabla* WHERE *condicion*;

Esto eliminará todos los registros de la tabla *nombre_tabla* que cumplan la condición *condicion*. Si no se especifica una condición, la instrucción DELETE simplemente borrará todas las instancias de la clase dada, dejándola vacía. El sistema no pedirá confirmación antes de hacer esto.

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.

Modificar tablas.
Una vez que se ha creado una tabla dentro de una base de datos, es posible modificar las características de las mismas. Para ello se dispone del comando ALTER TABLE. Con esta instrucción se pueden añadir y eliminar columnas, añadir y eliminar limitadores, cambiar valores por defecto, cambiar tipos de datos de columnas, renombrar columnas y renombrar tabla.

Privilegios.
Cuando un usuario crea una base de datos, automáticamente se define como propietario de la misma. Por defecto, el propietario de un objeto puede hacer cualquier cosa con el objeto. Si se desea que otros usuarios accedan a una base de datos, se le habrán de otorgar privilegios. Existen diferentes tipos de privilegios: SELECT, INSERT, UPDATE, DELETE, RULE, REFERENCES, TRIGGER, CREATE, TEMPORARY, EXECUTE y USAGE. Los privilegios aplicables a un tipo de objeto dependen del mismo.

Para asignar privilegios se dispone del comando SQL GRANT. La sintaxis de este comando es muy sencilla:

GRANT <lista_privilegios> ON *nombre_objeto* TO *nombre_usuario*
GRANT <lista_privilegios> ON *nombre_objeto* TO GROUP *nombre_grupo*

Si en *nombre_usuario* se especifica la palabra PUBLIC se hará referencia a todos los usuarios del sistema. Si en la lista de privilegios se especifica ALL se otorgaran todos los privilegios a los usuarios/grupos determinados.

Si se quieren revocar privilegios, existe la instrucción REVOKE, cuya sintaxis es igual a la del comando GRANT.

Los privilegios especiales para el propietario de un objeto, es decir, los privilegios DROP, GRANT, REVOKE, etc. van implícitamente asociados al propietario del objeto, por lo que no pueden ser otorgados ni revocados. El resto de privilegios si pueden ser modificados.

También es posible otorgar privilegios GRANT a un usuario distinto del propietario del objeto. Para ello bastará con añadir al final la cláusula WITH GRANT OPTIONS.

4.3.3.2.Uso de PostgreSQL en modo *batch*.

Aparte de la introducción por línea de comandos de las instrucciones una por una, existe otro modo de ejecutar estas instrucciones. PostgreSQL soporta la introducción de instrucciones SQL en modo *batch*, mediante el cual, a partir de un archivo de texto que contiene todas las instrucciones que se quieran ejecutar, se reproducirán por completo todas esas instrucciones. Una vez que se tiene este archivo de texto, con extensión SQL desde el terminal de Linux, se ejecuta el siguiente comando:

pgsql *nombre_bd* [opciones_monitor] < *fichero_batch*

El fichero *fichero_batch* se puede editar con cualquier editor de texto. También se podrá ejecutar el fichero *batch* desde el propio monitor *pgsql*. Para ello se utilizará la instrucción /i.

4.3.4. Estructura de las bases de datos.

Al ser una base de datos relacional, los datos en PostgreSQL se almacenan en tablas. En el apartado anterior ya se vieron algunas operaciones básicas sobre tablas, como son la creación y eliminación de tablas (clases). En este apartado se analizarán algunas tareas más que se pueden realizar sobre las tablas y algunos aspectos adicionales sobre las mismas en PostgreSQL.

La primera aclaración que se puede hacer en cuanto a las tablas en PostgreSQL es que estas pueden tener un máximo de 1600 columnas, aunque es cierto que para cierto tipo de datos este número puede reducirse hasta 250.

Cada tabla posee, además de las columnas definidas en su creación, una serie de columnas, llamadas columnas del sistema, que son definidas implícitamente al crear la tabla. Por tanto, los nombres de estas columnas no pueden ser usadas para nombrar columnas de usuario. Estas columnas son las siguientes:

Nombre columna	Descripción
oid	Identificador de objeto de una fila. Sólo esta presente en el caso de que la tabla se cree usando la sentencia WITH OIDS.
tableoid	Identificador de objeto de la tabla que contiene esta fila. Resulta muy útil en consultas que acceden a tablas heredadas.
xmin	Identificador de la transacción de inserción de este elemento.
cmin	Identificador de comando en la transacción de inserción.
xmax	Identificador de la transacción de borrado de este elemento.
cmax	Identificador de comando en la transacción de borrado.
ctid	Localización física de la fila dentro de la tabla.

Tabla 4.7. *Columnas del sistema en PostgreSQL.*

4.3.4.1.Esquemas.

El espacio de datos de PostgreSQL contiene una o mas bases de datos. Los usuarios y grupos comparten todo este espacio de datos, pero ningún dato es compartido entre bases de datos. Una conexión de cliente sólo puede acceder a una base de datos en el mismo momento. Pero además de los conceptos de base de datos y tabla, PostgreSQL introduce el concepto de esquema. Una base de datos contiene uno o más esquemas, cada uno con un nombre. Cada uno de ellos puede contener varias tablas, así como otros objetos, incluyendo tipos de datos, funciones y operadores. Objetos con el mismo nombre pueden ser empleados en diferentes esquemas sin presentarse ningún tipo de conflicto. Esto permite, por tanto, emplear tablas con el mismo nombre dentro de una misma base de datos. Las ventajas que ofrece el concepto de esquemas son las siguientes:

- Permite a varios usuarios usar una misma base de datos sin interferirse mutuamente.

- Organizar objetos de una base de datos en grupos lógicos más fácilmente manejables.

Por tanto, la estructura de datos de PostgreSQL queda de la siguiente forma:

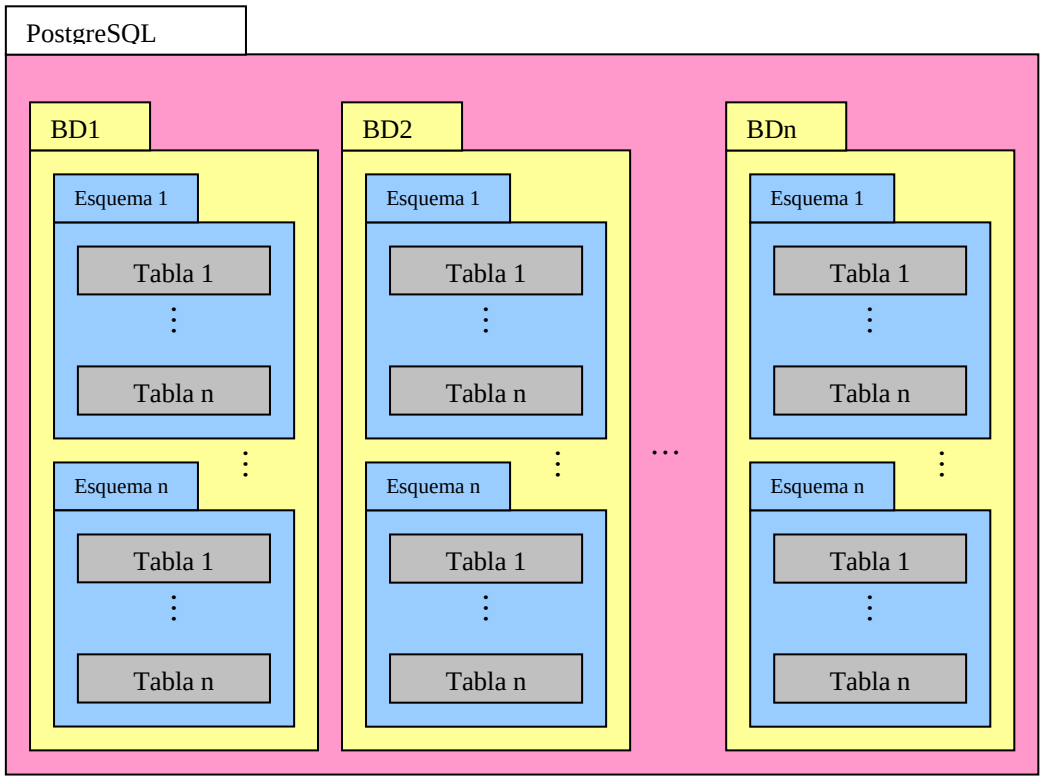

Figura 4.1. *Estructura de datos en PostgreSQL.*

Para crear un esquema se dispone del comando CREATE SCHEMA. La sintaxis es muy sencilla.

CREATE SCHEMA *nombre_esquema*;

Una vez creado el esquema, para crear, acceder o modificar tablas en el mismo se deberá especificar su nombre, un punto y el nombre de la tabla, esto es: *esquema.tabla*. De forma más general, el camino para acceder a una tabla será *nombre_bd.esquema.tabla*. Para eliminar un esquema, el comando que se emplea es DROP SCHEMA. Este comando no eliminará los objetos contenidos en él. Para hacer esto, se deberá añadir al final la cláusula CASCADE.

Las tablas que se crean sin especificar un esquema determinado se almacenan en lo que se denomina “esquema público”. No obstante, a veces resulta tedioso escribir el nombre completo (esquema y tabla) para acceder a una tabla determinada. Para ello PostgreSQL proporciona la ruta de búsqueda de esquema, mediante la cual sólo es necesario usar el nombre de la tabla, y PostgreSQL buscará la primera referencia a esa tabla en la ruta de búsqueda de esquema. El primer esquema que se encuentre que incluya dicha tabla es el que se empleará.

Los esquemas poseen el mismo sistema de privilegios para los usuarios que las tablas.

Además de los esquemas creados por el usuario y del esquema público, cada base de datos contiene un esquema *pg_catalog*, que contiene las tablas del sistema y las funciones, operadores y tipos de datos nativos.

4.3.4.2.Particionamiento.

PostgreSQL soporta particionamiento básico de tablas. El particionamiento de tablas permite dividir tablas grandes en partes más pequeñas. Algunas de las ventajas que ofrece el particionamiento son las siguientes:

- Mejora en el rendimiento de determinadas consultas.
- Mejor rendimiento en actualizaciones, debido a que las tablas pequeñas tienen índices más pequeños.
- La eliminación de grandes cantidades de datos es más simple.
- Los datos se pueden volcar a dispositivos de almacenamiento más baratos y lentos.

Actualmente, PostgreSQL soporta particiones vía herencia de tabla. Mediante este método, cada partición se crea como tabla sucesora de la tabla padre. Los siguientes métodos de partición pueden ser implementados en PostgreSQL:

- Particionamiento de rango. La tabla se divide en rangos definidos por una o varias columnas clave. No hay solapamiento entre rangos de valores asociados a diferentes valores.
- Particionamiento de lista. La tabla se divide listando explícitamente qué valores aparecen en cada partición.
- Particionamiento desordenado. Actualmente PostgreSQL no soporta este método.

4.3.4.3.Búsqueda de dependencias.

En estructuras de bases de datos complejas que involucran muchas tablas con claves externas, vistas, disparadores, funciones, etc. se crean muchas dependencias entre los distintos objetos. Para asegurar la integridad de la totalidad de la estructura, PostgreSQL se asegura de que no se puedan eliminar objetos que tengan alguna dependencia con otro ya existente, mostrando un mensaje de error si se intenta esto. Si realmente se pretende borrar el objeto, existe la sentencia DROP TABLE… CASCADE, que eliminará también las dependencias de ese objeto, eliminando también los objetos involucrados.

4.3.4.4.Tablas del sistema.

Como la mayoría de los sistemas gestores de bases de datos avanzados, PostgreSQL almacena la mayor parte de la información relativa a las bases de datos en ella contenida en tablas del sistema. Esto hace que el acceso a estos datos para analizar, depurar u otros propósitos sea muy fácil. Las tablas del sistema funcionan como tablas normales, es decir, que un usuario puede consultarla, eliminar registros, modificarlos, etc. Al ser tablas del sistema, no es muy recomendable modificar su contenido manualmente por el usuario. Para consultar las tablas de sistema disponibles, se puede ejecutar el comando *\d pg* desde el monitor *psql*. Todas las tablas del sistema son físicamente compartidas por todas las bases de datos existentes en el espacio de bases de datos. En la tabla 4.8 se

muestra un listado de todas las tablas del sistema, con una breve descripción de las mismas.

Nombre tabla	Propósito
pg_aggregate	Información sobre funciones globales.
pg_am	Métodos de acceso indexados.
pg_amop	Operadores de métodos de acceso.
pg_amproc	Procedimientos asociados a los métodos de acceso.
pg_attrdef	Valores por defecto de las columnas.
pg_attribute	Información sobre las columnas de las tablas (atributos).
pg_authid	Información sobre los roles existentes.
pg_auth_members	Relaciones entre miembros de un rol.
pg_autovacuum	Parámetros de configuración del proceso autovacuum.
pg_cast	Conversiones de tipos de datos.
pg_class	Tablas, índices, secuencias, vistas y relaciones existentes.
pg_constraint	Especificaciones check, unique, primary key y foreign key.
pg_conversion	Información sobre conversión de codificación.
pg_database	Bases de datos en el espacio de bases de datos.
pg_depend	Dependencias entre los objetos.
pg_description	Descripción o comentarios sobre los objetos de bases de datos.
pg_index	Información adicional sobre los índices.
pg_inherits	Jerarquía de herencia de tablas.
pg_language	Lenguajes disponibles para escribir funciones.
pg_largeobject	Objetos grandes.
pg_listener	Información sobre el soporte de notificación asíncrona.
pg_namespace	Información sobre los esquemas existentes.
pg_opclass	Información sobre clases de operador para métodos de acceso indexados.
pg_operator	Información sobre operadores.
pg_pltemplate	Plantillas para lenguajes de procedimiento.
pg_proc	Funciones y procedimientos.
pg_rewrite	Reglas de reescritura de consultas.
pg_shdepend	Dependencias de los objetos compartidos.
pg_statistic	Información estadística acerca de los contenidos de las bases de datos.
pg_tablespace	Tablespaces existentes en el espacio de bases de datos.
pg_trigger	Información sobre los disparadores.
pg_type	Información sobre los tipos de datos existentes.

Tabla 4.8. *Tablas del sistema en PostgreSQL*

Cada una de estas tablas posee varias columnas, que contienen toda la información relativa al propósito de la tabla. Para obtener más información sobre las columnas de cada tabla se puede acudir a la guía de referencia de PostgreSQL.

4.4. CARACTERÍSTICAS AVANZADAS DE PostgreSQL

Habiendo cubierto los aspectos básicos de PostgreSQL para acceder a los datos, a continuación se discutirá sobre aquellas características de PostgreSQL que lo distinguen de otros sistemas gestores de bases de datos. Estas características incluyen herencia, valores no-atómicos de datos (atributos basados en vectores y conjuntos), disparadores, sobrecarga de funciones, procedimientos y vistas.

4.4.1. Herencia

La herencia significa que una clase puede heredar funciones o atributos de otra clase. Si una nueva clase es derivada de una clase superior, hereda toda la información de la clase

superior. Además de toda esta información heredada, se puede implementar información adicional para la nueva clase. Dicha información no es visible para la clase padre. Para que una clase herede las características de otra, se debe declarar de la siguiente manera:

```
CREATE TABLE nombre_tabla (
columna_adicional tipo,
…
) INHERITS (nombre_clase_padre);
```

Esta instrucción creará la clase *nombre_tabla*, que heredará todas las características de la clase *nombre_clase_padre*, además de agregar información adicional (opcional). Si se desea hacer referencia a la clase padre y a todas las clases derivadas de esta, bastará con poner el símbolo "*" al final del nombre de la clase padre.

4.4.2. Valores no-atómicos.

Uno de los principios del modelo relacional es que los atributos de una relación son atómicos. PostgreSQL no posee esta restricción; los atributos pueden contener sub-valores 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.

PostgreSQL permite que los atributos de una instancia sean definidos como vectores multidimensionales de longitud fija o variable. Se pueden crear vectores de cualquiera de los tipos base o de tipos definidos por el usuario. Para ello, se deberá crear la clase de la forma habitual, y en los tipos que se quieran definir como vector se emplearán corchetes. Si entre los corchetes se pone un número, el atributo será una tabla de longitud fija. Si no se especifica número será de longitud variable.

```
CREATE TABLE nombre_tabla (
col1    tipo1,
col2    tipo2[],
col3    tipo3[][],
…
);
```

La consulta anterior creará una clase llamada *nombre_tabla* con una columna del tipo *tipo1*, un vector unidimensional del tipo *tipo2* y un vector bidimensional del tipo *tipo3*. A la hora de realizar instrucciones INSERT, cuando se agregan valores a un vector, se encierran los valores entre llaves y se separan mediante comas. PostgreSQL utiliza de forma predeterminada la convención de vectores "basados en uno", es decir, un vector de n elementos comienza con vector[1] y termina con vector[n].

4.4.3. Triggers

Los disparadores (*triggers*) se utilizan para iniciar ciertas funciones después de determinados eventos. Los disparadores se definen para tablas, y deben ser asociados a un evento del tipo INSERT o UPDATE.

4.4.4. Sobrecarga de funciones.

Como otros sistemas orientados a objetos, PostgreSQL soporta la sobrecarga de funciones. Mediante la sobrecarga de funciones, pueden existir muchas versiones de una función, diferenciándose únicamente en el número y en el tipo de parámetros que se le pasan.

4.4.5. Procedimientos.

Los procedimientos son funciones que se almacenan directamente en la base de datos. Dichos procedimientos se pueden crear a través de un lenguaje ofrecido por PostgreSQL, llamado PL/pgSQL. Este es muy parecido al lenguaje utilizado en Oracle (PL/SQL). Pero además, PostgreSQL ofrece lenguajes para oros interfaces de programación, como pueden ser PL/Tcl y PL/Perl.

4.4.6. Vistas

Las vistas son tablas virtuales que contienen información de otras tablas. Una vista no es nada más que el resultado de una consulta SELECT presentado como una tabla virtual por el sistema gestor de bases de datos. De esta forma, cada vez que se quiera acceder al resultado de esta consulta, no se tendrá que ejecutar la sentencia SELECT de nuevo, sino que bastará con echar un vistazo ala tabla virtual, simplificando por tanto las sentencias SQL. Para crear la vista, se ha de proceder de la siguiente manera:

CREATE VIEW *nombre_vista* AS
< Sentencia SELECT>;

De esta forma, se simplifica el acceso a consultas posteriores, pues sólo bastará con acceder a la tabla virtual creada.

