PostgreSQL

Administración y explotación de sus bases de datos

Archivos complementarios para descarga

Colección epsilon

Sébastien LARDIERE

PostgreSQL

Administración y explotación de sus bases de datos

El administrador de las bases de datos, el técnico del sistema operativo y el desarrollador, encontrarán en este libro la información imprescindible para explotar de la mejor manera posible, todas las posibilidades de **PostgreSQL** (en su versión 10, en el momento de escribir este libro).

Los primeros capítulos del libro cubren la **instalación de PostgreSQL** en Windows y GNU/Linux, así como la preparación del **entorno de ejecución del servidor**. A continuación, el autor presenta las **aplicaciones cliente** que se pueden utilizar, los diferentes **argumentos de seguridad** y los diferentes aspectos de PostgreSQL relativos **al soporte del lenguaje SQL**. Los capítulos posteriores introducen la **programación en PostgreSQL** y detallan **la administración y explotación** (de la configuración del servidor para las diferentes tareas de explotación, pasando por las copias de seguridad). Un capítulo presenta las herramientas adicionales que enriquecen la utilización de PostgreSQL. Para terminar, el autor introduce el tema de la **replicación de los datos** entre diferentes servidores, estudiando la replicación física y lógica integrada en PostgreSQL, así como la herramienta **Slony**.

Hay elementos adicionales que se pueden descargar en esta página.

Los capítulos del libro:
Preámbulo – Instalación – Inicialización del sistema de archivos – Conexiones – Definición de los datos – Programación – Explotación – Herramientas – Replicación

---

Sébastien LARDIÈRE

Después de haber sido desarrollador web, formador, consultor independiente y administrador de bases de datos PostgreSQL, en la actualidad **Sébastien LARDIERE** es consultor PostgreSQL dentro de Loxodata, empresa de consultoría especializada en PostgreSQL. Su experiencia en PostgreSQL es reconocida y la ha compartido voluntariamente a través de las páginas de este libro, para el beneficio de todos sus lectores.

Preámbulo

Introducción

PostgreSQL es un servidor de bases de datos relacionales y objetos. Se trata de un software libre, distribuido bajo los términos de la licencia BSD, desde 1996.

El proyecto PostgreSQL, tal y como lo conocemos en la actualidad, tiene más de veinte años. Pero el proyecto POSTGRES empezó en 1986, dentro de la universidad de Berkeley, en California, bajo la dirección de Michael Stonebraker. Este proyecto es la continuación del SGBD Ingres, de donde viene el nombre (Post-Ingres). En 1994, se añade un intérprete SQL y el código fuente se distribuye bajo el nombre de Postgres95. El lenguaje SQL sustituye al lenguaje PostQUEL y en 1996 el proyecto se renombra como PostgreSQL.

PostgreSQL es uno de los servidores de bases de datos más avanzados entre los distribuidos como software libre. El término «avanzado» puede calificar tanto el conjunto de las funcionalidades presentes como la capacidad de escritura del código fuente e incluso la estabilidad y la calidad de ejecución del software.

Además del código fuente, es el conjunto del proyecto, la manera en la que se desarrolla, la manera en la que se toman las decisiones, así como el conjunto de los desarrolladores, lo que se puede calificar de abiertos. Se trata de uno de los puntos importantes que permiten juzgar la calidad del software: todo el código fuente, todas las decisiones que afectan a la adición de una funcionalidad o el rechazo de una modificación, están disponibles para todos los que lo deseen.

Se expone al conjunto de los desarrolladores, independientemente de su aportación y todo está sujeto a discusión. Cada punto se puede someter a consulta y se decide una respuesta.

Esto es lo que hace que PostgreSQL sea hoy día un software libre y abierto en todos los sentidos del término y que además sea uno de los servidores de bases de datos más avanzados. Varias decenas de desarrolladores, en todos los rincones del globo, contribuyen con PostgreSQL. Además, hay varias empresas que participan directamente en el desarrollo de PostgreSQL.

El conjunto de los actores se agrupa bajo el término PGDG, que significa PostgreSQL Developer Group, grupo informal de publicación del software. Existe un equipo central llamado «core team» y otro ampliado de «committers», que pueden publicar el código de PostgreSQL en el almacén GIT del proyecto.

El desarrollo de PostgreSQL se articula alrededor de «comités» regulares, con el objetivo de producir una versión principal de PostgreSQL cada año. Durante estos «comités», cualquier desarrollador puede ofrecer una nueva funcionalidad en forma de «patch» que se revisará y después se discutirá y, tras un consenso, se validará en el almacén de fuente GIT de PostgreSQL.