4.5. CLAVES E ÍNDICES.

4.5.1. Índices.

Los índices, como se ha comentado anteriormente, se usan para mejorar el rendimiento de una base de datos. Se deben definir sobre columnas de las tablas (o atributos de clases, en el caso de PostgreSQL) de una base de datos. En PostgreSQL se pueden definir dos tipos de índices:

- Índices de valor. En ellos, los campos llave para un índice se especifican como nombres de columna. Una columna puede tener también un operador de clase asociado. Los operadores de clase se usan para especificar los operadores que se van a usar con un determinado índice.
- Índice funcional. Estos índices se definen como el resultado de una función definida por el usuario, aplicada a uno o más atributos de una clase simple. Los índices funcionales se pueden emplear para obtener rápido acceso a datos basados en operadores que normalmente requerirían alguna transformación para aplicarlos a los datos base.

PostgreSQL proporciona cuatro tipos de métodos de acceso para los índices: B-Tree, R-Tree, Hash y GiST.

- El método B-Tree es una estructura de datos bastante eficiente para obtener datos de una tabla. Los datos están almacenados en forma de árbol (nodos y hojas), de forma que el acceso a los datos es considerablemente rápido.
- El método R-Tree es similar al método B-Tree, en cuanto que también implementa la ordenación de los datos mediante una estructura en forma de árbol. Pero el algoritmo de indexado de este método es un poco mas complejo y con mayor rendimiento, sobre todo para cálculos sobre tipos de datos geométricos.
- El algoritmo lineal de acceso a datos desordenados (Hash) usado por PostgreSQL fue desarrollado por W. Litwin para sistemas monoprocesador basados en disco. Este algoritmo permite la reorganización dinámica de una base de datos desordenada cuando se insertan nuevos datos o se actualizan datos ya existentes. Este método de ordenamiento lineal permite al la función de ordenado cambiar mientras la base de datos esta siendo modificada. Únicamente una pequeña parte de la base de datos es afectada cuando se cambia la función de ordenado.
- Los índices GiST no son un tipo de índices simples, sino mas bien consisten en una infraestructura de varias estrategias de indexado. Se trata de un método de acceso balanceado con estructura de árbol, que actúa como una plantilla base a partir de la cual se pueden implementar otros esquemas de indexado arbitrarios. La ventaja de GiST es que permite el desarrollo de tipos de datos de usuario con sus apropiados métodos de acceso.

El optimizador de consultas de PostgreSQL considerará que tipo de índices usará en cada situación. Actualmente, sólo el método de acceso B-Tree soporta índices multi-columna, pudiéndose especificar hasta un total de 7.

Además, PostgreSQL soporta el uso de índices parciales, que son índices construidos a partir de una parte concreta de una tabla. Estos índices sólo contienen entradas para las filas de una tabla que cumplan una condición específica. Esta condición se suele expresar con una cláusula WHERE.

4.5.2. Claves.

Las claves son los campos usados para identificar una fila específica en una tabla. PostgreSQL, al igual que MySQL soporta los siguientes comandos para crear claves en las tablas existentes:

ALTER TABLE *nombre_tabla* ADD {KEY | INDEX} *nombre_indice* (*nombre_columna1*[,*nombre_columna2*,…]);
ALTER TABLE *nombre_tabla* ADD UNIQUE *nombre_indice* (*nombre_columna1*[,*nombre_columna2*,…]);
ALTER TABLE *nombre_tabla* ADD PRIMARY KEY *nombre_indice* (*nombre_columna1*[,*nombre_columna2*,…]);

Al igual que otros sistemas gestores de bases de datos, PostgreSQL soporta varios tipos de claves:

- Claves de una columna.
- Claves de columnas múltiples.
- Claves parciales.
- Claves parciales compuestas.
- Claves únicas.
- Claves externas. A diferencia de MySQL y otros sistemas gestores de bases de datos, PostgreSQL soporta las claves externas. Estas son claves que se toman de una tabla diferente. Las claves externas son extremadamente útiles cuando se trabaja con complejos modelos de datos, asegurando la integridad de los datos. Para incluir una clave externa, simplemente habrá que añadir la sentencia REFERENCE *nombre_tabla* (*columna*) a la columna que se desee.
- Claves primarias.
- Claves primarias de columnas múltiples.

Como se puede ver, PostgreSQL soporta una gran cantidad de tipos de claves, y en especial las claves externas, que no son soportadas por otros sistemas gestores de bases de datos.

4.6. TIPOS DE DATOS

PostgreSQL tiene un conjunto de tipos de datos nativo muy amplio, y por si esto no fuera suficiente, ofrece la posibilidad de añadir nuevos tipos de datos usando el comando CREATE TYPE. Cada tipo de datos posee una representación externa determinada por sus funciones de entrada y salida. A continuación se van a detallar todos los tipos de datos definidos en PostgreSQL.

4.6.1. Tipos numéricos

Los tipos numéricos consisten en enteros de 2, 4 y 8 bytes, números en punto flotante de 4 y 8 bytes y decimales de precisión variable. La tabla 4.9 muestra todos los tipos disponibles.

Tipo	Tamaño	Descripción	Rango
smallint	2 bytes	Entero de rango pequeño	-32768 a +32767
integer	4 bytes	Entero normal	-2147483648 a +2147483647
bigint	8 bytes	Entero de rango grande	-9223372036854775808 a 9223372036854775807
decimal [(p,s)]	variable	Numérico exacto de precisión variable.	Sin límite
numeric [(p,s)]	variable	Numérico exacto de precisión variable.	Sin límite
real / float(p)	4 bytes	Número en punto flotante de precisión simple.	Precisión de 6 dígitos decimales. Si se especifica con *float*, p entre 1 y 24.
double precision / float(p)	8 bytes	Número en punto flotante de precisión doble.	Precisión de 15 dígitos decimales. Si se especifica con *float*, p entre 25 y 53.
serial	4 bytes	Entero auto-incrementable.	1 a 2147483647
bigserial	8 bytes	Entero largo auto-incrementable.	1 a 9223372036854775807

Tabla 4.9. *Tipos de datos numéricos en PostgreSQL.*

Los argumentos *p* y *s* hacen referencia a la precisión y la escala con que se va a representar el número. La escala es el número de dígitos en la parte fraccionaria, y la precisión es el número de cifras significativas del número completo. Si no se especifica ninguno de estos valores, la columna podrá almacenar cualquier número hasta su límite de precisión.

En cuanto a los tipos en punto flotante, hay que decir que estos poseen una serie de valores que no están disponibles en los otros tipos. Estos valores son: *Infinity*, *-Infinity* y *NaN*, que representan los valores infinito (positivo y negativo) y un valor que no es un número (*Not a Number*).

Los tipos *serial* y *bigserial* en realidad no son ningún tipo de datos, sino una mera conveniencia de notación para establecer valores únicos en una columna (el efecto es similar a poner el modificador de columna AUTO_INCREMENT).

4.6.2. Tipo moneda

El tipo *money* almacena cantidades con precisión decimal fija. La entrada de datos de este tipo se permite en varios formatos, incluyendo entero y literales decimales. La tabla 4.10 describe este tipo.

Tipo	Tamaño	Descripción	Rango
money	4 bytes	Cantidad monetaria	-21474836.48 a +21474836.47

Tabla 4.10. *Tipo moneda en PostgreSQL.*

4.6.3. Tipos carácter.

En la tabla 4.11 se muestran los tipos de datos disponibles para el almacenamiento de caracteres.

Tipo	Tamaño	Descripción
character varying(n), varchar(n)	4+longitud bytes	Longitud variable con límite.
character(n), char(n)	4+n bytes	Longitud fija con límite.
text	4+longitud bytes	Longitud variable ilimitada.
"char"	1 byte	Carácter simple. Equivalente a *char*(1).
name	64 bytes	Tipo interno, usado para almacenar identificadores. No disponible para el usuario.

Tabla 4.11. *Tipos de datos carácter en PostgreSQL.*

Si no se especifica longitud alguna (n) se tomará un valor igual a uno en el caso de *char*, mientras que para *varchar* se aceptará cualquier tamaño. Los valores *char* se rellenan con espacios hasta ocupar la longitud n, pero estos espacios de relleno no se emplean en operaciones ni se tienen en cuenta al convertir a otro tipo.
En cuanto al tamaño en memoria, este tipo de datos ocupa 4 bytes más la longitud de la cadena, y en el caso del tipo *char*, más la longitud del relleno. En cualquier caso, el tamaño máximo de una cadena de caracteres que puede ser almacenada está en torno a 1 GB.

4.6.4. Tipo binario.

El tipo de datos binario permite almacenar cadenas binarias. La tabla 4.12 muestra las características de este tipo de datos.

Tipo	Tamaño	Descripción
bytea	4+longitud bytes	Cadena binaria de longitud variable.

Tabla 4.12. *Tipo de datos binario en PostgreSQL.*

El estándar SQL define un tipo de cadena binaria diferente, llamado BLOB. Este formato es sensiblemente diferente a *bytea*, pero las funciones y operadores soportados por ambos son prácticamente las mismas.

4.6.5. Tipos fecha/hora.

PostgreSQL soporta el conjunto completo de tipos de fecha y hora definidos en el estándar SQL. La tabla 4.13 muestra estos tipos de datos.

Tipo	Tamaño	Descripción	Valor menor	Valor mayor	Resolución
timestamp [(p)] [without time zone]	8 bytes	Fecha y hora.	4713 AC	5874897 DC	1 microseg / 14 dígitos
timestamp [(p)] with time zone	8 bytes	Fecha y hora, con zona horaria.	4713 AC	5874897 DC	1 microseg / 14 dígitos
interval [(p)]	12 bytes	Intervalos de tiempo.	-178000000 años.	178000000 años.	1 microseg / 14 dígitos
date	4 bytes	Solo fechas	4713 AC	32767 DC	1 día
time [(p)] [without time zone]	8 bytes	Horas del día.	00:00:00	24:00:00	1 microseg / 14 dígitos
time [(p)] with time zone	12 bytes	Horas del día, con zona horaria.	00:00:00+1359	24:00:00-1359	1 microseg / 14 dígitos

Tabla 4.13. *Tipos de dato fecha/hora en PostgreSQL.*

Los tipos *time*, *timestamp* e *interval* aceptan un valor opcional de precisión (p) que especifica el número de dígitos fraccionales contenidos en el campo de los segundos. Existen además dos tipos que PostgreSQL utiliza internamente, y que no están disponibles de cara al usuario. Estos tipos son *abstime* y *reltime*.

4.6.5.1.Formatos de fechas/horas.

La entrada/salida de fechas y horas es aceptada en PostgreSQL en la mayoría de los formatos razonables. Para especificar el orden en que se desea que se usen las fechas (día, mes y año) se dispone del parámetro de configuración *DateStyle*, que puede tener los valores MDY, DMY o YMD. Tanto las fechas como las horas se pueden expresar en una gran cantidad de formatos.
PostgreSQL soporta algunos valores específicos de fecha y hora para facilitar el uso de éstos formatos al usuario. Estos valores se muestran en la tabla 4.14.

Cadena de entrada	Tipos validos	Descripción
epoch	date, timestamp	1970-01-01 00:00 (tiempo “cero” del sistema Unix)
infinity	timestamp	Más tarde que cualquier otro tiempo valido.
-infinity	timestamp	Más temprano que cualquier otro tiempo valido.
now	date, time, timestamp	Tiempo actual.
today	date, timestamp	Medianoche del día actual
tomorrow	date, timestamp	Medianoche del día siguiente al actual.
yesterday	date, timestamp	Medianoche del día anterior al actual.

Tabla 4.14. *Valores específicos de fecha y hora en PostgreSQL.*

En cuanto al formato de salida de las fechas, en PostgreSQL se puede establecer a uno de los cuatro estilos disponibles (ISO 8601, SQL (Ingres), tradicional POSTGRES y German). Por defecto se toma el formato ISO.

4.6.5.2.Zonas horarias.

PostgreSQL soporta el uso de zonas horarias con los tipos de datos fecha/hora. PostgreSQL se esfuerza en ser compatible con el estándar SQL92, sin embargo, este estándar tiene una rara mezcla de tipos de fecha y hora. Esto redunda en dos problemas:

- Aunque el tipo fecha no tenga asociada una zona horaria, los tipos horarios si pueden tenerla.
- La zona horaria por defecto se define como un *offset* sobre el GMT/UTC.

Para solventar este problema, PostgreSQL asocia las zonas horarias solamente con los tipos que contienen fecha y hora, y asume zona horaria local para aquellos tipos que sólo contengan la fecha ó la hora. PostgreSQL proporciona este soporte para zonas horarias para fechas comprendidas entre 1902 y 2038. Fuera de este rango, se asume que todas las fechas se especifican y usan en la zona horaria GMT/UTC.

4.6.6. Tipo booleano.

PostgreSQL soporta el tipo booleano *bool*, que sólo puede tener dos estados: ‘*true’* o ‘*false’*. A diferencia de otros sistemas gestores de base de datos, un tercer estado del tipo ‘*unknown’* no es soportado. El tipo *bool* ocupa un byte en memoria.

4.6.7. Tipos de datos geométricos.

Una de las características que distingue a PostgreSQL de otros sistemas gestores de bases de datos es su soporte para tipos de datos geométricos. Estos tipos de datos representan objetos espaciales en dos dimensiones, lo que permite a PostgreSQL ser un excelente sistema gestor de base de datos para utilizar con algún tipo de aplicación de sistemas de información geográfica. Dentro de esta categoría, el tipo fundamental es el punto, que construye la base para el resto de tipos. En la tabla 4.15 se listan todos los tipos de datos geométricos.

Tipo	Tamaño	Descripción	Representación
point	16 bytes	Punto del plano.	(x,y)
line	32 bytes	Línea recta infinita.	((x1,y1),(x2,y2))

Tipo	Tamaño	Descripción	Representación
lseg	32 bytes	Segmento de línea recta.	((x1,y1),(x2,y2))
box	32 bytes	Caja rectangular.	((x1,y1),(x2,y2)) (esquinas opuestas de la caja)
path	16+16n bytes	Trayectoria cerrada (similar a polygon).	((x1,y1),...)
path	16+16n bytes	Trayectoria abierta.	[(x1,y1),...]
polygon	40+16n bytes	Polígono (similar a trayectoria cerrada).	((x1,y1),...)
circle	24 bytes	Círculo.	<(x,y),r> (centro y radio).

Tabla 4.15. *Tipos de datos geométricos.*

4.6.8. Tipos de direcciones de red.

PostgreSQL ofrece tipos de datos para almacenar direcciones de red IPv4, IPv6 y MAC. Esta característica lo diferencia de otros sistemas gestores de bases de datos, como por ejemplo MySQL. Otros sistemas que no poseen esta característica, deben usar tipos de texto para manejar direcciones de red. La ventaja que ofrece este tipo de de datos es que proporciona mecanismos de verificación de errores para las entradas de datos de este tipo, además de varios operadores y funciones especializados. En la tabla 4.16 se muestran las características de estos tipos de datos.

Tipo	Tamaño	Descripción
cidr	12 o 24 bytes	Direcciones IPv4 e IPv6.
inet	12 o 24 bytes	Hosts IPv4 e IPv6.
macaddr	6 bytes	Direcciones MAC.

Tabla 4.16. *Tipos de dato de dirección de red.*

El tipo *inet* almacena direcciones de *host* IPv4 o IPv6, y opcionalmente la identidad de la subred en la que se encuentra, todo ello en un campo. El tipo *cidr* almacena direcciones de red IPv4 o IPv6. El formato para almacenar estas direcciones es *dirección/y*, donde *dirección* es la dirección de red representada como una dirección IPv4 o IPv6 e *y* es el número de bits de la máscara de red. El tipo *macaddr* almacena direcciones MAC, es decir, direcciones hardware de las tarjetas de red.

4.6.9. Tipos de cadena de bits.

Existen dos tipos SQL para manejar cadenas de bits: *bit*(n) y *bit varying*(n), donde n es un entero positivo. La diferencia entre estos dos tipos es que *bit* es una cadena de bits de longitud fija (n) y *bit varying* es una cadena de longitud variable, con un tamaño máximo de n.

4.6.10. Tablas.

Como ya se comentó anteriormente, PostgreSQL permite que las columnas de una tabla sean definidas como tablas multidimensionales de cualquiera de los tipos mencionados en este apartado. Esta característica no está presente en otros sistemas gestores de bases de datos, como por ejemplo MySQL. La forma de definir una tabla con estas características ya se vio anteriormente cuando se comentaron las características avanzadas de PostgreSQL.

4.6.11. Tipos compuestos.

Como PostgreSQL permite al usuario crear sus propios tipos de datos, se puede utilizar esta característica para crear datos compuestos, es decir, tipos que engloban varios tipos. Para crear un tipo de datos compuesto, se empleará la siguiente sentencia SQL:

```
CREATE TYPE nombre_tipo AS (
n_tipo1 tipo1,
n_tipo2 tipo2
…
);
```

Una vez hecho esto, se puede emplear este tipo en cualquier tabla. Para introducir datos en este tipo, se deben englobar todos los campos entre paréntesis, separados por comas. Para acceder a un tipo de datos compuesto, se debe referenciar por el nombre del dato compuesto, seguido por un punto y el campo al que se quiere acceder, p.ej. *nombre.campo.* Esto es similar al método de acceso a un campo dentro de una tabla.

4.6.12. Tipos identificadores de objetos.

PostgreSQL usa internamente los tipos identificadores de objetos (OIDs) como claves primarias para varios sistemas de tablas. Estos tipos no se añaden a las tablas creadas por los usuarios, a menos que se especifique la cláusula WITH OIDS al crear la tabla. Estos tipos se implementan como un entero sin signo de 4 bytes. En la tabla 4.17 se muestra una breve descripción de estos tipos.

Tipo	Descripción
oid	Identificador de objeto numérico.
regproc	Nombre de función.
regprocedure	Nombre de función con tipo de argumento.
regoper	Nombre de operador.
regoperator	Nombre de operador con tipos de argumento.
regclass	Nombre de relación.
regtype	Nombre de tipo de datos.

4.17. *Tipos identificadores de objeto en PostgreSQL.*

4.6.13. Pseudo-tipos.

PostgreSQL contiene un número de entradas especiales, denominadas pseudo-tipos. Un pseudo-tipo no puede ser utilizado por el usuario como tipo para una columna, pero puede ser usado para declarar un argumento de una función o el tipo de resultado que devuelve la misma. En la tabla 4.18 se listan los pseudo-tipos existentes.

Tipo	Descripción
any	Indica que la función acepta cualquier tipo de entrada.
Anyarray	Indica que la función acepta cualquier tipo de dato tabular.
Anyelement	Indica que la función acepta cualquier tipo de datos.
Cstring	Indica que la función acepta o devuelve una cadena de caracteres terminada en NULL.
Internal	Indica que la función acepta o devuelve un tipo de datos interno del servidor.
Language_handler	Un manejador de procedimiento de llamada de lenguaje se declara para devolver

Tipo	Descripción
	language_handler.
Record	Identifica una función que devuelve un tipo de columna no especificado.
Trigger	Una función de disparador (*trigger*) se declara para devolver *trigger*.
Void	Indica que la función no devuelve valor alguno.
Opaque	Actualmente en desuso.

Tabla 4.18. *Pseudo-tipos en PostgreSQL.*

4.6.14. Tipos definidos por el usuario.

PostgreSQL permite a los usuarios crear sus propios tipos de datos. Parte de esto ya se ha visto cuando se han comentado los tipos de datos compuestos. Pero PostgreSQL ofrece aún más versatilidad en este aspecto, y no sólo permite la creación de tipos de datos personalizados con el comando CREATE TYPE, sino que además permite tipos de datos definidos en un lenguaje de programación, en concreto C. En este último caso, un tipo de datos definido por el usuario debe tener siempre funciones de entrada y salida. Estas funciones determinan cómo aparece el tipo en las cadenas (para la entrada por el usuario y para la salida para el usuario) y cómo se organiza el tipo en memoria. La función de entrada toma una cadena de caracteres delimitada por NULL como su entrada y devuelve la representación interna (en memoria) del tipo. La función de salida toma la representación interna del tipo y devuelve una cadena de caracteres delimitada por NULL. Para mas información sobre este método de creación de tipos, se remite al lector a la guía de referencia de PostgreSQL.

4.7. FUNCIONES.

PostgreSQL proporciona un gran número de funciones y operadores para los tipos de datos nativos. Además, ofrece la posibilidad al usuario de crear sus propias funciones. Para saber las funciones que implementa PostgreSQL esta disponible el comando \df, que se debe ejecutar desde el monitor *psql*.

Muchas de las funciones disponibles no están especificadas en el estándar SQL, por lo que se debe tener en cuenta este aspecto si se tienen es cuenta factores de portabilidad a otros sistemas. Aunque no obstante mucha de esta funcionalidad extendida esta presente en otras bases de datos basadas en SQL. En este capítulo se detallan las funciones que proporciona PostgreSQL para el manejo de datos, tanto las que pertenecen al estándar SQL como las que no.

4.7.1. Expresiones condicionales.

Al igual que en otros sistemas gestores de bases de datos, en PostgreSQL existen una serie de expresiones condicionales. Las disponibles son las siguientes.

CASE
Esta es una expresión condicional genérica, similar a las sentencias IF/ELSE de otros lenguajes.

COALESCE
La función COALESCE devuelve el primer argumento no nulo de los que se le pasan. Devuelve NULL sólo si todos los argumentos que recibe son NULL.

NULLIF
Esta función devuelve un valor nulo sólo si los dos parámetros que recibe son iguales. En caso de que no sean iguales, devuelve el primer parámetro.

GREATEST y **LEAST**
Estas funciones seleccionan el mayor o el menor de los valores de los que se le pasan como parámetros. Cabe comentar que estas dos funciones no están incluidas en el estándar SQL, pero son una extensión muy común.

4.7.2. Funciones matemáticas.

PostgreSQL proporciona una gran cantidad de funciones para realizar operaciones matemáticas. En la tabla 4.19 se describen todas las funciones disponibles. En dicha tabla, "*dp*" hace referencia a un número en doble precisión. Muchas de estas funciones se proporcionan en múltiples formas con diferentes argumentos.

Función	Descripción
abs(*x*)	Valor absoluto.
cbrt(dp)	Raíz cúbica.
ceil(dp o numerico)	Entero más pequeño no menor que el argumento.
ceiling(dp o numerico)	Igual que ceil.
degrees(dp)	Radianes a grados.
exp(dp o numerico)	Exponencial.
floor(dp o numerico)	Entero más grande no mayor que el argumento.
ln(dp o numerico)	Logaritmo natural.
log(dp o numerico)	Logaritmo en base 10.
log(*b* numerico,*x* numerico)	Logaritmo en base b.
mod(*y*, *x*)	Resto de *y*/*x*.
pi()	Número pi.
power(*a* dp, *b* dp)	"*a*" elevado a la potencia "*b*"
power(*a* numerico, *b* numerico)	"*a*" elevado a la potencia "*b*"
radians(dp)	Grados a radianes.
random()	Valor aleaorio entre 0.0 y 1.0
round(dp o numerico)	Redondeo al entero más cercano.
round(*v* numerico, *s* int)	Redondeo con "*s*" cifras decimales.
setseed(dp)	Establece la semilla par alas posteriores llamadas a random().
sign(dp o numerico)	Signo del argumento (-1, 0,+1).
sqrt(dp o numerico)	Raíz cuadrada.
trunc(dp o numerico)	Trunca la cifra sin ningún decimal.
trunc(*v* numerico, *s* int)	Trunca la cifra con "*s*" decimales.
width_bucket(*op* numerico, *b1* numerico, *b2* numerico, *count* int)	Retorna el cubo al que el valor "*op*" debería ser asignado en un histograma con "*count*" cubos, un límite superior de "*b1*" y un límite inferior de "*b2*".
acos(*x*)	Coseno inverso
asin(*x*)	Seno inverso.
atan(*x*)	Tangente inversa.
atan2(*x*, *y*)	Tangente inversa de *y*/*x*.
cos(*x*)	Coseno.
cot(*x*)	Cotangente.
sin(*x*)	Seno.
tan(*x*)	Tangente.
avg(*expression*)	Promedio de los valores de entrada.
bit_and(*expression*)	AND bit a bit de todos los valores de entrada.
bit_or(*expression*)	OR bit a bit de todos los valores de entrada.

Función	Descripción
bool_and(*expression*)	Verdadero si todas las expresiones de entrada son verdaderas, en caso contrario, falso.
bool_or(*expression*)	Verdadero si alguna de las expresiones de entrada es verdadera, en caso contrario, falso.
count(*)	Devuelve el número de parámetros de entrada.
count(*expression*)	Número de valores de entrada para los cuales el valor de *expression* no es NULL.
every(*expression*)	Equivalente a bool_and.
max(*expression*)	Máximo valor de los parámetros de entrada.
min(*expression*)	Mínimo valor de los parámetros de entrada.
stddev(*expression*)	Desviación estándar de los valores de entrada.
sum(*expression*)	Suma de todos los valores de entrada.
variance(*expression*)	Varianza de los valores de entrada.

Tabla 4.19. *Funciones matemáticas en PostgreSQL.*

4.7.3. Funciones de cadena.

A continuación se van a describir las funciones que proporciona PostgreSQL para examinar y manipular cadenas de caracteres. Todas estas funciones funcionan con todos los tipos cadena (*character*, *character varying* y *text*), a menos que se indique lo contrario.

Función	Descripción
string || *string*	Concatenación de cadenas.
bit_length(*string*)	Número de bits en una cadena.
char_length(*string*) o character_length(*string*)	Número de caracteres en una cadena.
convert(*string* using *conversion_name*)	Cambia la codificación de la cadena usando un tipo de conversión específico.
lower(*string*)	Convierte una cadena a minúsculas.
octet_length(*string*)	Número de bytes en una cadena.
overlay(*string* placing *string* from *int* [for *int*])	Reemplaza una subcadena.
position(*substring* in *string*)	Posición de una determinada subcadena dentro de una cadena.
substring(*string* [from *int*] [for *int*])	Extrae subcadena.
substring(*string* from *pattern*)	Extrae una subcadena empleando expresiones regulares POSIX.
substring(*string* from *pattern* for *escape*)	Extrae una subcadena usando expresiones SQL regulares.
trim([leading | trailing | both] [*characters*] from *string*)	Borra la cadena mas larga que contenga sólo los "*characters*" (por defecto un espacio) del principio/fin/ambos de la cadena.
upper(*string*)	Convierte la cadena a mayúsculas.
ascii(*text*)	Código ASCII del primer carácter del argumento.
btrim(*string* text [, *characters* text])	Elimina la cadena mas larga que contenga solo los caracteres en "*characters*" (por defecto un espacio) del principio y el final de la cadena.
chr(*int*)	Carácter con el código ASCII dado.
convert(*string* text, [*src_encoding* name,] *dest_encoding* name)	Convierte la cadena a "*dest_encoding*". La codificación original se especifica en "*src_encoding*".
decode(*string* text, *type* text)	Decodifica datos binarios de una cadena previamente codificada con encode().
encode(*data* bytea, *type* text)	Codifica datos binarios a una representación ASCII única.
initcap(*text*)	Convierte la primera letra de cada palabra en mayúsculas y el resto

Función	Descripción
	en minúsculas.
length(*string* text)	Número de caracteres en una cadena.
lpad(*string* text, *length* int [, *fill* text])	Rellena la cadena por la izquierda hasta una longitud "*lenght*" con los caracteres "*fill*" (por defecto un espacio). Si la cadena es mas larga que la longitud, se recorta por la izquierda.
ltrim(*string* text [, *characters* text])	Elimina la cadena más larga que contenga sólo los caracteres de "*characters*" (por defecto un espacio) desde el principio de la cadena.
md5(*string* text)	Calcula el MD5 de una cadena, devolviendo el resultado en hexadecimal.
pg_client_encoding()	Nombre de codificador de cliente actual.
quote_ident(*string* text)	Devuelve la conveniencia de usar la cadena dada como identificador en una sentencia SQL.
quote_literal(*string* text)	Devuelve la conveniencia de usar la cadena dada como identificador literal en una sentencia SQL.
repeat(*string* text, *number* int)	Repite la cadena un número determinado de veces.
replace(*string* text, *from* text, *to* text)	Reemplaza todas las ocurrencias de "*from*" por la subcadena "*to*".
rpad(*string* text, *length* int [, *fill* text])	Similar a lpad, pero rellenando por la derecha.
rtrim(*string* text [, *characters* text])	Similar a ltrim, pero empezando desde el final.
split_part(*string* text, *delimiter* text, *field* int)	Divide la cadena con "*delimiter*", y devuelve la parte indicada en "*field*".
strpos(*string*, *substring*)	Localización de una subcadena dentro de una cadena.
substr(*string*, *from* [, *count*])	Extrae una subcadena.
to_ascii(*text* [, *encoding*])	Convierte texto a ASCII desde otra codificación.
to_hex(*number* int o bigint)	Convierte un número a su representación hexadecimal equivalente.
translate(*string* text, *from* text, *to* text)	Cualesquiera caracteres en la cadena que coincidan con "*from*" son remplazados por "*to*".

Tabla 4.20. *Funciones de cadena en PostgreSQL.*

4.7.4. Funciones para cadenas binarias.

PostgreSQL también proporciona un conjunto de funciones para trabajar con datos binarios. Estas funciones se muestran en la tabla 4.21.

Función	Descripción
string || *string*	Concatenación de cadenas.
octet_length(*string*)	Número de bytes en una cadena binaria.
position(*substring* in *string*)	Localización de una subcadena específica.
substring(*string* [*from* int] [*for* int])	Extrae subcadena.
trim([both] *bytes* from *string*)	Elimina la cadena más larga que contenga sólo los bytes "*bytes*" desde el principio hasta el final de la cadena.
get_byte(*string*, *offset*)	Extrae un byte de la cadena.
set_byte(*string*, *offset*, *newvalue*)	Establece un byte en la cadena.
get_bit(*string*, *offset*)	Extrae un bit de la cadena.
set_bit(*string*, *offset*, *newvalue*)	Establece un bit en la cadena.
btrim(*string bytea*, *bytes* bytea)	Elimina la subcadena mas larga que contenga sólo los bytes en "*bytes*" desde el principio hasta el final de la cadena.
length(*string*)	Longitud de la cadena binaria.

md5(*string*)	Calcula el MD5 de la cadena, devolviendo el resultado en hexadecimal.
decode(*string* text, *type* text)	Decodifica una cadena binaria previamente codificada con encode().
encode(*string* bytea, *type* text)	Codifica una cadena binaria a una representación ASCII única.

Tabla 4.21. *Funciones para cadenas binarias en PostgreSQL.*

4.7.5. Funciones de formato.

Las funciones de formato proveen un poderoso conjunto de herramientas para convertir varios tipos de datos (*date/time*, *int*, *float*, *numeric*) a texto formateado y convertir de texto formateado a su tipos de datos originales

Funcion	Descripción
to_char(datetime, text)	Convierte datetime a string.
to_char(timestamp, text)	Convierte timestamp a string.
to_char(int, text)	Convierte int4/int8 a string.
to_char(float, text)	Convierte float4/float8 a string.
to_char(numeric, text)	Convierte numeric a string.
to_datetime(text, text)	Convierte string a datetime.
to_date(text, text)	Convierte string a date.
to_timestamp(text, text)	Convierte string a timestamp.
to_number(text, text)	Convierte string a numeric.

Tabla 4.22. *Funciones de formato en PostgreSQL.*

4.7.6. Funciones de fecha/hora.

Las funciones de Fecha/Hora proveen un poderoso conjunto de herramientas para manipular varios tipos Date/Time. Las funciones disponibles en PostgreSQL se muestran en la tabla 4.23.

Función	Descripción
age(timestamp, timestamp)	Preserva meses y años.
age(timestamp)	Preserva meses y años de la fecha actual.
current_date	Fecha de hoy.
current_time	Hora del dia.
current_timestamp	Fecha y hora actuales.
date_part(text, timestamp)	Obtiene una parte de la fecha/hora (equivalente a extract).
date_part(text, interval)	Obtiene una parte del intervalo (equivalente a extract).
date_trunc(text, timestamp)	Trunca con una precisión especificada.
extract(field from timestamp)	Equivalente a date_part.
extract(field from interval)	Equivalente a date_part.
isfinite(timestamp)	¿Tiempo finito?
isfinite(interval)	¿Intervalo finito?
justify_hours(interval)	Ajusta el intervalo de forma que los periodos de 24h se representan como días.
justify_days(interval)	Ajusta el intervalo de forma que los periodos de 30 días se representan como meses.
localtime	Hora del día.
localtimestamp	Fecha y hora local.
now()	Equivalente a current_timestamp.
timeofday()	Fecha y hora actual.
EXTRACT (*field* FROM *source*)	Extrae un campo de la fecha/hora. Los campos que se pueden

	extraer con esta función son muchos más que con algunas similares listadas mas arriba.
Date_trunc('*field*',*source*)	Similar a trunc.
AT TIME ZONE *zone*	Convierte a una zona horaria determinada.

Tabla 4.23. *Funciones de Fecha y hora en PostgreSQL.*

4.7.7. Funciones geométricas.

Los tipos geométricos *point*, *box*, *lseg*, *line*, *path*, *polygon* y *circle* tienen un gran conjunto de funciones nativas soportadas en PostgreSQL. En la tabla 4.24 se muestra un listado de todas estas funciones.

Función	Descripción
area(box)	Área del rectángulo.
area(circle)	Área del círculo.
box(box,box)	Rectángulo de intersección de rectángulos.
center(box)	Centro del objeto.
center(circle)	Centro del objeto.
diameter(circle)	Diámetro del círculo.
height(box)	Tamaño vertical del rectángulo.
isclosed(path)	¿Ruta cerrada?
isopen(path)	¿Ruta abierta?
length(lseg)	Longitud de la línea segmento.
length(path)	Longitud de la ruta.
pclose(path)	Convierte *path* a *closed*.
point(lseg,lseg)	Intersección.
points(path)	Número de puntos.
popen(path)	Convierte *path* a *open*.
radius(circle)	Radio del círculo.
width(box)	Tamaño horizontal.
box(circle)	Convierte círculo a rectángulo.
box(point,point)	Convierte puntos a rectángulo
box(polygon)	Convierte polígono a rectángulo.
circle(box)	Convierte a circulo.
circle(point,float8)	Convierte a circulo.
lseg(box)	Convierte diagonal a *lseg*.
lseg(point,point)	Convierte a *lseg*.
path(polygon)	Convierte a *path*.
point(circle)	Convierte a punto (centro).
point(lseg,lseg)	Convierte a punto (intersección).
point(polygon)	Centro de poligono.
polygon(box)	Convierte a polígono con 12 puntos.
polygon(circle)	Convierte a polígono con 12 puntos.
polygon(npts,circle)	Convierte a poligono npts.
polygon(path)	Convierte a *polygon*.

Tabla 4.24. *Funciones geométricas en PostgreSQL.*

4.7.8. Funciones de direcciones de red.

En este apartado se muestran las funciones que proporciona PostgreSQL para trabajar con los tipos de datos *cidr*, *inet* y *macaddr*.

Función	Descripción
broadcast(inet)	Dirección de broadcast para una red.
host(inet)	Extrae la dirección IP como texto.
masklen(inet)	Extrae la longitud de la mascara de red.
set_masklen(inet, int)	Establece la longitud de la máscara de red para un valor *inet*/*cidr*.
netmask(inet)	Construye la mascara de red para una dirección.
hostmask(inet)	Construye la mascara de host para una dirección.
network(inet)	Extrae la parte de red de una dirección.
text(inet)	Extrae la dirección IP y la longitud de la máscara de red como texto.
abbrev(inet)	Devuelve la representación abreviada de la dirección como texto.
family(inet)	Extrae una familia de direcciones; 4 para IPv4, 6 para IPv6.
Trunc(macaddr)	Establece los últimos 3 bytes a cero.

Tabla 4.25. *Funciones de direcciones de red en PostgreSQL.*

4.7.9. Funciones de manipulación de secuencias.

Las secuencias son un tipo especial de tablas que se crean con el comando CREATE SEQUENCE. Las secuencias habitualmente se utilizan para generar identificadores únicos para las filas de una tabla. Las funciones de manipulación de secuencias proporcionan métodos para obtener valores de las secuencias de los objetos de tipo secuencia. Estas funciones se muestran en la tabla 4.26.