El conjunto del código fuente, la documentación y las listas de discusión están disponibles en el sitio web https://www.postgresql.org. Algunas versiones binarias, listas para ser instaladas, están disponibles también para los sistemas Windows. La mayor parte de las distribuciones GNU/Linux integran sus propios paquetes de software de PostgreSQL, así como los sistemas FreeBSD e incluso Sun Solaris.

Una «wiki» completa la documentación con numerosa información práctica, tanto de la vida del proyecto como de los casos prácticos de uso de PostgreSQL: https://wiki.postgresql.org/

El proyecto ofrece binarios de todas las versiones soportadas para los sistemas GNU/Linux Red Hat, CentOS, Debian y Ubuntu.

El proyecto soporta cada versión principal durante cinco años. Por lo tanto, aparecen regularmente versiones menores que contienen arreglos de errores o de seguridad. La versión más antigua soportada cuando se están escribiendo estas líneas es la versión 9.4.20, es decir, 20 versiones menores. Se soporta hasta noviembre de 2018. La versión actual es la 11. Con la versión 10 se ha introducido un cambio en la numeración de las versiones del software: con anterioridad, las versiones principales se representaban con dos cifras, por ejemplo: 9.6, después la tercera cifra representaba la versión menor, es decir, los arreglos de errores y de seguridad. Desde la versión 10, solo este identificador hace referencia a la versión principal; el número siguiente se corresponde con la publicación de los arreglos. Por lo tanto, la primera publicación de la versión actual es la 11.0. Después, para las próximas versiones principales de PostgreSQL, este identificador se incrementará: 12 para 2019, 13 para 2020 y así sucesivamente.

Muchas empresas ofrecen servicios relacionados con PostgreSQL. Se listan en el sitio web oficial: https://www.postgresql.org/support/professional_support/europe/

Presentación de los proyectos

Se puede asociar software al servidor PostgreSQL. Se unen en la granja de proyectos PgFoundry. Contribuyen a la calidad del servidor, poniendo a disposición de las herramientas gráficas controladores para el acceso a los servidores para diferentes lenguajes de programación, muchas extensiones y herramientas de terceros, que facilitan el mantenimiento y la administración.

El conjunto de los proyectos alojados por la granja de proyectos PgFoundry está disponible bajo licencia de software libre. Se puede acceder a esta granja de proyectos en la dirección http://www.pgfoundry.org. Algunos proyectos también se desarrollan en otro ámbito distinto al de PgFoundry.

Los que se utilizan en este libro están disponibles en las siguientes direcciones:

- DBeaver: https://dbeaver.jkiss.org/
- PgBouncer: https://wiki.postgresql.org/wiki/PgBouncer
- Londiste: https://wiki.postgresql.org/wiki/SkyTools

Objetivos de este libro

Este libro cubre la versión 10 del servidor y pretende ofrecer una guía para el administrador de bases de datos, dándole una visión de conjunto del funcionamiento del servidor, así como los detalles prácticos que debe conocer para el mantenimiento, administración y explotación diaria de PostgreSQL.

Algunos de los puntos mencionados permitirán al administrador entender el conjunto de las funcionalidades disponibles para los desarrolladores.

El capítulo Instalación cubre la instalación de PostgreSQL en los sistemas operativos Windows y GNU/Linux, utilizando los archivos fuente y los archivos binarios proporcionados.

El capítulo Inicialización del sistema de archivos cubre la preparación del entorno de ejecución del servidor, desde la inicialización del sistema de archivos hasta el arranque del servidor.

El capítulo Conexiones cubre las aplicaciones cliente, que se pueden utilizar por la persona que explota el sistema o por el administrador, así como los argumentos de seguridad para la apertura de conexiones desde estas aplicaciones cliente.

El capítulo Definición de los datos resume los diferentes aspectos de PostgreSQL relativos al soporte del lenguaje SQL.

El capítulo Programación está dedicado al tema de la programación alrededor de PostgreSQL, tanto desde el lado servidor con los procedimientos almacenados como del lado cliente.

El capítulo Explotación se refiere a las diferentes tareas de administración y explotación, desde la configuración del servidor hasta las diferentes tareas de explotación, pasando por las copias de seguridad.

El capítulo Herramientas presenta las herramientas que permiten al usuario, tanto administrador como desarrollador, conectarse a las instancias PostgreSQL para manipular los datos.

El capítulo Replicación introduce el tema de la replicación de los datos entre diferentes servidores, abordando las herramientas Slony y Londiste.

Límites de la herramienta

PostgreSQL es una herramienta dedicada al almacenamiento y tratamiento de datos. Una de las cuestiones importantes durante la elección de un sistema de administración de bases de datos es la referida al tamaño del volumen aceptado por el software. De manera general, PostgreSQL se basa extensamente en las capacidades del sistema operativo y, por lo tanto, no impone límites de volumen. Sin embargo, hay algunas excepciones:

- El tamaño de un campo está limitado a 1 gigabyte. En la práctica, este tamaño queda limitado por la cantidad de memoria RAM disponible para leer este campo.
- El tamaño de un registro está limitado a 1,6 terabytes.
- El tamaño de una tabla está limitado a 32 terabytes.
- El número máximo de columnas en una tabla puede ir desde 250 hasta 1600, según el tamaño de un bloque de datos y el tamaño de los tipos de datos utilizados para las columnas.

Una vez que se tienen en cuenta estas consideraciones, PostgreSQL no impone otros límites sobre el tamaño de una base de datos o la cantidad de registros en una tabla. En contraposición, el sistema operativo utilizado puede aportar sus propias limitaciones.

Instalación

Fuentes

Cada vez que sale una nueva versión de PostgreSQL, se entrega como archivos fuente. Se trata del modo de distribución por defecto, como para cualquier software libre. A partir de estos archivos fuente se crearán los paquetes binarios para Windows o GNU/Linux. En este caso, la operativa es relativamente sencilla, ya que las herramientas necesarias se instalan con antelación. Estas herramientas, disponibles en todas las distribuciones GNU/Linux, son:

- La herramienta GNU Make.
- Un compilador C ISO/ANSI (convendría una versión reciente de GCC).
- La herramienta tar, con gzip o bzip2.
- La librería GNU Readline.
- La librería de compresión zlib.

Además, se puede contemplar la posibilidad de algunas herramientas o instalaciones adicionales:

- Las herramientas MingW o Cygwin, para una compilación para un sistema Windows.
- Una instalación del software Perl, Python o Tcl para instalar los lenguajes de procedimientos almacenados PL/Perl, PL/python y PL/Tcl.
- La librería Gettext para activar el soporte de los lenguajes nativos.
- Kerberos, OpenSSL, Pam, si se prevé que se van a utilizar.

Como sucede habitualmente con otro tipo de software libre, la construcción de los binarios a partir de los archivos fuente se basa en un script `configure` que genera los archivos Makefile, los cuales contienen las instrucciones destinadas al compilador.

1. Descargar los archivos fuente

La primera etapa consiste en descargar un archivo con los archivos fuente desde el sitio web de PostgreSQL: https://www.postgresql.org/ftp/source/

La versión actual es la versión 11 y la última entrega en la actualidad es la 11.1. En los ejemplos de este libro se va a utilizar la versión estable 10.0. Los diferentes formatos de entrega están disponibles en el directorio v10.0. Los archivos están en formato tar, comprimidos con las herramientas gzip o bzip2. Por lo tanto, los archivos .tar.bz2 y .tar.gz tienen el mismo contenido. A cada archivo le corresponden los archivos .md5 y .sha256, que contienen las sumas de control que permiten verificar la calidad del contenido descargado.

Los siguientes comandos permiten verificar la suma md5 de un archivo, comparándola con el valor idéntico en el archivo .md5; ejemplo:

```
[user]$ cat postgresql-10.0.tar.bz2.md5
cc582bda3eda3763926e1de404709026  postgresql-10.0.tar.bz2
[user]$ md5sum postgresql-10.0.tar.bz2
cc582bda3eda3763926e1de404709026
```

Es importante que las sumas sean idénticas. En el caso contrario, la descarga del archivo probablemente haya sido defectuosa.

Una vez que se ha realizado la descarga del archivo, hay que abrirla con el comando `tar`. Las versiones recientes de tar detectan automáticamente la compresión utilizada; en el caso contrario, hay que indicarla:

```
[user]$ tar xf postgresql-10.0.tar.bz2
```

2. Elección de las opciones de compilación

Se crea un directorio `./postgresql-10.0/` que contiene principalmente un script `configure`, el cual va a permitir preparar la compilación. Hay disponibles diferentes opciones:

```
[user]$ cd postgresql-10.0/
[user]$./configure --help
```

Las opciones principales son:

- `--prefix`: permite indicar el directorio de instalación de PostgreSQL. El valor por defecto es /usr/local/pgsql.
- `--with-perl`, `--with-python`: activa el soporte de los procedimientos almacenados, respectivamente PL/Tcl, PL/Perl, PL/Python.
- `--with-pgport=PORTNUM`: define el puerto TCP: 5432 por defecto.
- `--with-blocksize=BLOCKSIZE`: define el tamaño de un bloque de datos, en KB: 8 KB por defecto.
- `--with-segsize=SEGSIZE`: define el tamaño de un segmento de la tabla, en GB: 1 GB por defecto.
- `--with-wal-blocksize=BLOCKSIZE`: define el tamaño de un bloque de WAL, en KB: 8 KB por defecto.
- `--with-wal-segsize=SEGSIZE`: define el tamaño de un archivo WAL, en MB, 16 MB por defecto.
- `--with-selinux`: activa el soporte de SELinux.
- `--without-readline`: desactiva el soporte de Readline.
- `--without-zlib`: desactiva el soporte de Zlib.
- `--with-gssapi`: activa el soporte de GSSAPI.
- `--with-ldap`: activa el soporte de LDAP.
- `--with-openssl`: activa el soporte de las conexiones seguras.
- `--with-icu`: activa el soporte de la librería ICU (Administración de las clasificaciones estandarizadas).
- `--enable-dtrace`: activa los marcadores para los sistemas de seguimiento Dtrace o SystemTap.
- `--enable-debug`: permite la depuración del software compilado.
- `--with-systemd`: activa el soporte de Systemd como sistema de arranque, útil con las versiones actuales de las distribuciones GNU/Linux.
- `--enable-nls`: activa el soporte de los lenguajes en el servidor.

Una vez que se ha realizado la elección de las opciones, hay que ejecutar el script `configure` con estas opciones para generar los archivos Makefile. Por ejemplo:

```
[user]$./configure --prefix=/usr/local/pgsql00
--with-python --with-openssl --enable-dtrace --enable-debug
--with-systemd
```

3. Compilación

Una vez que se validan las pruebas y se crean los archivos Makefile, el comando `make` interpreta estos archivos para lanzar la compilación propiamente dicha:

```
[user]$ make world
```

Después, el destino `install` de los archivos Makefile crea el directorio de instalación y ubica los archivos generados en la arborescencia:

```
[user]$ sudo make install-world
```

A partir de este momento, se instalan los comandos cliente, los archivos de librerías, las extensiones, la documentación y el servidor de PostgreSQL.

4. Etapas postinstalación

Para terminar la instalación, hay que guardar en las variables de entorno los directorios donde se instalan las librerías y los comandos de PostgreSQL. Esto va a permitir la utilización de los comandos en los scripts o directamente desde el intérprete de comandos:

```
[user]$ export PATH=$PATH:/usr/local/pgsql100/bin
[user]$ export LD_LIBRARY_PATH=/usr/local/pgsql100/lib
```

Los valores identificados tienen en cuenta el valor de la opción `--prefix` del script `configure`. Es suficiente con ubicar estos comandos en el archivo de configuración del intérprete para hacer que estos ajustes sean permanentes.

La siguiente etapa de la instalación es la creación de una cuenta de usuario sin permisos en el sistema operativo que será la encargada de arrancar el servidor PostgreSQL. No es necesario que esta cuenta se pueda conectar al sistema. Será la propietaria de todos los archivos de las bases de datos. Por lo tanto, es importante utilizar una cuenta dedicada a PostgreSQL para aislar estos archivos del resto de las cuentas sin premisos del sistema operativo. La cuenta de usuario se puede crear como sigue:

```
[root]# adduser --home=/data/postgresql postgres
```

Una etapa importante es la inicialización del grupo de bases de datos, que se detallará en el siguiente capítulo. Esta etapa crea todos los directorios y archivos necesarios para la ejecución de un servidor. Este comando se debe lanzar por el usuario de sistema postgres:

```
[root]# su - postgres
[postgres]$ mkdir -p 10/data
[postgres]$ initdb -D 10/data
```

El comando `initdb` se presenta en detalle en el capítulo Inicialización del sistema de archivos.

5. Integración en el sistema operativo

Para terminar, la última etapa es la integración del sistema operativo con el sistema de arranque, gracias al archivo de servicio disponible con la documentación de PostgreSQL. Es suficiente con renombrar el archivo `postgresql-10.servico`, por `postgresql`:

```
[Unit]
Description=PostgreSQL database server
Documentation=man:postgres(1)
[Service]
Type=notify
User=postgres
ExecStart=/usr/local/pgsql00/bin/postgres -D /data/postgresql/10/data
ExecReload=/bin/kill -HUP $MAINPID
KillMode=mixed
KillSignal=SIGINT
TimeoutSec=0

[Install]
WantedBy=multi-user.target
```

El contenido se adapta a los directorios de instalación seleccionados, así como al directorio de datos.

Una vez que se crea el archivo, hay que copiarlo al directorio de archivos de servicio. Después, hay que guardar el servicio con los siguientes comandos:

```
[root]# cp postgresql-10.service /usr/lib/systemd/user/
[root]# systemctl enable postgresql-10
```

Para terminar, hay que arrancar PostgreSQL:

```
[root]# systemd start postgresql-10
```

Después de estos comandos, el servidor PostgreSQL se debe arrancar de nuevo.