Función	Descripción
nextval(*regclass*)	Avanza en la secuencia y devuelve el nuevo valor.
currval(*regclass*)	Devuelve el último valor obtenido por *nextval* para la secuencia dada.
lastval()	Retorna el último valor obtenido con *nextval*.
setval(*regclass*, bigint)	Establece el valor actual de la secuencia.
setval(*regclass*, bigint, boolean)	Establece el valor actual de la secuencia y el indicador *is_called*.

Tabla 4.26. *Funciones de manipulación de secuencias en PostgreSQL.*

La secuencia que debe ser operada en las funciones anteriores se especifica en el argumento *regclass*, que simplemente es el OID (identificador de objeto) de la secuencia definido en la variable del sistema *pg_class*.

4.7.10. Funciones para el manejo de arrays.

A continuación se listan las funciones que proporciona PostgreSQL para el manejo de datos tabulares.

Función	Descripción
array_cat (anyarray, anyarray)	Concatena dos tablas.
array_append (anyarray, anyelement)	Añade un elemento al final de la tabla.
array_prepend (anyelement, anyarray)	Añade un elemento al principio de la tabla.
array_dims (anyarray)	Devuelve las dimensiones de la tabla en formato texto.
array_lower (anyarray, int)	Devuelve el menor valor de la dimensión determinada.
array_upper (anyarray, int)	Devuelve el mayor valor de la dimensión determinada.
array_to_string (anyarray, text)	Concatena los elementos de la tabla usando el delimitador especificado.
string_to_array (text, text)	Divide una cadena usando el delimitador proporcionado y forma una tabla con las partes obtenidas.

Tabla 4.27. *Funciones para tipos de datos tabulares en PostgreSQL.*

4.7.11. Funciones que devuelven mas de un valor.

En esta sección se describen funciones que pueden devolver más de una fila. Actualmente PostgreSQL sólo implementa funciones generadoras de series. Estas funciones son las mostradas en la tabla 4.28.

Función	Descripción.
generate_series(*start*, *stop*)	Genera una serie de valores, desde *start* hasta *stop*, con un incremento de uno.
generate_series(*start*, *stop*, *step*)	Genera una serie de valores, desde *start* hasta *stop*, con un incremento *step*.

Tabla 4.28. *Funciones generadoras de series en PostgreSQL.*

4.7.12. Funciones de información del sistema.

PostgreSQL proporciona una serie de funciones que le permiten al usuario obtener información del sistema o de alguna sesión iniciada en el mismo. La tabla 4.29 muestra este conjunto de funciones.

Función	Descripción
current_database()	Nombre de la base de datos actual.
current_schema()	Nombre del esquema actual.
current_schemas(boolean)	Nombre de todos los esquemas en el camino de búsqueda.
current_user	Nombre de usuario en el contexto de ejecución actual.
inet_client_addr()	Dirección IP de la conexión remota.
inet_client_port()	Puerto de la conexión remota.
inet_server_addr()	Dirección IP de la conexión local.
inet_server_port()	Puerto de la conexión local.
session_user	Nombre de usuario de la sesión.
pg_postmaster_start_time()	Fecha y hora de inicio del servidor PostgreSQL (*postmaster*).
user	Equivalente a *current_user*.
version()	Información de la versión de PostgreSQL.
has_table_privilege(*user*, *table*, *privilege*)	Comprueba si el usuario tiene privilegios para una tabla.
has_table_privilege(*table*, *privilege*)	Comprueba si el usuario actual tiene privilegios para una tabla.
has_database_privilege(*user*, *database*, *privilege*)	Comprueba si el usuario tiene privilegios para una base de datos.
has_database_privilege(*database*, *privilege*)	Comprueba si el usuario actual tiene privilegios para una base de datos.
has_function_privilege(*user*, *function*, *privilege*)	Comprueba si el usuario tiene privilegios para una función.
has_function_privilege(*function*, *privilege*)	Comprueba si el usuario actual tiene privilegios para una función.
has_language_privilege(*user*, *language*, *privilege*)	Comprueba si el usuario tiene privilegios para un lenguaje.
has_language_privilege(*language*, *privilege*)	Comprueba si el usuario actual tiene privilegios para un lenguaje.
pg_has_role(*user*, *role*, *privilege*)	Comprueba si el usuario tiene privilegios para un rol.
pg_has_role(*role*, *privilege*)	Comprueba si el usuario actual tiene privilegios para un rol.
has_schema_privilege(*user*, *schema*,	Comprueba si el usuario tiene privilegios para un

Función	Descripción
privilege)	esquema.
has_schema_privilege(*schema*, *privilege*)	Comprueba si el usuario actual tiene privilegios para un esquema.
has_tablespace_privilege(*user*, *tablespace*, *privilege*)	Comprueba si el usuario tiene privilegios para un tablespace.
has_tablespace_privilege(*tablespace*, *privilege*)	Comprueba si el usuario actual tiene privilegios para un tablespace.
pg_table_is_visible(*table_oid*)	¿Tabla visible?
pg_type_is_visible(*type_oid*)	¿Tipo visible?
pg_function_is_visible(*function_oid*)	¿Función visible?
pg_operator_is_visible(*operator_oid*)	¿Operador visible?
pg_opclass_is_visible(*opclass_oid*)	¿Clase del operador visible?
pg_conversion_is_visible(*conversion_oid*)	¿Conversión visible?
format_type(*type_oid*, *typemod*)	Obtiene el nombre SQL de un tipo de datos.
pg_get_viewdef(*view_name*)	Obtiene el comando CREATE VIEW para una vista (en desuso).
pg_get_viewdef(*view_name*, *pretty_bool*)	Obtiene el comando CREATE VIEW para una vista (en desuso).
pg_get_viewdef(*view_oid*)	Obtiene el comando CREATE VIEW para una vista.
pg_get_viewdef(*view_oid*, *pretty_bool*)	Obtiene el comando CREATE VIEW para una vista.
pg_get_ruledef(*rule_oid*)	Obtiene el comando CREATE RULE para un *rule_oid* determinado.
pg_get_ruledef(*rule_oid*, *pretty_bool*)	Obtiene el comando CREATE RULE para un *rule_oid* determinado.
pg_get_indexdef(*index_oid*)	Obtiene el comando CREATE INDEX para un índice determinado.
pg_get_indexdef(*index_oid*, *column_no*, *pretty_bool*)	Obtiene el comando CREATE INDEX para un índice determinado, o la definición de sólo una columna de índice cuando *column_no* es distinto de cero.
pg_get_triggerdef(*trigger_oid*)	Obtiene el comando CREATE [CONSTRAINT] TRIGGER para un disparador.
pg_get_constraintdef(*constraint_oid*)	Obtiene la definición de una constante.
pg_get_constraintdef(*constraint_oid*, *pretty_bool*)	Obtiene la definición de una constante.
pg_get_expr(*expr_text*, *relation_oid*)	Descompila la estructura interna de una expresión, asumiendo que cualquier variable en ella se refiere a la relación indicada en el segundo parámetro.
pg_get_expr(*expr_text*, *relation_oid*, *pretty_bool*)	Descompila la estructura interna de una expresión, asumiendo que cualquier variable en ella se refiere a la relación indicada en el segundo parámetro.
pg_get_userbyid(*roleid*)	Obtiene el nombre de rol con la *id* dada.
pg_get_serial_sequence(*table_name*, *column_name*)	Obtiene el nombre de la secuencia que usan los tipos *serial* o *bigserial*.
pg_tablespace_databases(*tablespace_oid*)	Obtiene una relación de OIDs que tienen objetos en el *tablespace*.
obj_description(object_oid, catalog_name)	Obtiene comentario para un objeto de una base de datos.
obj_description(object_oid)	Obtiene comentario para un objeto de una base de datos. (obsoleta)
col_description(table_oid, column_number)	Obtiene comentario sobre una columna de una tabla.

Tabla 4.29. *Funciones de información del sistema.*

4.7.13. Funciones de administración del sistema.

El último conjunto de funciones que ofrece PostgreSQL es el correspondiente a las funciones administrativas del sistema. A través de estas funciones se podrán modificar parámetros del sistema, gestionar consultas, etcétera. Las funciones disponibles se muestran en la tabla 4.30.

Función	Descripción
current_setting(*setting_name*)	Valor actual de *setting*.
set_config(*setting_name*, *new_value*, *is_local*)	Establece un parámetro y devuelve el valor.
pg_cancel_backend(*pid* int)	Cancela la ejecución de la consulta actual.
pg_reload_conf()	Origina que los procesos del servidor recarguen sus ficheros de configuración.
pg_rotate_logfile()	Cambia de fichero de registros.
pg_start_backup(*label* text)	Inicia la ejecución de un backup en línea.
pg_stop_backup()	Detiene la ejecución del backup.
pg_column_size(any)	Número de bytes empleados para almacenar un determinado valor (posiblemente comprimido).
pg_tablespace_size(oid)	Espacio en disco usado por el *tablespace* con el OID especificado.
pg_tablespace_size(name)	Espacio en disco usado por el *tablespace* con el nombre especificado.
pg_database_size(oid)	Espacio en disco usado por la base de datos con el OID especificado.
pg_database_size(name)	Espacio en disco usado por la base de datos con el nombre especificado.
pg_relation_size(oid)	Espacio en disco usado por la tabla ó índice con el OID especificado.
pg_relation_size(text)	Espacio en disco usado por la tabla ó índice con el nombre especificado.
pg_total_relation_size(oid)	Espacio total en disco ocupado por la tabla con el nombre especificado, incluyendo índices y datos grabados.
pg_total_relation_size(text)	Espacio total en disco ocupado por la tabla con el nombre especificado, incluyendo índices y datos grabados.
pg_size_pretty(bigint)	Convierte un tamaño en bytes a un formato legible con unidades de tamaño.
pg_ls_dir(*dirname* text)	Lista el contenido de un directorio.
pg_read_file(*filename* text, *offset* bigint, *length* bigint)	Devuelve el contenido de un fichero de texto.
pg_stat_file(*filename* text)	Devuelve la información acerca de un fichero.

Tabla 4.30. *Funciones de administración de PostgreSQL.*

4.7.14. Funciones definidas por el usuario.

Además de todas las funciones expuestas anteriormente, PostgreSQL ofrece al usuario la posibilidad de crear sus propias funciones. MySQL ofrece tres posibilidades de crear este tipo de funciones:

- Funciones de lenguaje de consultas. El usuario puede crear funciones para emplear en cualquier sintaxis SQL. Para ello, PostgreSQL ofrece el comando CREATE FUNCTION.
- Funciones de lenguaje de procedimiento. PostgreSQL ofrece a los programadores una serie de lenguajes de procedimiento. Estos lenguajes no están construidos dentro de PostgreSQL, sino que se proporcionan como

módulos. Con estos lenguajes es posible crear funciones de usuario más versátiles que con el comando CREATE FUNCTION.

- Funciones de lenguaje compilado (C). Las funciones escritas en C se pueden compilar en objetos que se pueden cargar de forma dinámica, y usar para implementar funciones SQL definidas por el usuario. La primera vez que la función definida por el usuario es llamada dentro del *backend* el cargador dinámico carga el código objeto de la función en memoria, y enlaza la función con el ejecutable.

Por tanto, PostgreSQL es un sistema bastante versátil en cuanto a la posibilidad de creación de funciones de usuario.

4.8. TRANSACCIONES.

Las transacciones constituyen una característica muy interesante y útil en PostgreSQL. La mayoría de los sistemas gestores de bases de datos sofisticados de hoy en día ofrecen algún tipo de implementación de transacciones. Las transacciones permiten que las operaciones que se hacen en una base de datos tengan un carácter atómico, es decir, o se hacen, o no se hacen, pero no permite que una operación se quede a medias en su ejecución. Por ejemplo, imagínese que se ejecuta en PostgreSQL una instrucción para insertar 100000 elementos en una tabla. Si se pulsara el botón de reset del ordenador mientras se ejecuta esta operación, sólo una parte de los registros se habrán insertado, y será complicado ver cuales se han insertado y cuales no. Con las transacciones no hubiese ocurrido esto, porque cuando se iniciara de nuevo el ordenador se vería que no se ha escrito ninguna fila. Por tanto, las transacciones constituyen una característica bastante potente en PostgreSQL para asegurar la integridad de los datos y las relaciones entre objetos del sistema.

Las transacciones también juegan un papel muy importante en el control de concurrencia multi-versión (MVCC), que se verá un poco más adelante. En la mayoría de los casos, el usuario no será consciente de las transacciones que se están ejecutando en su base de datos, pero a veces resulta útil empezar y finalizar transacciones manualmente. PostgreSQL ofrece esta posibilidad. Se pueden comenzar transacciones explícitamente usando el comando BEGIN. Para finalizar una transacción explícitamente existe el comando COMMIT.

Una de las utilidades más interesantes de las transacciones es que pueden usarse para deshacer cambios. Para ello, PostgreSQL dispone del comando ROLLBACK, que permite deshacer los efectos de la última transacción. La única excepción a esto es en el comando DROP TABLE, cuya transacción no puede ser desecha.

Cabe resaltar que los distintos usuarios de una base de datos sólo pueden ver los resultados de las transacciones que hayan terminado, y no de aquellas que están en curso. Además, PostgreSQL sólo puede ejecutar una transacción por *login* al mismo tiempo.

Actualmente, PostgreSQL no soporta la salvaguarda de puntos intermedios en las transacciones, al contrario de otros sistemas gestores de bases de datos como Oracle. Este aspecto resulta de gran utilidad en transacciones largas.

4.9. CONTROL DE CONCURRENCIA MULTI-VERSIÓN

El control de concurrencia multi-versión (MVCC) es una técnica avanzada para mejorar las prestaciones de una base de datos en un entorno multiusuario. A diferencia de la mayoría de otros sistemas gestores de bases de datos que usan bloqueos para el control de concurrencia, PostgreSQL mantiene la consistencia de los datos usando un modelo multi-versión. Esto significa que mientras se consulta una base de datos, cada transacción ve una imagen de los datos (una versión de la base de datos) como si fuera tiempo atrás, sin tener en cuenta el estado actual de los datos que hay por debajo. Esto evita que la transacción vea datos inconsistentes que pueden ser causados por la actualización de otra transacción concurrente en la misma fila de datos, proporcionando aislamiento transaccional para cada sesión de la base de datos.

La principal diferencia entre multi-versión y el modelo de bloqueo es que en los bloqueos MVCC derivados de una consulta (lectura) de datos no entran en conflicto con los bloqueos derivados de la escritura de datos, y de este modo la lectura nunca bloquea la escritura y la escritura nunca bloquea la lectura. No obstante, los bloqueos a nivel de tabla y a nivel de fila también están disponibles en PostgreSQL para aplicaciones que no pueden adaptar fácilmente el comportamiento multi-versión.

4.9.1. Aislamiento transaccional.

El estándar ANSI/ISO SQL define cuatro niveles de aislamiento transaccional en función de tres hechos que deben ser tenidos en cuenta entre transacciones concurrentes. Estos hechos no deseados son:

- Lecturas "sucias". Una transacción lee datos escritos por una transacción no esperada, no cursada.
- Lecturas no repetibles. Una transacción vuelve a leer datos que previamente había leído y encuentra que han sido modificados por una transacción cursada.
- Lectura "fantasma". Una transacción vuelve a ejecutar una consulta, devolviendo un conjunto de filas que satisfacen una condición de búsqueda y encuentra que otras filas que satisfacen la condición han sido insertadas por otra transacción cursada.

Los cuatro niveles de aislamiento y sus correspondientes acciones se describen en la tabla 4.31.

Nivel de aislamiento	Lectura "sucia"	Lectura no repetible	Lectura "fantasma"
Lectura no cursada	Posible	Posible	Posible
Lectura cursada	No posible	Posible	Posible
Lectura repetible	No posible	No posible	Posible
Serializable	No posible	No posible	No posible

Tabla 4.31. *Niveles de aislamiento transaccional en PostgreSQL.*

En PostgreSQL se puede elegir cualquiera de los cuatro niveles de aislamiento estándar, pero internamente sólo hay dos niveles de aislamiento distintos, que corresponden a los niveles de lectura cursada y “serializable”. Cuando se selecciona el nivel de lectura no cursada realmente se obtiene el nivel de lectura cursada, y cuando se selecciona el nivel de lectura repetible realmente se obtiene el nivel “serializable”, de forma que el nivel de

aislamiento real en estos casos es más estricto que el que realmente se selecciona. Esto esta permitido por el estándar SQL. La razón por la que PostgreSQL sólo proporciona dos niveles de aislamiento es porque esta es la única forma de mapear los niveles de aislamiento estándar a la arquitectura de concurrencia multi-versón. Para establecer el nivel de aislamiento transaccional de una transacción se dispone del comando SET TRANSACTION.

4.9.2. Nivel de lectura cursada.

Lectura cursada es el nivel de aislamiento por defecto en PostgreSQL. Cuando una transacción se ejecuta en este nivel, una consulta sólo ve los datos de antes de que la consulta comenzara, y nunca ve datos “sucios” pero si cambios en transacciones concurrentes confirmadas durante la ejecución de la consulta. Los resultados de la ejecución de SELECT o INSERT (con una consulta) no se verán afectados por transacciones concurrentes.

4.9.3. Nivel de aislamiento serializable.

La “serailización” proporciona el nivel más alto de aislamiento transaccional. Cuando una transacción está en el nivel “serializable”, una consulta sólo ve los datos cursados antes de que la transacción comience y nunca ve ni datos “sucios” ni los cambios de transacciones concurrentes cursados durante la ejecución de la transacción. Por lo tanto, este nivel emula la ejecución de transacciones en serie, como si las transacciones fueran ejecutadas un detrás de otra, en serie, en lugar de concurrentemente. Una transacción “serializable” no puede modificar filas cambiadas por otras transacciones después de que la transacción “serializable” haya empezado. Hay que tener en cuenta que los resultados de la ejecución de SELECT o INSERT (con una consulta) no se verán afectados por transacciones concurrentes.

4.9.4. Bloqueos y tablas.

PostgreSQL ofrece varios modos de bloqueo para controlar el acceso concurrente a los datos en tablas. Estos bloqueos resultan de utilidad en aplicaciones en las que el control de concurrencia multi-versión no da el comportamiento deseado. Algunos de estos modos de bloqueo los adquiere PostgreSQL automáticamente antes de la ejecución de una declaración, mientras que otros son proporcionados para ser usados por las aplicaciones. Todos los modos de bloqueo (excepto *AccessShare*) adquiridos en una transacción se mantienen hasta la duración de la transacción. En PostgreSQL existen dos tipos de bloqueos: a nivel de tabla y a nivel de fila.