Linux: distribuciones Debian y Ubuntu

El proyecto Debian ofrece paquetes de PostgreSQL para su distribución, pero la versión más estable actual solo ofrece la versión 11.1 y en lo que respecta a la versión estable de Ubuntu LTS, ofrece la versión 9.5.

Las versiones estables de estas distribuciones tienen una duración de vida más larga que el periodo entre dos versiones de PostgreSQL, lo que no le permite integrar todas las versiones principales de PostgreSQL.

Los almacenes ofrecidos por el proyecto PostgreSQL proponen muchos otros paquetes en el ecosistema PostgreSQL y, por lo tanto, pueden ser útiles para desplegar herramientas satélite, tales como los administradores de conexiones o las herramientas de copias de seguridad o, simplemente, la parte cliente de PostgreSQL.

1. Almacén apt.postgresql.org

El proyecto PostgreSQL ofrece los paquetes binarios de las cinco versiones principales para las distribuciones Debian y Ubuntu.

El almacén ofrece la infraestructura de alojamiento de los paquetes, que permiten añadir solo la fuente del almacén al sistema en el que se debe instalar PostgreSQL.

Algunos paquetes se deben instalar antes de la instalación, para determinar la versión del sistema e instalar las claves numéricas, con el siguiente comando:

```
[root]# apt-get install wget ca-certificate lsb-release
```

El almacén se firma con una clave numérica que conviene descargar inicialmente con el siguiente comando:

```
[root]# wget --quiet -O
- https://www.postgresql.org/media/keys/ACCC4CF8.asc | apt-key add -
```

El siguiente comando permite determinar el nombre de la versión del sistema operativo actual:

```
[root]# lsb_release -cs
```

Se deben escribir las siguientes líneas en un archivo de almacenes, es decir, en el archivo principal `/etc/apt/sources s.list`, o en un archivo dedicado `/etc/apt/sources s.list.d/pgdg.list`:

```
deb http://apt.postgresql.org/pub/repos/apt/ misistema-pgdg main 10
deb-src http://apt.postgresql.org/pub/repos/apt/ misistema-pgdg main 10
```

La versión principal al final del registro se puede omitir, lo que permite la instalación de todas las versiones de PostgreSQL al mismo tiempo. Sea cual sea la versión principal elegida, el hecho de indicarlo en el archivo de almacén permite dar seguridad a la instalación, evitando confusiones de versiones, por ejemplo en las versiones de la librería `libpq`.

Una vez que se refresca la base de paquetes con el comando `apt-get update`, los paquetes están disponibles para la instalación. El siguiente comando permite preguntar la base de datos de las aplicaciones disponibles, filtrando en la expresión `^postgresql`.

El siguiente extracto resalta las diferentes versiones disponibles:

```
[root] # apt-cache search ^postgresql
postgresql-10 - object-relational SQL database, version 10 server
postgresql-client-10 - front-end programs for PostgreSQL 10
postgresql-contrib-10 - additional facilities for PostgreSQL
postgresql-doc-10 - documentation for the PostgreSQL database
management system
postgresql-server-dev-10 - development files for PostgreSQL 10
server-side programming
```

Los paquetes principales son:

- `postgresql-10`: contiene los programas servidor, las librerías de funciones, los archivos compartidos que sirven para inicializar las instancias y las extensiones proporcionadas por PostgreSQL.
- `postgresql-client-10`: contiene los programas cliente como `psql` o `pg_dump`.
- `postgresql-server-dev-10`: contiene los programas y archivos de encabezados, útiles para el desarrollo.

Hay extensiones disponibles como paquetes individuales; por ejemplo, `postgresql-10-ip4r`.

Hay scripts específicos de estas distribuciones Debian y Ubuntu, escritos y disponibles en los paquetes `postgresql-common` y `postgresql-client-common`. Estos scripts permiten administrar varias versiones diferentes de PostgreSQL en una misma máquina, así como varias instancias de una misma versión de PostgreSQL.

Para instalar la versión 10, es suficiente con ejecutar el siguiente comando:

```
[root]# apt-get install postgresql-10
```

El sistema de resolución de dependencias instala automáticamente los paquetes necesarios. Por ejemplo, el paquete `libpq5` contiene la librería que implementa el protocolo cliente/servidor. Se instala en esta etapa porque esta librería es necesaria para los paquetes solicitados.