4.9.4.1.Bloqueos a nivel de tabla.

En este apartado se muestran los modos de bloqueos a nivel de tabla existentes en PostgreSQL. Algunos de estos bloqueos se utilizan automáticamente cuando se ejecuta determinada consulta, pero se pueden especificar en un momento dado con la instrucción LOCK TABLES.

AccessShare
Un modo de bloqueo adquirido automáticamente sobre tablas que están siendo consultadas con SELECT o ANALYZE. PostgreSQL libera estos bloqueos después de

que se haya ejecutado la declaración. Puede presentar conflictos con los bloqueos *AccessExclusive*.

RowShare
Adquirido por las instrucciones SELECT FOR UPDATE, SELECT FOR SHARE y LOCK TABLE (IN ROW SHARE MODE). Además del bloqueo *AccessShare*, bloquea cualquier otra tabla que es referenciada en la instrucción pero no seleccionada para UPDATE o SHARE. Entra en conflictos con los modos *Exclusive* y *AccessExclusive*.

RowExclusive
Lo adquieren UPDATE, DELETE, INSERT y LOCK TABLE (IN ROW EXCLUSIVE MODE). Además del bloqueo *RowShare*, bloquea cualquier otra tabla que es referenciada en la instrucción. En general, este modo de bloqueo será adquirido por cualquier comando que modifique datos en una tabla. Choca con los modos *Share*, *ShareRowExclusive*, *Exclusive* y *AccessExclusive*.

Share
Lo adquieren CREATE INDEX y LOCK TABLE (IN SHARE MODE). Este modo protege la tabla frente a cambios concurrentes en los datos. Está en conflicto con los modos *RowExclusive*, *ShareUpdateExclusive*, *ShareRowExclusive*, *Exclusive* y *AccessExclusive*.

ShareUpdateExclusive
Este tipo de bloqueo es adquirido por el comando VACUUM. Este modo protege contra cambios concurrentes en la base de datos y ejecuciones de VACUUM. Presenta conflictos con *Share*, *ShareRowExclusive*, *Exclusive* y *AccessExclusive*.

ShareRowExclusive
Lo toma LOCK TABLE (IN SHARE ROW EXCLUSIVE MODE). Este modo no es adquirido automáticamente por ningún comando de PostgreSQL, sino que se debe especificar explícitamente con LOCK TABLE. Está en conflicto con los modos *RowExclusive*, *ShareUpdateExclusive*, *Share*, *ShareRowExclusive*, *Exclusive* y *AccessExclusive*.

Exclusive
Lo toma LOCK TABLE para declaraciones IN EXCLUSIVE MODE. Este modo no es adquirido automáticamente por ningún comando de PostgreSQL, sino que se debe especificar explícitamente con LOCK TABLE. Sólo se pueden ejecutar lecturas en paralelo mientras una transacción posea este modo de bloqueo. Entra en conflicto con los modos *RowShare*, *ShareUpdateExclusive*, *RowExclusive*, *Share*, *ShareRowExclusive*, *Exclusive* y *AccessExclusive*.

AccessExclusive
Lo toman ALTER TABLE, DROP TABLE, REINDEX, CLUSTER, VACUUM FULL y LOCK TABLE (IN ACCESS EXCLUSIVE MODE). Este modo de bloqueo garantiza que una transacción sea la única que acceda a la tabla. Choca con *AccessShare*, *RowShare*, *RowExclusive*, *ShareUpdateExclusive*, *Share*, *ShareRowExclusive* y *Exclusive*. Sólo el bloqueo *AccessExclusive* bloquea la declaración SELECT (sin FOR UPDATE/SHARE).

4.9.4.2.Bloqueos a nivel de fila.

Este tipo de bloqueos se producen cuando campos internos de una fila son actualizados (o borrados o marcados para ser actualizados). Estos bloqueos son usados internamente por PostgreSQL y no están disponibles para el usuario. PostgreSQL no retiene en memoria ninguna información sobre filas modificadas y de este modo no tiene límites para el número de filas bloqueadas en el mismo instante. Sin embargo, hay que tener en cuenta que SELECT FOR UPDATE modificará las filas seleccionadas marcándolas, de tal modo que se escribirán en el disco. Los bloqueos a nivel de fila no afectan a los datos consultados. Estos son usados para bloquear escrituras a la misma fila únicamente.

4.9.4.3.Deadlocks.

El uso de bloqueos explícitos puede aumentar la probabilidad de que ocurran “bloqueos muertos” (*deadlocks*) cuando dos o más transacciones poseen bloqueos que las demás pretenden. Además, este tipo de problemas puede ocurrir en bloqueos a nivel de fila, e incluso en bloqueos a nivel de tabla que no se definen explícitamente. Cuando suceda esto, PostgreSQL detectará esta situación y abortara una de las transacciones. La única defensa para evitar los *deadlocks* es asegurar que las aplicaciones que accedan a una base de datos adquieran los bloqueos en un orden consistente.

4.9.5. Chequeos de consistencia de datos.

Ya que las lecturas en PostgreSQL no bloquean los datos, sin tener en cuenta el nivel de aislamiento de la transacción, los datos leídos por una transacción pueden ser sobrescritos por otra. En otras palabras, si una fila es devuelta por SELECT esto no significa que esta fila realmente exista en el momento en que se devolvió (un tiempo después de que la declaración o la transacción comenzaran, por ejemplo) ni que la fila esté protegida de borrados o actualizaciones por la transacción concurrente antes de que ésta se lleve a cabo o se pare.

Para asegurar la existencia de una fila y protegerla contra actualizaciones concurrentes, se debería usar SELECT FOR UPDATE o una declaración de tipo LOCK TABLE más apropiada. Esto debe tenerse en cuenta cuando desde otros entornos se estén portando aplicaciones hacia PostgreSQL utilizando el modo “serializable”.

4.9.6. Bloqueos e índices.

Aunque PostgreSQL proporciona acceso carente de bloqueos para lectura/escritura de datos en tablas, no ocurre así para cada método de acceso al índice implementado en PostgreSQL. Los diferentes tipos de índices son manejados de la siguiente manera:

Indices GiST y B-tree
Nivel de bloqueo de índice a nivel de página del tipo compartición/exclusividad para acceso de lectura/escritura. El bloqueo tiene lugar inmediatamente después de que las filas del índice hayan sido insertadas o accedidas. Estos tipos de índices proporcionan la mejor concurrencia sin bloqueos.

Indices hash

Se usa el bloqueo a nivel bloque desordenado para acceso lectura/escritura. El bloqueo tiene lugar después de que el bloque completo haya sido procesado. Los bloqueos a nivel de bloque producen mejor concurrencia que los bloqueos a nivel de índice pero pueden provocar "*deadlocks*".

R-tree
Se usan bloqueos a nivel de índice del tipo compartición/exclusividad en los accesos de lectura/escritura. Los bloqueos se llevan a cabo inmediatamente después de que se complete la totalidad del comando.

4.10. HERRAMIENTAS DE ADMINISTRACIÓN.

En este capítulo se van a describir las herramientas que ofrece PostgreSQL para realizar tareas de administración de una base de datos. Echando un simple vistazo al sistema, se ve que PostgreSQL carece de alguna herramienta para hacer tareas de administración a través de alguna interfaz gráfica más cómoda de utilizar de cara al usuario. Existen algunas herramientas desarrolladas por terceras partes, como por ejemplo la aplicación *phpMyAdmin*, pero no dejan de ser interfaces Web para enviar consultas al *postmaster*, nada que ver con algunas herramientas mas avanzadas que ofrecen otros sistemas como MySQL y Oracle. Todas las tareas de administración que se quieran hacer en PostgreSQL se deberán hacer a través de sentencias SQL en el monitor interactivo *psql* o a través de programas ejecutables desde el *shell* de Linux.

4.10.1. Entorno de tiempo de ejecución.

En este apartado se detalla la interacción entre PostgreSQL y el sistema operativo.

4.10.1.1. Utilizando PostgreSQL desde Linux.

Todas las órdenes de PostgreSQL que se ejecutan directamente desde un *shell* de Linux se encuentran en el directorio ".../bin". Incluir este directorio en la variable de entorno *path* facilitará mucho la ejecución de los mismos.

Existe una colección de catálogos del sistema en cada servidor. Dicha colección incluye una clase (*pg_user*) que contiene una instancia para cada usuario válido en PostgreSQL. La instancia especifica un conjunto de privilegios sobre PostgreSQL, como la posibilidad de actuar como superusuario, la posibilidad de crear/destruir bases de datos y la posibilidad de actualizar los catálogos del sistema. Un usuario de Linux no puede hacer nada con PostgreSQL hasta que se instale una instancia apropiada en dicha clase. Ya se vio en un apartado anterior cómo se realizaba el inicio y detención del servidor PostgreSQL. En los apartados siguientes se verán algunas de las opciones disponibles para configurar el servidor y algunas herramientas para administrar bases de datos.

4.10.1.2. El archivo *pg_options*.

El archivo data/pg_options contiene opciones usadas por el *backend* para controlar los mensajes de trazado y otros parámetros ajustables del mismo. El archivo se vuelve a leer cuando el *backend* recibe la señal SIGHUP, permitiendo cambiar las opciones de tiempo de ejecución al vuelo, sin que sea preciso reiniciar PostgreSQL (esto no sucede en otros sistemas gestores de bases de datos). En este archivo se pueden incluir opciones

de depuración usadas por el paquete de trazado (backend/utils/misc/trace.c) o parámetros numéricos usados por el *backend* para controlar su comportamiento. Todas las *pg_options* se inicializan al poner en marcha el *backend*. Si se añaden o se modifican opciones, serán leídas por todos los *backend* que se inicien a continuación. Para que cualquier cambio tome efecto en los *backend* que están activos, es preciso enviar una señal SIGHUP al *postmaster*, quien reenviará la señal a todos los *backends* activos. Se pueden activar los cambios para un *backend* específico enviándole directamente una señal SIGHUP.

Las *pg_options* pueden especificarse también con la opción -T del comando *postgres* de la siguiente forma:

postgres [opciones] -T "*lista_parametros_pg_options*"

El formato del archivo pg_options es como sigue:

```
# comentario
opción=valor_entero  # Establece el valor de  opción
opción               # establece opción = 1
opción+              # establece opción = 1
opción-              # establece opción = 0
```

La lista de opciones reconocidas se lista a en la tabla 4.32.

Opción	Descripción	Valores permitidos
all	Marca de traza global.	0) Mensajes de trazado activados individualmente. 1) Activar todos los mensajes de trazado. -1) Inhibir todos los mensajes de trazado.
verbose	Marca de nivel de depuración.	0) Sin mensajes, éste es el valor por omisión. 1) Escribir mensajes de información. 2) Escribir más mensajes de información.
query	Trazar peticiones.	0) No escribir la petición. 1) Escribir una versión condensada de la petición en una línea. 4) Escribir la consulta completa.
plan	Escribir el plan de consulta.	
parse	Escribir la salida del traductor de consultas.	
rewritten	Escribir la consulta rescrita.	
parserstats	Escribir las estadísticas del traductor de consultas.	
plannerstats	Escribir las estadísticas del planificador.	
executorstats	Escribir las estadísticas de ejecución.	
shortlocks	De momento no se usa, pero se precisa para habilitar nuevas características en el futuro.	
locks	Trazar bloqueos.	
userlocks	Trazar bloqueos de usuario.	

Opción	Descripción	Valores permitidos
spinlocks	Trazar '*spin locks*'.	
notify	Trazar funciones de notificación.	
malloc	Sin uso por el momento.	
palloc	Sin uso por el momento.	
lock_debug_oidmin	OID con mínimo parentesco trazado por los bloqueos.	
lock_debug_relid	OID, si no es cero, de parentesco trazado por los bloqueos.	
lock_read_priority	Sin uso por el momento.	
deadlock_timeout	Temporizador de comprobación de bloqueos circulares.	
syslog	Marca de *syslog*.	0) Mensajes a *stdout*/*stderr*. 1) Mensajes a *stdout*/*stderr* y *syslog*. 2) Mensajes solamente a *syslog*.
hostlookup	Habilitar la consulta de nombre de host en *ps_status*.	
showportnumber	Mostrar el número de puerto en *ps_status*.	
notifyunlock	Desbloqueo de *pg_listener* después de *notify*.	
notifyhack	Borrar *tuplas* duplicadas de *pg_listener*.	

Tabla 4.32. *Opciones de pg_options.*

4.10.2. Configuración del servidor.

Existen muchos parámetros de configuración que afectan al comportamiento de PostgreSQL. Todos los parámetros de configuración en PostgreSQL son sensibles a mayúsculas/minúsculas. Cada parámetro puede tomar un valor de tipo booleano, entero, punto flotante o cadena.

Un método para establecer estos parámetros del sistema es editar el fichero *postgresql.conf*, que normalmente se encuentra en el directorio de datos (*initdb* instala una copia por defecto en dicha localización). Un ejemplo de cómo es este fichero se muestra a continuación.

```
#  Comentario
log_connections = yes
log_destination = ’syslog’
search_path = ’$user, public’
```

En este fichero, se especifica un parámetro por línea. El signo “=” entre el nombre del parámetro y el valor es opcional.
El fichero de configuración es releído cuando el *postmaster* recibe una señal SIGHUP (el comando *pg_ctl reload* envía esta señal). El *postmaster* propaga esta señal a todos los procesos del servidor, de forma que todas las sesiones existentes ven esta nueva configuración. No obstante, algunos parámetros sólo pueden establecerse al inicio del servidor. Un segundo método para establecer estos parámetros de configuración es proporcionarlos a través de las opciones del comando *postmaster*. La forma es la siguiente:

```
postmaster -c parámetro1=valor1 parámetro2=valor2 …
```

También es posible asignar una serie de parámetros a un usuario determinado o a una base de datos en concreto. Cuando se comienza una sesión se cargan los parámetros por defecto definidos para el usuario y la base de datos con que se inicia la sesión. Para tal efecto se disponen de los comandos ALTER USER y ALTER DATABASE. Las opciones por base de datos invalidan las opciones recibidas por el comando *postmaster* o por el archivo de configuración, y estas son invalidadas por las opciones por usuario. Las opciones por base de datos y por usuario son invalidadas por las opciones por sesión.

Algunos parámetros de configuración se pueden cambiar también en sesiones individuales SQL mediante el comando SET. El comando SQL SHOW muestra una vista de los valores de todos los parámetros del sistema. La lista de parámetros configurables es muy amplia y no se entrará en detalle, simplemente comentar que se ofrecen parámetros para:

- Cambiar las localizaciones de ficheros empleados por el sistema.
- Configurar los parámetros reconexión, seguridad y autentificación.
- Establecer los recursos de memoria disponible.
- Configurar el "*Write-Ahead Logging*" (WAL), que es una utilidad empleada internamente por PostgreSQL para asegurar que los cambios en los ficheros de datos (donde residen las tablas e índices) sólo deben ser escritos después de que dichos cambios hayan quedado patentes en un fichero de registro.
- Establecer los parámetros de planificación que emplea el optimizador de consultas.
- Configurar los informes de errores y registros del sistema.
- Establecer los parámetros de configuración de las estadísticas en tiempo de ejecución.
- Configurar las tareas automáticas de limpieza y análisis de PostgreSQL, ejecutadas por el subproceso *autovacuum*.
- Establecer los parámetros por defecto de las conexiones de cliente.
- Configurar el manejo de bloqueos.
- Modificar parámetros referentes a la compatibilidad de versiones y plataformas.
- Ver los parámetros por defecto del sistema cuando este es compilado e instalado (esta serie de parámetros son de solo lectura).
- Configurar los módulos adicionales de PostgreSQL, como pueden ser, por ejemplo, los lenguajes de procedimiento.
- Configurar opciones relativas al trabajo con el código fuente de PostgreSQL y a la ayuda para la recuperación de bases de datos dañadas gravemente.

4.10.3. Roles y privilegios de bases de datos.

PostgreSQL gestiona los permisos de acceso a bases de datos usando el concepto de roles. Un rol puede ser entendido como un usuario, o como un grupo de usuarios, dependiendo de cómo se defina dicho rol. Los roles pueden ser propietarios de objetos de bases de datos (por ejemplo tablas) y pueden asignar privilegios en esos objetos a otros roles para controlar quién tiene acceso a qué objetos. Además, también es posible otorgar la condición de miembro de un rol a otro rol.

El concepto de rol absorbe los conceptos de usuarios y grupos. Cualquier rol puede actuar como un usuario, como un grupo o como ambos. Esta característica de roles está disponible a partir de la versión 8.1 de PostgreSQL.

4.10.3.1. Roles de bases de datos.

Los roles de bases de datos están conceptualmente separados de los usuarios del sistema operativo. Los roles tienen carácter global en todo el sistema de bases de datos (y no para bases de datos individuales). Para crear un rol, se dispone del siguiente comando SQL:

CREATE ROLE *nombre* [atributos];

Si se quiere eliminar un rol existente, se empleará el comando DROP de la siguiente forma:

DROP ROLE *nombre*;

PostgreSQL proporciona dos programas externos para crear usuarios y eliminar usuarios (en realidad roles). Estos programas se pueden ejecutar desde el shell de linux, y son los siguientes:

createuser *nombre*
dropuser *nombre*

Los roles existentes se almacenan en una tabla del catálogo del sistema, denominada *pg_roles*. Por defecto, en PostgreSQL existe un usuario que se crea con su instalación. Este es el superusuario, y por defecto tendrá el mismo nombre que el nombre de usuario del sistema operativo que inicializo el espacio de bases de datos (a través de la herramienta *initdb*).

Cada conexión a una base de datos se hace con el nombre de un rol en particular, y este rol determina los privilegios de acceso iniciales en esa conexión. Por ejemplo, en el monitor *psql* se puede emplear la opción -U para indicar el rol con el que se procederá a la conexión de la base de datos.

4.10.3.2. Atributos de roles.

Un rol en una base de datos tiene una serie de atributos que definen sus privilegios e interactúan con el sistema de autentificación de cliente. Estos atributos se muestran en la tabla 4.33.