Un grupo de bases de datos se crea automáticamente en el directorio `/var/lib/postgresql/10/main/`, con los archivos de configuración correspondientes en `/etc/postgresql/10/main/`. Se puede arrancar la instancia con el siguiente comando:

```
[root]# pg_ctlcluster 10 main start
```

En las versiones recientes de la distribución Debian GNU/Linux, el sistema de arranque por defecto es Systemd. El script pg_ctlcluster interactúa automáticamente con Systemd, y la instancia o las instancias PostgreSQL se controlan directamente desde SystemD con el siguiente comando:

```
[root]# systemctl start postgresql@10-main
```

2. Distribuciones RPM

La mayor parte de las distribuciones GNU/Linux que utilizan el formato RPM para sus paquetes integran una versión de PostgreSQL. Por lo tanto, lo habitual es poder instalar PostgreSQL directamente desde los sistemas multimedia de instalación de la distribución. Sin embargo, un proyecto alojado en la granja de proyecto PgFoundry se encarga de realizar los paquetes RPM para las versiones recientes de PostgreSQL.

La dirección del proyecto es: https://yum.postgresql.org/

Descarga

Se soportan las distribuciones Fedora y Red Hat, así como CentOS, Scientific Linux y Oracle Enterprise Linux. Las tres últimas derivan directamente de Red Hat.

Los binarios proporcionados por el grupo PostgreSQL están disponibles directamente en el sitio web, en la dirección https://yum.postgresql.org/repopackages.php. Según la versión de PostgreSQL deseada, por ejemplo la versión 10 para Red Hat Enterprise Server 7 para un procesador de la familia Intel 64 bits, un archivo RPM instala la configuración necesaria para YUM, haciendo disponibles en el sistema los diferentes paquetes de PostgreSQL. Es suficiente con descargar el archivo RPM a través del enlace de cada sistema o, más simplemente, copiar este enlace y pasarlo como argumento al comando `rpm`, como en el siguiente ejemplo:

```
[root]# rpm -ivh http://yum.postgresql.org/10/redhat/
rhel-7-x86_64/pgdg-centos10-10-1.noarch.rpm
```

Este paquete no instala PostgreSQL, pero sí el archivo de almacén para YUM, lo que permite a YUM instalar los paquetes PostgreSQL. En el ejemplo anterior, el archivo instalado es:

```
/etc/yum.repos.d/pgdg-10-centos.repo
```

Ahora, los paquetes de la versión PostgreSQL están disponibles para la instalación.

Descripción de los paquetes

La instalación de PostgreSQL se separa en varios paquetes para permitir seleccionar las funcionalidades útiles.

Por ejemplo, es posible instalar solo el software cliente y no el servidor o no instalar las extensiones.

Cada uno de los diferentes archivos se corresponde con una necesidad concreta, de tal manera que no es necesario instalarlos todos:

- `postgresql10`: este paquete contiene los comandos cliente necesarios para administrar un servidor local o remoto.
- `postgresql10-server`: este paquete contiene los archivos necesarios para que funcione un servidor PostgreSQL.
- `postgresql10-contrib`: este paquete contiene las contribuciones para añadir funcionalidades al servidor.
- `postgresql10-libs`: este paquete contiene las librerías compartidas utilizadas por el software cliente y el servidor.
- `postgresql10-devel`: este paquete contiene todos los archivos de encabezados y las librerías necesarias para el desarrollo de aplicaciones que interactúan directamente con PostgreSQL.
- `postgresql10-docs`: este paquete contiene la documentación en formato HTML.

Instalación y actualización

La instalación se realiza gracias al comando `yum`:

```
[root]# yum install postgresql10 postgresql10-server
postgresql10-contrib postgresql10-docs
```

En el caso de una nueva versión menor de PostgreSQL, es posible simplemente hacer una actualización utilizando el comando `yum`. Por ejemplo:

```
[root]# yum update
```

Atención: este método no permite realizar actualizaciones entre versiones principales de PostgreSQL, como las versiones 9.6 y 10. Solo funcionará con las versiones menores, como las versiones 10.0 y 10.1.

Arranque del servicio

Como muchas otras, la distribución Red Hat Linux utiliza el sistema de arranque SystemD. El archivo postgresql10-server instala un archivo de servicio `postgresql-10.service` en el directorio `/usr/lib/systemd/system/`, lo que permite controlar la instancia PostgreSQL desde SystemD.

Este archivo contiene la ruta a los datos, inicializada en la variable PGDATA. Durante la instalación, los datos no se crean. La ruta por defecto es `/var/lib/pgsql/10/data`. Se puede modificar editando el archivo de servicio. Los siguientes comandos permiten inicializar y arrancar el servidor:

```
[root]# /usr/pgsql-10/bin/postgresql-10-setup initdb
[root]# systemctl start postgresql-10
```

El argumento `initdb` permite inicializar los archivos del servidor PostgreSQL. Los detalles de este comando se explican en el siguiente capítulo.