Atributo	Descripción
LOGIN	Sólo los roles que tengan este atributo pueden usarse como rol inicial en una conexión. Puede ser considerado como un usuario de la base de datos.
SUPERUSER	Este atributo hace que el rol se salte todas las verificaciones de permisos. Sólo un superusuario puede crear un rol que a su vez tenga el atributo de superusuario.
CREATEDB	Permite al rol crear bases de datos.
CREATEROLE	Permite al rol crear otros roles, modificarlos o eliminarlos (excepto los roles que tienen atributo de superusuario).

Atributo	Descripción
PASSWORD '*cadena*'	Especifica que el rol posee una clave para conectarse al sistema. Los *passwords* de usuario se almacenan en el fichero del sistema *pgpass.conf*.
INHERIT	El rol hereda los privilegios de un grupo si se añade al mismo.

Tabla 4.33. *Atributos de roles en PostgreSQL.*

Para modificar los atributos de un rol ya existente, se dispone del comando ALTER ROLE.

4.10.3.3. Privilegios.

Los privilegios definen lo que un rol puede hacer con un objeto determinado del sistema. Para crear/modificar/eliminar privilegios de un rol se dispone de los comandos GRANT y REVOKE. Estos comandos ya se explicaron en un apartado anterior.

4.10.3.4. Miembros de un rol.

Una práctica muy conveniente es agrupar usuarios para facilitar la gestión de privilegios. En PostgreSQL esto se consigue creando un rol que represente al grupo. Para establecer un rol de grupo, primero se ha de crear el rol. Una vez creado, se le pueden añadir o eliminar usuarios usando los comandos GRANT y REVOKE. La sintaxis para hacer esto es la siguiente.

GRANT *nombre_rol_grupo* TO *rol1*, …;
REVOKE *nombre_rol_grupo* TO *rol1*, …;

Se pueden incluir al grupo roles de grupo, también, pero con la única restricción de que no se pueden crear bucles circulares (cuando un rol de grupo añade a otro rol que contiene al primer rol de grupo).

Los miembros de un rol pueden usar los privilegios del rol de grupo de dos formas. Primero, cada miembro de un grupo puede utilizar explícitamente el comando SET ROLE para "llegar a ser" temporalmente el rol de grupo. En este estado, la sesión tiene acceso a los privilegios del rol de grupo en vez de a los del rol original, y cualquier objeto creado se considera perteneciente al rol de grupo, y no al rol que originalmente inició la sesión. Segundo, los roles miembros de un rol de grupo que tienen el atributo INHERIT automáticamente tienen uso de los privilegios de los roles de los cuales son miembro.

Para eliminar un rol de grupo, se debe usar el comando DROP ROLE. Con este comando todos los privilegios pertenecientes al rol de grupo son revocados, pero los roles miembros no se ven afectados individualmente, sino que simplemente dejan de pertenecer al rol de grupo.

4.10.3.5. Funciones y disparadores.

Las funciones y los disparadores permiten a los usuarios insertar código en el *backend* del servidor. Es posible que otros usuarios puedan usar este código inintencionadamente. Esto puede representar un agujero en la seguridad del sistema. La única protección real contra esto es tener un control apropiado sobre quién define las funciones. Las

funciones se ejecutan en el *backend* con los permisos de sistema operativo del servidor. Si el lenguaje de programación usado para la función permite accesos a memoria no controlados, es posible cambiar las estructuras de datos internas del servidor. Para controlar un poco este hecho, PostgreSQL sólo permite a los superusuarios crear funciones escritas en esos lenguajes.

4.10.4. Gestión de bases de datos.

Cada instancia de un servidor PostgreSQL gestiona una o mas bases de datos. Una base de datos es una colección de objetos. Generalmente, cada objeto de una base de datos pertenece a una única base de datos, aunque existen una serie de objetos del sistema, como por ejemplo la tabla *pg_database* (que contiene información sobre las bases de datos existentes en el sistema), que pertenecen a todo el espacio de bases de datos y son accesibles desde cualquier base de datos dentro de este espacio. Ya se vio en un apartado anterior la estructura de las bases de datos en PostgreSQL.

Cuando un cliente se conecta a una base de datos, debe especificar en su solicitud de conexión el nombre de la base de datos a la que se quiere conectar. No es posible acceder a más de una base de datos por conexión, pero una aplicación no tiene ningún tipo de restricción en cuanto al número de conexiones que establece a la misma u otra base de datos).

4.10.4.1. Creación de bases de datos.

Como se comento anteriormente, para crear una nueva base de datos PostgreSQL dispone del programa *createdb* ejecutable desde el *shell* de Linux. También se pueden crear bases de datos desde el monitor *psql* con la instrucción SQL CREATE DATABASE.

PostgreSQL también ofrece la posibilidad de crear bases de datos para otros usuarios (pertenecientes a un rol del cual el rol actual no es miembro). Para ello se deberá especificar la opción *-O nombre_rol* en el programa *createdb* o añadir la cláusula OWNER *nombre_rol* a la instrucción CREATE DATABASE. Sólo los superusuarios pueden crear bases de datos para algún otro.

4.10.4.2. Plantillas de bases de datos.

Cuando se crean nuevas bases de datos en PostgreSQL, se hacen a partir de una determinada plantilla. Por defecto la plantilla empleada es una base de datos del sistema llamada *template1*. Si se añade algún elemento a esta plantilla, las nuevas bases de datos que se creen contendrán dichos elementos añadidos. Existe una segunda plantilla estándar, llamada *template0*. Esta base de datos contiene los mismos datos que el contenido inicial de *template1*, con la diferencia de que esta plantilla no puede ser modificada. Si se quiere usar *template0* como plantilla se deberá especificar este hecho añadiendo la cláusula “TEMPLATE *template0*” a la instrucción SQL CREATE DATABASE o añadiendo la opción -T *template0* al programa *createdb*.
Es posible crear plantillas de bases de datos adicionales, y de hecho se pueden crear copias de bases de datos existentes en el sistema sustituyendo en las expresiones anteriores el nombre de la plantilla por el nombre de alguna base de datos existente en el sistema. PostgreSQL garantiza que esta base de datos que se va a duplicar no esta en

uso por parte de otra sesión en el comienzo de la copia, pero no hace lo mismo durante el proceso, por lo que puede resultar en una base de datos que no sea una copia exacta de la original. Por tanto, se recomienda que las bases de datos que se usen como plantillas sean declaradas como bases de datos de solo lectura.

En la tabla *pg_database* existe una columna llamada *datistemplate* que sirve para especificar si una base de datos se puede usar como plantilla o no (un superusuario siempre la podrá utilizar como plantilla). También existe otra columna (*datallowconn*) que permite especificar si se permiten nuevas conexiones a la base de datos que se quiere usar como plantilla.

4.10.4.3. Configuración de una base de datos.

En un apartado anterior se vio que PostgreSQL dispone de varias variables de configuración para determinar el funcionamiento del sistema. Algunos de estor valores se pueden especificar para una base de datos en concreto. Para ello, se puede emplear el siguiente comando:

ALTER DATABASE *nombre_bd* SET *var* TO *valor*;

Esta instrucción no modificará el parámetro inmediatamente, sino que tendrá valor en la siguiente conexión al servidor. Si se quiere devolver el valor de un parámetro a su valor por defecto se podrá emplear también la sentencia ALTER DATABASE de la siguiente forma:

ALTER DATABASE *nombre_bd* RESET *var*;

4.10.4.4. Eliminar bases de datos.

En un apartado anterior se vio que para eliminar una base de datos del sistema se podía emplear la utilidad *dropdb* desde el *shell* de Linux. Esta acción también se puede hacer con la instrucción SQL DROP DATABASE.

Hay que tener en cuenta que sólo el propietario de la base de datos o un superusuario pueden eliminar la misma. No se puede ejecutar el comando DROP DATABASE mientras se está conectado a la base de datos objetivo.

4.10.4.5. Tablespaces.

PostgreSQL proporciona *tablespaces*, que permiten a los administradores de bases de datos definir localizaciones en el sistema de ficheros donde los archivos que representan objetos de bases de datos se pueden almacenar. Una vez creado, un *tablespace* se puede referenciar por un nombre cuando se crean objetos en la base de datos. Esto permite a un administrador una mejor gestión del espacio en disco, pues parte de la información contenida en el sistema de bases de datos se puede almacenar en una localización diferente, como por ejemplo otra partición del sistema o un disco de acceso rápido. Para definir un *tablespace* se debe usar el siguiente comando SQL:

CREATE TABLESPACE *nombre* LOCATION ‘*ruta*’;

La ruta debe ser un directorio vacío existente propiedad del usuario del sistema PostgreSQL. Todos los objetos creados en este *tablespace* serán almacenados en este directorio.

Sólo los superusuarios pueden crear *tablespaces*, pero una vez creados se pueden asignar privilegios a los demás usuarios para hacer uso de él. A un *tablespace* se le pueden asignar tablas, índices o bases de datos enteras. Para ello en la sentencia SQL de creación del objeto de deberá añadir al final la cláusula TABLESPACE *nombre_espacio*. También se puede cambiar el valor del *tablespace* empleado por defecto, que está contenido en la variable del sistema *default_tablespace*. Para cambiar el valor de esta variable se puede emplear el comando SET de la siguiente forma:

SET default_tablespace = *nombre_espacio*;

Al instalar PostgreSQL e inicializarlo se crean dos *tablespaces*: *pg_global* y *pg_default*. El primero es el empleado por elementos compartidos del sistema, y el segundo es el *tablespace* por defecto para las bases de datos plantilla y el resto de bases de datos.
Para eliminar un *tablespace* se puede emplear el comando DROP TABLESPACE.
Si se quiere obtener información sobre los *tablespaces* existentes en el sistema se puede consultar la tabla del sistema *pg_tablespace*.

4.10.5. Localización y juegos de caracteres.

En este apartado se van a comentar las características de localización disponibles desde el punto de vista del administrador. PostgreSQL soporta la localización desde dos puntos de vista:

- Usando las características locales del sistema operativo para proporcionar formatos de números, ordenes para comparación, mensajes traducidos y otros aspectos.
- Proporcionando un número de juegos de caracteres definidos en el servidor de PostgreSQL, incluyendo juegos de caracteres multi-bytes, para proveer de almacenamiento de texto en toda clase de lenguajes; y proporcionando una traducción de los juegos de caracteres entre cliente y servidor.

4.10.5.1. Soporte local.

PostgreSQL permite respetar las preferencias culturales en lo que se refiere a alfabeto, orden, formato de números, etc. Para ello emplea los servicios locales estándar ISO C y POSIX proporcionados por el sistema operativo servidor.

El soporte local se inicializa automáticamente cuando se crea el espacio de bases de datos mediante *initdb*. Esta aplicación inicializa PostgreSQL con las características locales de su sistema operativo. Si se quiere inicializar el espacio de bases de datos con alguna otra configuración local se deberá llamar a *initdb* con la opción *--locale*. Algunas de las características locales deben ser fijas para siempre en un espacio de base de datos, pues estas características afectan al modo de ordenar índices, por ejemplo. Otras de estas características locales se pueden cambiar cuando PostgreSQL está ejecutándose sin más que actualizar el valor de una variable determinada. Estas variables se almacenan en el fichero *postgresql.conf*.

Nótese que el comportamiento local de un servidor está determinado por las variables de entorno vistas por el mismo, y no por el entorno de ningún cliente.
La configuración local influye sobre las siguientes características de PostgreSQL:

- Orden en las consultas que usan ORDER BY en datos textuales.
- El uso de índices con cláusulas LIKE.
- Las funciones *upper*, *lower* e *initcap*.
- La familia de funciones *to_char*.

El inconveniente que tiene el uso de características locales distintas de las de los estándares C o POSIX es su repercusión en el rendimiento del sistema.

4.10.5.2. Juegos de caracteres locales.

PostgreSQL permite el almacenamiento de texto en una gran variedad de juegos de caracteres, tanto multi-bytes como de un solo byte. Todos los juegos de caracteres pueden ser utilizados transparentemente a través del servidor. El juego de caracteres por defecto se selecciona cuando se inicializa PostgreSQL al ejecutar la aplicación *initdb*. Se puede emplear cualquiera de los disponibles con la opción -E de este comando. No obstante, cuando se crea una base de datos es posible definir el juego de caracteres que empleará de la siguiente forma:

createdb -E *juego_caracteres nombre_bd*
CREATE DATABASE *nombre_db* WITH ENCODING '*juego_caracteres*';

La codificación de cada base de datos se almacena en la tabla del sistema *pg_database*. La tabla 4.34 muestra los juegos de caracteres disponibles en PostgreSQL.

JUEGOS DE CARACTERES DISPONIBLES EN PostgreSQL			
BIG5	EUC_CN	EUC_JP	EUC_KR
EUC_TW	GB18030	GBK	ISO_8859_5
ISO_8859_6	ISO_8859_7	ISO_8859_8	JOHAB
KOI8	MULE_INTERNAL	SJIS	SQL_ASCII
UHC	UTF8	WIN866	WIN874
WIN1250	WIN1251	WIN1252	WIN1256
WIN1258			

Tabla 4.34. *Juegos de caracteres disponibles en PostgreSQL.*

Una característica muy interesante que proporciona PostgreSQL para el manejo de varios juegos de caracteres es la traducción entre cliente y servidor. PostgreSQL soporta conversión automática entre cliente y servidor para ciertos juegos de caracteres. La información sobre las conversiones se almacena en la tabla del sistema *pg_conversion*. Además de todas las conversiones predefinidas, PostgreSQL permite crear nuevas conversiones mediante el comando SQL CREATE CONVERSION.

4.10.6. Herramientas para mantenimiento rutinario.

PostgreSQL proporciona las herramientas adecuadas para realizar las tareas de rutina para el mantenimiento de una base de datos. Estas tareas son:

- Copias de seguridad.
- Vaciado de la base de datos.
- Gestión de los ficheros de registro.
- Reindexados.

Comparado con otros sistemas, PostgreSQL requiere menos atenciones de mantenimiento, lo cual puede resultar en una ventaja bastante evidente de cara al trabajo de un administrador de bases de datos.

4.10.6.1. Vaciado.

Las tareas de vaciado permiten recuperar espacio inservible para su uso común. Para ello PostgreSQL proporciona el comando VACUUM. Este comando se puede emplear para varias tareas:

- Para recuperar espacio en disco ocupado por filas actualizadas o borradas.
- Para actualizar las estadísticas usadas por el planificador de consultas de PostgreSQL.
- Para proteger el sistema de la pérdida de datos muy antiguos.

Desde la versión 7.2 de PostgreSQL se puede ejecutar el comando VACUUM en paralelo con las operaciones normales sobre una base de datos sin afectar severamente al rendimiento del sistema. Además, desde la versión 8.1 es posible automatizar el mecanismo de las operaciones de vaciado mediante un proceso paralelo llamado *autovacuum*.
Cuando se ejecuta en PostgreSQL una operación de borrado o actualización de una fila, dicha fila no es inmediatamente eliminada. Esto es necesario para disfrutar de los beneficios del control de concurrencia multi-versión. Por tanto, el espacio que ocupa la fila eliminada o la versión anterior de una fila actualizada debe ser reclamado para su nuevo uso. Esto se lleva a cabo a través del comando VACUUM. El comando VACUUM se puede utilizar de dos modos para conseguir este objetivo. La primera forma (indicando simplemente VACUUM) marca las filas obsoletas para una reutilización futura, pero no reclama ese espacio ocupado para uso inmediato. La segunda forma es mediante VACUUM FULL. Esta versión emplea un algoritmo más agresivo para reclamar el espacio obsoleto. Cualquier espacio que sea liberado por VACUUM FULL es inmediatamente devuelto al sistema operativo. Por lo general mientras se ejecuta el comando VACUUM FULL el rendimiento del sistema puede verse algo perjudicado, debido a una serie de bloqueos que implementa internamente.

El planificador de consultas de PostgreSQL depende de la información estadística acerca de los contenidos en las tablas. Estas estadísticas son generadas por el comando ANALYZE, que puede ser ejecutado directamente o bien como un paso opcional en la ejecución del comando VACUUM. Por tanto, en tablas que son frecuentemente actualizadas, es conveniente realizar una actualización de las estadísticas cada cierto periodo de tiempo, para así conseguir un buen funcionamiento del planificador de consultas, lo que resultará en una mejora del rendimiento del sistema.

La semántica de transacciones del control de concurrencia multi-versión de PostgreSQL se basa en la comparación de las identidades de transacción (XID). Una fila con un XID

insertado mayor que el XID de la transacción actual se considera como un elemento "futuro", y no debería ser visible para la transacción actual. El problema de esto es que las XID tienen un tamaño limitado (32 bits actualmente). Si se rebasa este número máximo, se vuelve a empezar de cero, y por tanto la transacción actual vera a todos los datos de la base de datos como "futuros", lo que conllevaría a la pérdida de una gran cantidad de datos. Para evitar esto se puede emplear la herramienta VACUUM. Dado el tamaño actual de los XID, habría que ejecutar VACUUM al menos una vez cada 4 billones de transacciones para que no ocurriese este problema. PostgreSQL se encarga de generar mensajes de alerta cuando el número de transacciones se acerca a este valor.

4.10.6.2. Reindexado.

En algunas ocasiones resulta útil reconstruir los índices periódicamente con el comando REINDEX. En versiones de PostgreSQL anteriores a la 7.4 el reindexado periódico era frecuentemente necesario para evitar desbordamientos de índice por falta de espacio reclamado en los índices B-Tree. Cualquier situación en la que los rangos de los índices cambiaran significativamente podría resultar en un desbordamiento, porque las páginas de índices correspondientes a los rangos que no se utilizan no eran reclamadas para nuevo uso. Como consecuencia, el tamaño del índice llegaba a ser mucho más grande que el conjunto de datos útiles.

A partir de PostgreSQL v7.1 las páginas de índices que se tornan vacías son reclamadas para su reutilización, con el consiguiente beneficio en cuanto a espacio inutilizado. No obstante esta característica no está del todo implementada para índices que no son del tipo B-Tree.