El comando `systemctl` permite volver a arrancar, recargar y detener el servidor PostgreSQL, con las opciones respectivas `restart`, `reload` y `stop`.

Instalación en un sistema MS-Windows

La instalación en Windows se realiza gracias a las contribuciones proporcionadas por los actores externos del grupo de desarrollo de PostgreSQL. Si es posible compilar PostgreSQL en un sistema MS-Windows, el método más sencillo es utilizar una de estas contribuciones. El instalador gráfico proporcionado por EnterpriseDB permite obtener en pocas pantallas un servidor PostgreSQL plenamente funcional. Los binarios están disponibles para su descarga directamente en el sitio de EnterpriseDB. La empresa OpenSCG ofrece la distribución BigSQL, que permite instalar PostgreSQL y numerosas contribuciones para MS-Windows y GNU-Linux.

El instalador gráfico de EnterpriseDB también permite instalar contribuciones como PgBouncer o el controlador .Net Npgsql. OpenSCG y EnterpriseDB forman parte de las empresas que participan en el desarrollo de PostgreSQL.

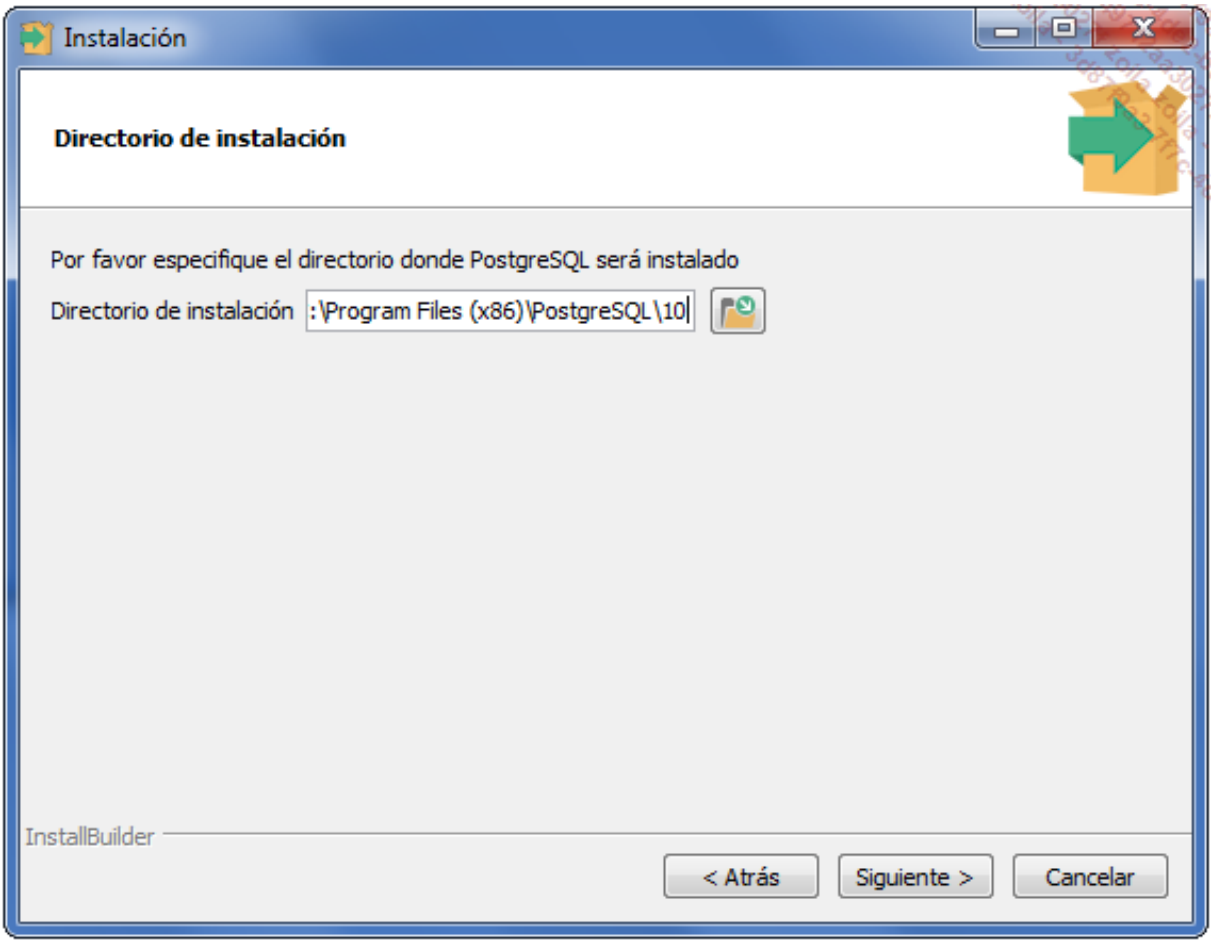
1. Descarga del instalador EnterpriseDB para MS-Windows

En la página https://www.enterprisedb.com/products-services-training/pgdownload y según la versión de PostgreSQL deseada, es suficiente con pulsar en el sistema operativo deseado, por ejemplo `Win x86-64`, para acceder a la descarga del instalador. El archivo que hay que descargar es un ejecutable con una extensión .exe.