4.10.6.3. Gestión de ficheros de registro.

Siempre resulta una buena idea almacenar el fichero de registro del sistema, sobre todo cuando es necesaria una diagnosis para detectar algún problema. Sin embargo, este fichero tiende a ser muy voluminoso, y por lo tanto no se deseará almacenarlo indefinidamente. Entonces surge la necesidad de "rotar" de forma que se cree un nuevo fichero de registro y los anteriores sean eliminados cada cierto período de tiempo.
Si simplemente se redirecciona la salida de errores estándar del *postmaster* a un fichero de registro, el único modo de truncar este registro es con la detención y reinicio del servidor. En ocasiones, un administrador de bases de datos no puede permitirse el lujo de detener el servidor cada cierto periodo de tiempo. Una solución mejor para esto es enviar la salida de errores estándar a algún tipo de programa de rotación. Para ello PostgreSQL proporciona un programa nativo, que se puede emplear estableciendo el parámetro de configuración *redirect_stderr* a *true* en el fichero *postgresql.conf*. Esta aplicación gestiona la creación de un nuevo fichero de registro, pero no se encarga de eliminar los ficheros de registro anteriores. Esto se deberá realizar manualmente o estableciendo algún trabajo por lotes.

4.10.7. Copias de seguridad y restauración.

Todo buen sistema gestor de bases de datos debe incluir una serie de herramientas para hacer copias de seguridad de los datos almacenados en las bases de datos y de la estructura de las mismas, así como para recuperar estos datos previamente guardados en caso de un error grave del sistema. PostgreSQL no es menos, y proporciona sus propias

herramientas para hacer esto, si bien son herramientas que deben ejecutarse desde línea de comandos, al estilo de todas las utilidades de PostgreSQL, pues no se proporciona ninguna clase de interfaz gráfico para realizar estas tareas.

Existen tres enfoques diferentes en cuanto a la copia de seguridad de datos en PostgreSQL:

- SQL Dump.
- Copia de seguridad a nivel de ficheros del sistema.
- Copia de seguridad online.

4.10.7.1. SQL Dump.

La idea de SQL Dump es la de generar un fichero de texto que contenga todas las instrucciones SQL necesarias para reproducir exactamente el estado de una base de datos. Para ello, PostgreSQL proporciona el programa *pg_dump*. Este programa se puede ejecutar desde la línea de comandos de Linux, y su uso es como sigue:

pg_dump *nombre_db* > *fichero_salida*

Esta utilidad escribirá el resultado en un fichero de salida, que posteriormente se podrá emplear para reproducir en su totalidad la base de datos objetivo.

Las copias de seguridad realizadas con *pg_dump* son internamente consistentes, esto es, que cualquier actualización que se le haga a la base de datos mientras *pg_dump* se está ejecutando no constará en la copia de seguridad. El programa *pg_dump* no bloquea otras operaciones en la base de datos mientras de está ejecutando.

Una vez que se ha hecho la copia de seguridad de la base de datos, si fuese necesario recuperarla, debido, por ejemplo, a un error grave en el sistema, se puede emplear el propio monitor *psql* desde la línea de comandos de Linux. Así, la forma de recuperar una base de datos a partir de un fichero es la siguiente:

psql *nombre_db* < *nombre_fichero*

La base de datos *nombre_db* no se creará con la ejecución del programa, sino que se deberá haber creado previamente usando la plantilla template0. Además, no sólo se debe crear previamente la base de datos, sino que también tienen que existir los usuarios que son propietarios de algún objeto en la base de datos.

La aplicación *pg_dump* permite realizar copias de seguridad de alguna base de datos en concreto. Pero si lo que se desea es hacer una copia de seguridad de todo el sistema de base de datos se debe emplear una aplicación proporcionada por PostgreSQL denominada *pg_dumpall*.

pg_dumpall > *fichero_salida*

La instrucción anterior, al ser ejecutada desde la línea de comandos de Linux, realizará una copia de seguridad de todo el sistema, incluyendo información relativa a usuarios o

grupos. Para restaurar la copia de seguridad que se ha realizado mediante *pg_dumpall* se debe emplear el programa *psql* al igual que en el caso anterior.

psql -f *fichero_entrada* postgres

Para restaurar bases de datos con *pg_dumpall*, es necesario hacerlo como superusuario. Dado que PostgreSQL permite tablas de mayor tamaño que el permitido por el sistema de ficheros, puede resultar problemático el volcado de una tabla a un fichero, ya que el fichero resultante seguramente superará el tamaño máximo permitido. Como *pg_dump* escribe en la salida estándar, puede usar las herramientas Linux para sortear estos posibles problemas:

- Uso de volcados comprimidos.
- Usar *split*. El comando *split* permite dividir la salida en piezas con un tamaño aceptable para el sistema.
- Usando la opción -Fc en *pg_dump*. Si PostgreSQL fue instalado en un sistema que posea instalada la librería de compresión *zlib*, el proceso normal de salvaguarda comprimirá directamente los datos que escriba en el fichero de salida.

4.10.7.2. Copia de seguridad a nivel de ficheros del sistema.

Esta estrategia de salvaguarda de datos no hace uso de ninguna herramienta proporcionada por PostgreSQL, sino que se trata de realizar una copia directamente de los ficheros del sistema. Esto se puede hacer teniendo en cuenta dos restricciones:

- El servidor no debe estar en funcionamiento.
- No se pueden hacer copias de seguridad de ciertas tablas o bases de datos individualmente a partir de sus respectivos ficheros o directorios. Esto no funcionaría porque en estos ficheros o directorios no está contenida toda la información necesaria. El resto de la información necesaria se encuentra en los ficheros de registro de *commit* pg_clock/*, que contienen información sobre el estado de finalización de todas las transacciones. Una tabla únicamente es útil si va acompañada de esta información. Tampoco funcionaría el hacer la copia de la tabla y sus datos *pg_clock* asociados, porque esto interpretaría que el resto de tablas en el espacio de datos son inservibles.

Es por estos motivos por los que estos métodos sólo son útiles para hacer copias de seguridad de todo el sistema de bases de datos.

4.10.7.3. Copia de seguridad online.

Continuamente PostgreSQL mantiene un registro denominado *write ahead log* (WAL). Este registro se almacena en el subdirectorio *pg_xlog*, y describe cada cambio que se ha producido en los ficheros de datos de la base de datos. La funcionalidad principal de este registro es para motivos de recuperación del sistema en caso de errores graves. Si el sistema tiene un error grave, es posible restaurar las bases de datos “reproduciendo” las entradas de este registro desde el último punto de verificación. Sin embargo, la existencia de este registro hace posible el uso de una tercera estrategia para hacer copias de seguridad de bases de datos, combinando la estrategia de copia de seguridad a nivel

de fichero con la copia de seguridad de los ficheros WAL. Esta estrategia es más compleja de administrar que las anteriores, pero tiene sus beneficios:

- No se necesita una copia de seguridad perfectamente consistente en el punto de comienzo. Cualquier inconsistencia interna será reparada con la reproducción de los ficheros de registro.
- Se pueden concatenar largas secuencias de ficheros WAL, por lo tanto, copias de seguridad continuas se pueden conseguir sin más que salvaguardar los ficheros WAL. Esto permite que no se tenga que hacer la salvaguarda completa tan a menudo.
- La reproducción de la información contenida en los ficheros WAL no tiene por qué ser completada hasta el final. Esto permite la recuperación del sistema hasta un punto concreto.
- Si se alimenta continuamente con ficheros WAL otra máquina que ha sido cargada con la misma copia de seguridad, se tendrá un buen sistema “en espera”, pues en cualquier momento se puede volcar el segundo sistema para tener una copia casi actual de la base de datos.

Al igual que el método anterior, este método sólo soporta la restauración de todo el sistema de bases de datos. Es posible configurar el almacenamiento de los ficheros WAL para que se salven en una localización diferente. Para ello se puede emplear la variable *archive_command* presente en el archivo *postgresql.conf*.

La clave del proceso anterior está en establecer un fichero de comandos de recuperación que describa cómo y desde cuando se quiere recuperar. Para este cometido se puede emplear el fichero *recovery.conf.sample* (normalmente instalado en el directorio /share) como prototipo. La única cosa que hay que especificar obligatoriamente en este fichero es *restore_command*, que dice a PostgreSQL cómo volver a los ficheros WAL. El uso más simple de esta variable sería como sigue:

restore_command = ‘cp /mnt/server/archivedir/%f %p’

La línea anterior copiará los segmentos WAL previamente almacenados con el nombre %f desde /mnt/server/archivedir/ a la localización especificada por %p.

Este método conlleva una serie de inconvenientes:

- No funciona con índices Hash y R-tree en el sentido de que la recuperación no actualizará estos tipos de índices.
- Si se ejecuta el comando CREATE DATABASE mientras se está haciendo la copia de seguridad base y la plantilla que CREATE DATABASE copió se modifica mientras la copia de seguridad base aún está en curso, es posible que la recuperación del sistema propague estas modificaciones a la base de datos creada.
- Los comandos CREATE TABLESPACE se registran en los ficheros WAL con la ruta absoluta del *tablespace*, y serán replicados en caso de restauración con la misma ruta absoluta. Esto puede resultar no deseable si la restauración del sistema se está efectuando en una máquina diferente. Puede resultar peligroso incluso si la restauración se realiza en la misma máquina, pero en un nuevo directorio de datos.

4.10.8. Herramientas de monitorización.

PostgreSQL proporciona diversas herramientas para monitorizar la actividad de la base de datos y analizar el rendimiento del sistema. A estas herramientas se les pueden añadir las proporcionadas por Linux, como pueden ser *ps*, *top*, *iostat* y *vmsat*. Lo primero que se echa en falta es la existencia de un cliente gráfico que monitorice en tiempo real el estado del sistema, al estilo de MySQL Administrator en el sistema MySQL. Todas las herramientas que proporciona PostgreSQL están basadas en aplicaciones ejecutables desde línea de comandos, lo cual puede resultar una desventaja frente a otros sistemas gestores de bases de datos como Oracle o MySQL. A continuación se detallan algunas de estas utilidades disponibles.

4.10.8.1. EXPLAIN.

Para ver el rendimiento de una consulta individual, se puede emplear el comando de PostgreSQL EXPLAIN. Este comando se debe introducir desde el monitor *psql*. Lo que hace este comando es mostrar el plan de consultas del planificador de consultas ante una consulta determinada. Su sintaxis es la siguiente:

EXPLAIN <consuta SQL>

La estructura de un plan de consultas es en forma de árbol con nodos y hojas. En el nivel mas bajo están los nodos de búsqueda de tablas, que devuelven las filas de una tabla. Existen diferentes nodos de búsqueda para los diferentes métodos de acceso a las tablas: búsquedas secuenciales, búsquedas indexadas, y búsquedas indexadas por mapa de bits. Si la consulta requiere uniones, agregaciones, ordenado o cualquier otra operación sobre las filas se crearán nodos a otro nivel para realizarlas. La salida de EXPLAIN tiene una línea por cada nodo del plan de consultas, mostrando el tipo de nodo básico más los costos que estima el planificador para la ejecución del nodo. La primera línea (nodo superior) muestra el costo total estimado de ejecución de la consulta. Es éste tiempo el que el planificador trata de minimizar. Los costos se miden en unidades de páginas de disco procesadas, es decir, un valor de 1.0 equivale a una lectura secuencial de página de disco. Cabe notar que el costo de un nodo superior incluye los costos de todos sus nodos hijos. Esto se entenderá mejor con un ejemplo.

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2

                                        QUERY PLAN
------------------------------------------------------------------------------------
Nested Loop (cost=2.37..553.11 rows=106 width=488)
        -> Bitmap Heap Scan on tenk1 t1 (cost=2.37..232.35 rows=106 width=244)
                Recheck Cond: (unique1 < 100)
                -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..2.37 rows=106 width=0)
                        Index Cond: (unique1 < 100)
        -> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.00..3.01 rows=1 width=244)
                Index Cond: ("outer".unique2 = t2.unique2)
```

4.10.8.2. El recopilador de estadísticas.

El recopilador de estadísticas de PostgreSQL es un subsistema que se encarga de la recolección y presentación de información sobre la actividad del servidor. Para poder utilizar esta herramienta, debe configurarse el parámetro de configuración *stats_start_collector* del fichero *postgresql.conf*. Para ello habrá que especificarle un valor de *true*. Por defecto, esta característica se encuentra activada, y no puede ser cambiada mientras el servidor se está ejecutando. Además, este subsistema afecta muy levemente al rendimiento del servidor.

Los parámetros *stats_command_string*, *stats_block_level* y *stats_row_level* del fichero *postgresql.conf* controlan qué cantidad de información se envía al recopilador. Los parámetros anteriores determinan respectivamente si un proceso servidor envía su cadena de comando actual, estadísticas de acceso a nivel de bloqueo de disco y estadísticas de acceso a nivel de fila al recopilador. Normalmente, estos parámetros están activados para todos los subprocesos, pero es posible desactivarlos para una sesión concreta utilizando el comando SET.

Para monitorizar las estadísticas existen varias vistas predefinidas. No obstante, se pueden construir otras vistas personalizadas. Un aspecto que se debe tener en cuenta es que la información mostrada por el monitor no es actualizada instantáneamente. Cada proceso servidor transmite información al recopilador justo antes de volver al estado de espera, de forma que una consulta o transacción que aún continúe en proceso no afecta al resultado mostrado. Lo que ocurre es que el colector emite un informe de las estadísticas cada cierto numero de milisegundos definido por la variable PGSTAT_STAT_INTERVAL (por defecto tiene un valor de 500 milisegundos). Esto provoca que la información mostrada pueda tener un cierto retraso respecto a la situación real del servidor en cierto instante.

Cuando se solicita a un proceso servidor que muestre la información del recopilador, éste coge los últimos resultados obtenidos por el recopilador para utilizarlos con las vistas o funciones que sean oportunas. Las vistas disponibles para monitorizar las estadísticas se muestran en la tabla 4.35.

Nombre	Descripción
pg_stat_activity	Una fila por proceso, mostrando OID y nombre de la base de datos, Id del proceso, Id y nombre del usuario, consulta actual, tiempo que lleva ejecutándose la consulta actual, hora a la que comenzó el proceso, dirección del cliente y número de puerto. Estas columnas se ven como NULL si no se examinan como superusuario o como el usuario que propietario del proceso que se está reportando.
pg_stat_database	Una fila por cada base de datos, mostrando OID y nombre de la base de datos, número de procesos activos conectados a esa base de datos, número de transacciones completadas y canceladas en esa base de datos, lecturas de bloque de disco totales y accesos al buffer.
pg_stat_all_tables	Para cada tabla en la base de datos actual, muestra el OID de la tabla, el nombre del esquema y el nombre de la tabla, el número de búsquedas secuenciales iniciadas, el número de filas obtenidas por búsquedas secuenciales, número de búsquedas indexadas iniciadas (sobre índices pertenecientes a la tabla), número de filas obtenidas en consultas indexadas y número de inserciones, actualizaciones y borrados de fila.
pg_stat_sys_tables	Igual que *pg_stat_all_tables*, pero sólo muestra las tablas del sistema.

Nombre	Descripción
pg_stat_user_tables	Igual que *pg_stat_all_tables*, pero sólo muestra las tablas de usuario.
pg_stat_all_indexes	Para cada índice en la base de datos actual muestra el OID de la tabla y el índice, nombres del esquema, tabla e índice, Número de búsquedas indexadas con ese índice, número de entradas devueltas por búsquedas de índice y número de filas alcanzadas en búsquedas indexada empleando dicho índice.
pg_stat_sys_indexes	Igual que *pg_stat_all_indexes*, pero sólo muestra los índices de tablas del sistema.
pg_stat_user_indexes	Igual que *pg_stat_all_indexes*, pero sólo muestra los índices de tablas de usuario.
pg_statio_all_tables	Para cada tabla en la base de datos actual muestra el OID de la tabla, el nombre de la tabla y de su esquema, número de bloques leídos desde esa tabla, numero de accesos al buffer, número de bloques leídos y accesos al buffer en todos los índices de esa tabla, etc.
pg_statio_sys_tables	Igual que *pg_statio_all_tables*, pero sólo se muestran las tablas del sistema.
pg_statio_user_tables	Igual que *pg_statio_all_tables*, pero sólo se muestran las tablas de usuario.
pg_statio_all_indexes	Para cada índice de la base de datos actual muestra el OID de la tabla e índice, el nombre de la tabla y el índice y el número de bloques de disco leídos y accesos al buffer en ese índice.
pg_statio_sys_indexes	Igual que *pg_statio_all_indexes*, sino que sólo se muestran las tablas del sistema.
pg_statio_user_indexes	Igual que *pg_statio_all_indexes*, sino que sólo se muestran las tablas de usuario.
pg_statio_all_sequences	Para cada objeto secuencia en la base de datos actual muestra el OID de la secuencia, el nombre del esquema y de la secuencia, y el número de bloques de disco leídos y accesos al buffer en esa secuencia.
pg_statio_sys_sequences	Actualmente no están definidas secuencias de sistema, por tanto esa vista siempre esta vacía.
pg_statio_user_sequences	Igual que *pg_statio_all_sequences*, sino que sólo se muestran las secuencias de usuarios.

Tabla 4.35. *Vistas de monitorización de estadísticas en PostgreSQL.*

Otro modo de consultar las estadísticas, a parte de las vistas anteriores, es escribir consultas que hagan uso de funciones que proporciona PostgreSQL para acceder al contenido de esas estadísticas. Estas funciones tienen como parámetro de entrada el OID de la base de datos o de la tabla de la que se quiera consultar la estadística. Las funciones que devuelven estadísticas del servidor reciben como parámetro de entrada el número de proceso. Todas estas funciones se muestran en la tabla 4.35.

Función	Descripción
pg_stat_get_db_numbackends(oid)	Número de procesos activos en una base de datos.
pg_stat_get_db_xact_commit(oid)	Transacciones completadas en una base de datos.
pg_stat_get_db_xact_rollback(oid)	Transacciones desechas en una base de datos.
pg_stat_get_db_blocks_fetched(oid)	Número de solicitudes de búsqueda de bloques de disco por base de datos.
pg_stat_get_db_blocks_hit(oid)	Número de solicitudes de búsqueda de bloques de disco encontradas en la caché por base de datos.
pg_stat_get_numscans(oid)	Número de búsquedas secuenciales hechas cuando el argumento es una tabla, o número de búsquedas por índice hechas cuando el argumento es un índice.
pg_stat_get_tuples_returned(oid)	Número de filas leídas en búsquedas secuenciales cuando el argumento es una tabla, o número de entradas de índice devueltas cuando el argumento es un índice.

Función	Descripción
pg_stat_get_tuples_fetched(oid)	Número de filas alcanzadas por búsquedas de mapas de bits cuando el argumento es una tabla, o número de filas alcanzadas por búsquedas indexadas cuando el argumento es un índice.
pg_stat_get_tuples_inserted(oid)	Número de filas insertadas en una tabla.
pg_stat_get_tuples_updated(oid)	Número de filas actualizadas en una tabla.
pg_stat_get_tuples_deleted(oid)	Número de filas eliminadas en una tabla.
pg_stat_get_blocks_fetched(oid)	Número de solicitudes de búsqueda de bloques de disco por tabla o índice.
pg_stat_get_blocks_hit(oid)	Número de solicitudes de bloques de disco encontradas en la caché para una tabla o índice.
pg_stat_get_backend_idset()	Lista de los números de proceso de los procesos activos en el servidor.
pg_backend_pid()	Id del proceso asociado a la sesión actual.
pg_stat_get_backend_pid(integer)	Id del proceso dado.
pg_stat_get_backend_dbid(integer)	Id de la base de datos del proceso dado.
pg_stat_get_backend_userid(integer)	Id de usuario del proceso dado.
pg_stat_get_backend_activity(integer)	Comando activo del proceso dado (NULL si el usuario que la llama no es superusuario ni el usuario de la sesión que se quiere consultar).
pg_stat_get_backend_activity_start(integer)	Devuelve la hora a la que empezó la consulta actual que ejecuta un proceso dado.
pg_stat_get_backend_start(integer)	Hora en la que se inició el proceso dado.
pg_stat_get_backend_client_addr(integer)	Dirección IP del cliente para un determinado proceso. Null si la conexión es a través de un *socket* Unix.
pg_stat_get_backend_client_port(integer)	Número de puerto del cliente para un determinado proceso. -1 si la conexión es por un *socket* Unix.
pg_stat_reset()	Resetea todas las estadísticas recogidas.

Tabla 4.35. *Funciones de monitorización de estadísticas en PostgreSQL.*

4.10.8.3. Monitorización de bloqueos.

Otra herramienta bastante útil para monitorizar la actividad de PostgreSQL es la tabla del sistema *pg_locks*. Esta tabla permite al administrador de una base de datos ver información relativa a los bloqueos en el sistema. Algunos ejemplos de uso de esta característica son:

- Ver todos los bloqueos actualmente implementados, todos los bloqueos que involucran a una base de datos determinada, todos los bloqueos en una relación particular o todos los bloqueos existentes en una sesión en particular.
- Determinar la relación de la base de datos actual con la mayoría de bloqueos no implementados.
- Determinar el efecto de conflictos entre bloqueos en el rendimiento de la base de datos.

4.10.8.4. Monitorización del uso de espacio en disco.

En PostgreSQL cada tabla tiene un fichero asociado donde se almacenan la mayoría de los datos. Si la tabla posee alguna columna que pueda contener valores potencialmente amplios, existe un fichero adicional asociado a dicha tabla que almacena los valores que son demasiado amplios para encajar confortablemente en la tabla principal. Puede haber

también índices asociados a esa tabla base. Cada tabla e índice se almacenan en un fichero del disco distinto (posiblemente mas de un fichero si ocupan mas de 1 Gigabyte). El espacio en disco del sistema se puede monitorizar de tres formas diferentes:

- Usando funciones SQL. Estas funciones se vieron en un apartado anterior.
- Usando información relativa al comando VACUUM.
- Desde la línea de comandos, empleando las herramientas existentes en contrib/oid2name.

De los tres métodos anteriores, el más sencillo para usar y generar informes acerca de tablas, tablas con índices, bases de datos, *tablespaces* y tablas con valores grandes es el uso de funciones SQL.

Usando *psql* en una base de datos en la que se ha ejecutado recientemente un comando VACUUM se pueden ejecutar consultas para ver el espacio en disco de cualquier tabla. Sólo habrá que consultar determinados campos de la tabla del sistema *pg_class*.

4.10.9. Aplicaciones.

Para terminar con el apartado dedicado a las tareas de administración en PostgreSQL se va a listar el juego de herramientas que proporciona PostgreSQL para realizar tareas administrativas. Estas herramientas se deben ejecutar desde la línea de comandos de Linux, y se dividen en aplicaciones cliente y aplicaciones servidor. A continuación se lista una descripción de todas las utilidades disponibles. Para más información sobre la utilización y las opciones de estas aplicaciones consulte la guía de referencia proporcionada por PostgreSQL.

Aplicaciones cliente

Aplicación	Descripción
clusterdb	Reorganiza las tablas en una base de datos PostgreSQL.
createdb	Crea una nueva base de datos PostgreSQL.
createlang	Añade un nuevo lenguaje de programación a una base de datos PostgreSQL.
createuser	Crea un nuevo usuario PostgreSQL.
dropdb	Borra una base de datos PostgreSQL existente.
droplang	Borra un lenguaje de programación de una base de datos PostgreSQL.
dropuser	Borra un usuario PostgreSQL.
ecpg	*Embedded SQL C preprocessor* (preprocesador de C incorporado en SQL).
pg_config	Proporciona información acerca de la versión instalada de PostgreSQL.
pg_dump	Realiza copias de seguridad de bases de datos PostgreSQL en un *script* u otro formato de archivo.
pg_dumpall	Extrae todas las bases de datos PostgreSQL en un archivo de *script*.
pg_restore	Restaura una base de datos PostgreSQL a partir de un archivo creado por *pg_dump*.
psql	Terminal interactivo de PostgreSQL.
vacuumdb	Limpia y analiza una base de datos PostgreSQL.

Tabla 4.36. *Aplicaciones cliente en PostgreSQL.*

Aplicaciones servidor

Aplicación	Descripción
initdb	Crea un nuevo espacio de bases de datos.
initlocation	Crea un área de almacenamiento secundaria.
ipcclean	Limpia la memoria compartida y los semáforos de "*backends*" abortados.
pg_controldata	Muestra información de un espacio de base de datos.
pg_ctl	Inicia, detiene o reinicia el servidor PostgreSQL.
pg_passwd	Manipula el fichero plano de *passwords*.
pg_resetxlog	Resetea los ficheros de registro y otra información de control de PostgreSQL.
pg_upgrade	Permite la actualización de una versión anterior sin tener que volver a recargar los datos.
postgres	Ejecuta un proceso PostgreSQL de usuario único.
postmaster	Ejecuta el servidor (*backend*) multiusuario de PostgreSQL.

Tabla 4.37. *Aplicaciones servidor en PostgreSQL.*

4.11. SEGURIDAD.

Los sistemas de seguridad de PostgreSQL se pueden dividir en dos componentes. Una parte es la restricción de acceso para ciertos usuarios a determinados objetos. Estas configuraciones son válidas para objetos dentro de una base de datos. La segunda parte tiene que ver con las restricciones de acceso global y la autentificación de usuarios.

Con la ayuda del sistema operativo, es posible incluso alcanzar mayores niveles de seguridad que los proporcionados con estos dos niveles usando los métodos de seguridad de red comúnmente conocidos, tales como Netfilter y SSH.

En este apartado se estudiarán los aspectos de seguridad de PostgreSQL. El primero de los dos componentes de seguridad comentados al principio de este capítulo se gestiona, como ya se vio en un apartado anterior, con los comandos GRANT y REVOKE, otorgando o eliminando determinados privilegios sobre los objetos a los usuarios de la base de datos. Aquí se hará especial énfasis en el segundo de los componentes de seguridad.

Cuando se trabaja con más de un usuario, el proceso de autentificación es un componente esencial de cualquier aplicación. PostgreSQL proporciona varios métodos para negociar con el usuario eficientemente.

Cuando un usuario intenta conectarse a una base de datos, PostgreSQL verifica si el usuario tiene permiso para conectarse y a qué objetos tiene acceso.

PostgreSQL ofrece varios tipos de autentificación de cliente. El método empleado para la autentificación de una determinada conexión cliente se puede seleccionar partiendo de la terna host, base de datos y usuario.

Para configurar el sistema de autentificación de cliente, se debe utilizar un fichero del sistema denominado *pg_hba.conf*, que se encuentra en el directorio $PGDATA de la máquina servidor. Este fichero contiene información acerca de qué *hosts* se conectan a qué bases de datos. Cada vez que un usuario intenta conectarse a una base de datos, este fichero es leído. Esto puede resultar bastante conveniente, pues no es necesario reiniciar el servidor PostgreSQL si se desea realizar alguna modificación en la configuración de la autentificación.

En cuanto al contenido del fichero *pg_hba.conf*, se pueden encontrar tres tipos de líneas:

- Comentarios. Las líneas de comentario empiezan por #.
- Vacías. Las líneas vacas se ignoran.
- Registros. Contienen un dato real de configuración.

Los registros contienen una serie de campos que pueden ser separados por espacios o tabulaciones. Todos los espacios al inicio o al final de la línea se ignoran. Cada registro sólo puede ocupar una línea. De esta forma, un registro podrá tener uno de los siguientes formatos:

local	*nombrebd*	*usuario*	*metodo*	[opciones]		
host	*nombrebd*	*usuario*	*dirección-CIDR*	*metodo*	[opciones]	
hostssl	*nombrebd*	*usuario*	*dirección-CIDR*	*metodo*	[opciones]	
hostnossl	*nombrebd*	*usuario*	*dirección-CIDR*	*metodo*	[opciones]	
host	*nombrebd*	*usuario*	*dirección-IP*	*mascara-IP*	*metodo*	[opciones]
hostssl	*nombrebd*	*usuario*	*dirección-IP*	*mascara-IP*	*metodo*	[opciones]
hostnossl	*nombrebd*	*usuario*	*dirección-IP*	*mascara-IP*	*metodo*	[opciones]

PostgreSQL soporta tres tipos de registros:

- Host. Un registro host define un host que tiene permiso para conectarse a una base de datos. Esto funciona sólo cuando el servidor se inicia con la bandera -i habilitada. De otro modo, no se aceptarán conexiones vía TCP/IP.
- Hostssl. Este tipo de registro define un host que tiene permiso para conectarse a una base de datos empleando TCP/IP, pero sólo cuando la conexión se hace con encriptación SSL. Si se especifica Hostnossl el efecto será el contrario, es decir, sólo se permitirán las conexiones vía TCP/IP que no emplean SSL.
- Local. Este registro define una conexión permitida empleando *sockets* de Unix, es decir conexiones locales.

En cuanto al resto de parámetros que contiene cada registro, *nombrebd* especifica la base de datos a la que se tiene acceso (un valor de *all* especifica a todas las bases de datos), *usuario* especifica el nombre de usuario que tiene permitido el acceso (un valor *all* especifica cualquier usuario). Es posible introducir más de un usuario o base de datos, separados por comas. El parámetro *direccón-CIDR* especifica la dirección IP de la máquina cliente. Este parámetro contiene la dirección IP en notación decimal estándar y la longitud de la máscara con el formato *dirección_ip/longitud_máscara*. Las direcciones IP sólo se pueden especificar numéricamente, es decir, no son válidos nombres de host o de dominios. La longitud de la máscara indica el número de bits de mayor orden de la dirección IP que deben coincidir para que se garantice el acceso. Existe una alternativa para especificar las direcciones IP, y es escribiendo la dirección IP y la máscara en columnas diferentes (*dirección-IP* y *máscara-IP*).
En el parámetro *método* se especifica el método de autentificación empleado. PostgreSQL proporciona varios métodos:

- Trust. Permite la conexión incondicionalmente. Este método permite el acceso a cualquiera que pueda conectarse al servidor de base de datos PostgreSQL sin necesidad de *password*.
- Reject. Rechaza la conexión incondicionalmente. Se emplea para filtrar ciertos host dentro de un grupo.

- Md5. Solicita al cliente un *password* encriptado con md5 para la autentificación.
- Crypt. Solicita al cliente una clave encriptada mediante *cript*() para la autentificación. Actualmente se emplea md5 en lugar de *crypt*.
- Password. Solicita al cliente un *password* sin encriptar para la autentificación. Como el *password* se envía "tal cual" a través de la red, este método no se debería emplear en redes no fiables.
- Krb5. Emplea el método de autentificación Kerberos V5. Este método sólo está disponible para conexiones vía TCP/IP. Este método es un sistema de autentificación estándar bastante seguro, adecuado para sistemas distribuidos en una red pública.
- Ident. Obtiene el nombre de usuario de sistema operativo del cliente y verifica si el usuario tiene permiso para conectarse como el usuario de base de datos solicitado consultando un fichero de mapeo que lista los pares permitidos, y que se especifica en las opciones, después de la palabra clave *ident*.
- Pam. Autentifica empleando el servicio "Pluggable Authentication Method" (PAM) proporcionado por el sistema operativo. Este método es similar a la autentificación con *password*.

Las opciones disponibles para cada registro dependen del tipo de autentificación que contenga dicho registro. No se hará especial hincapié en estas opciones, remitiendo al lector a la guía de referencia de PostgreSQL para más información sobre las mismas.

Se ha visto que PostgreSQL proporciona seguridad a varios niveles y con diferentes mecanismos de autentificación. Además de esto, cabe incluir el soporte para conexiones seguras, como por ejemplo con SSL y SSH. El único inconveniente reside en que, al contener la información de autentificación en un fichero del sistema, se deben cuidar muy escrupulosamente los permisos asociados a este fichero. Todo esto hace que PostgreSQL sea una gran solución para bases de datos de red. Sin embargo, esta lejos de ser un sistema de usuarios y grupos perfecto.

4.12. INTERFACES.

PostgreSQL incluye muy pocos interfaces de programación con su distribución base. Por defecto, sólo vienen incluidos *libpq* y *ecpg*.

libpq es la interfaz de programación en C para PostgreSQL. Consiste en una serie de librerías que permiten a los programas clientes escritos en C enviar consultas al *backend* PostgreSQL y recibir los resultados de esas consultas. Además, *libpq* constituye la base para el soporte de otros interfaces de programación. Como la mayoría de los paquetes de software en los sistemas Linux, PostgreSQL está escrito en C.

ecpg (embedded SQL C Preprocessor) consiste en un potente preprocesador de C disponible en PostgreSQL. La idea de incluir un preprocesador de C es tener una interfaz fácil y eficiente para generar código C optimizado. Los ficheros que contienen código ECPG se convierten a código C y pueden ser compilados como cualquier otro programa escrito en C. La ventaja de utilizar ECPG en vez del código C normal es que es mucho más fácil escribir código ECPG que un programa en C.

Como se puede comprobar, PostgreSQL incluye muy pocas interfaces de programación con la distribución base. Las librerías *libpq* se incluyen porque constituyen la interfaz

primaria de C, y porque muchas otras interfaces se construyen partiendo de ella. El preprocesador *ecpg* se incluye porque está bastante asociado a la gramática del servidor, de forma que es bastante dependiente de la versión del sistema gestor de bases de datos. Las otras interfaces de programación disponibles constituyen proyectos independientes a PostgreSQL, y deben ser instaladas por separado. Algunos de los interfaces de programación más populares son:

- psqlODBC. Es la interfaz más común para aplicaciones Windows. Con esta interfaz se pueden conectar bases de datos PostgreSQL con bases de datos de OpenOffice Base o de MS Access, de la misma forma que se explico anteriormente en el capítulo dedicado a MySQL.
- pgJDBC. Interfaz JDBC.
- Npgsql. Interfaz .Net para las aplicaciones de Windows mas recientes.
- libpqxx. Es una nueva interfaz de C++.
- libpq++. Interfaz de C++ más antigua.
- pgperl. Es una interfaz para Perl con una API similar a la de *libpq*.
- DBD-Pg. Interfaz Perl que usa la API DBD estándar.
- pgtclng. Una nueva versión de la interfaz Tcl.
- pgtcl. Versión original de la interfaz Tcl.
- PyGreSQL. Librerías para la interfaz Python.

PostgreSQL se diseñó desde un principio para ser bastante extensible. Por esta razón, todas las extensiones cargadas en el sistema pueden funcionar como características incluidas en el paquete de PostgreSQL, aunque están desarrolladas por terceras partes y no por el equipo de desarrollo de PostgreSQL.

4.13. LENGUAJES DE PROCEDIMIENTO.

PostgreSQL permite que las funciones de usuario sean escritas en otros lenguajes aparte de SQL y C. Estos otros lenguajes se llaman genéricamente lenguajes de procedimiento (PLs). Para una función escrita en un lenguaje de procedimiento, el servidor de base de datos no tiene conocimiento intrínseco de cómo interpretar el texto fuente de la función. De hecho, esta tarea se pasa a un manejador especial que conoce todos los detalles del lenguaje. Este manejador podría hacer por si mismo todo el trabajo de análisis sintáctico, ejecución, etc. O también podría servir como enlace entre PostgreSQL y una implementación existente de un lenguaje de programación. El manejador es en si mismo una función en lenguaje C compilada en un objeto compartido y cargada bajo demanda, al igual que cualquier otra función de C. Esta característica hace relativamente fácil implementar otros lenguajes de Procedimiento diferentes de los disponibles. Actualmente existen cuatro lenguajes de procedimiento disponibles en la distribución estándar de PostgreSQL, que son los siguientes:

- PL/pgSQL
- PL/Tcl
- PL/Perl
- PL/Python

Un lenguaje de procedimiento debe ser “instalado” en cada base de datos en la que se vaya a usar. Los lenguajes de procedimiento instalados en la base de datos template1

estarán disponibles en las bases de datos que se creen posteriormente, puesto que todas estas bases de datos heredarán las características de la plantilla template1. De esta forma, el administrador de bases de datos puede decidir qué lenguajes de procedimientos están disponibles en determinadas bases de datos.

Para los lenguajes proporcionados en la distribución estándar, sólo es necesario ejecutar el comando CREATE LANGUAGE *nombre_lenguaje* para instalar el lenguaje en la base de datos actual. Alternativamente el programa *createlang* puede utilizarse desde la línea de comandos de Linux de la siguiente forma:

createlang *nombre_lenguaje nombre_bd*

dónde *nombre_lenguaje* es el nombre del lenguaje que se quiere instalar y *nombre_bd* es la base de datos en la cual se quiere instalar.

Como se acaba de ver, PostgreSQL es un sistema gestor de bases de datos bastante versátil en cuanto al uso de lenguajes de procedimiento, que permiten crear objetos y funciones definidas por el usuario empleando varios lenguajes, hecho del que no todos los sistemas gestores de bases de datos pueden presumir.