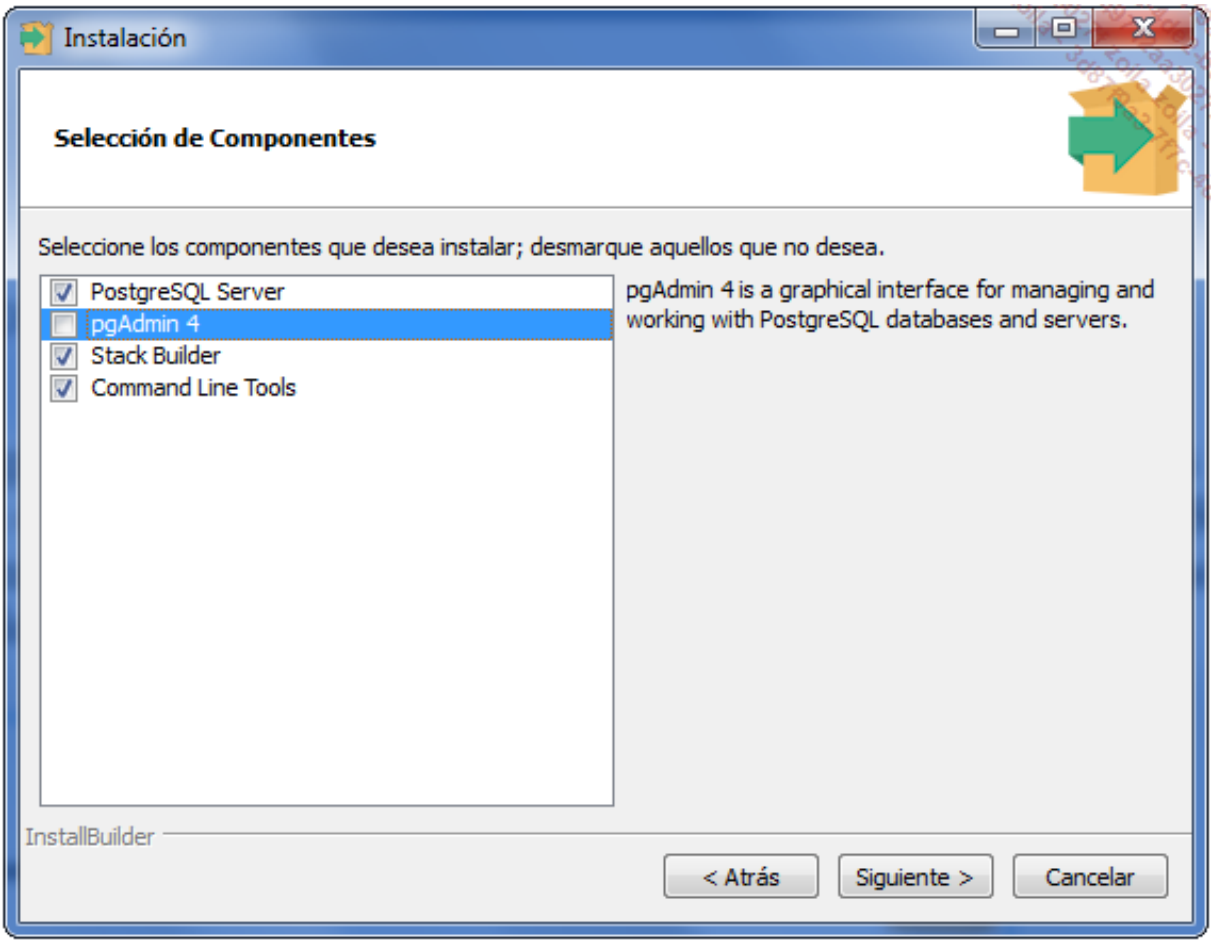
2. Instalación

Una vez que se descarga el ejecutable, es suficiente con hacer doble clic en el archivo para lanzar la instalación. Después de haber validado la ejecución (según los permisos del sistema), debe aparecer la primera pantalla. El instalador es muy clásico para un usuario acostumbrado a Windows. Es suficiente con pulsar en **Siguiente** para avanzar en el proceso de instalación.

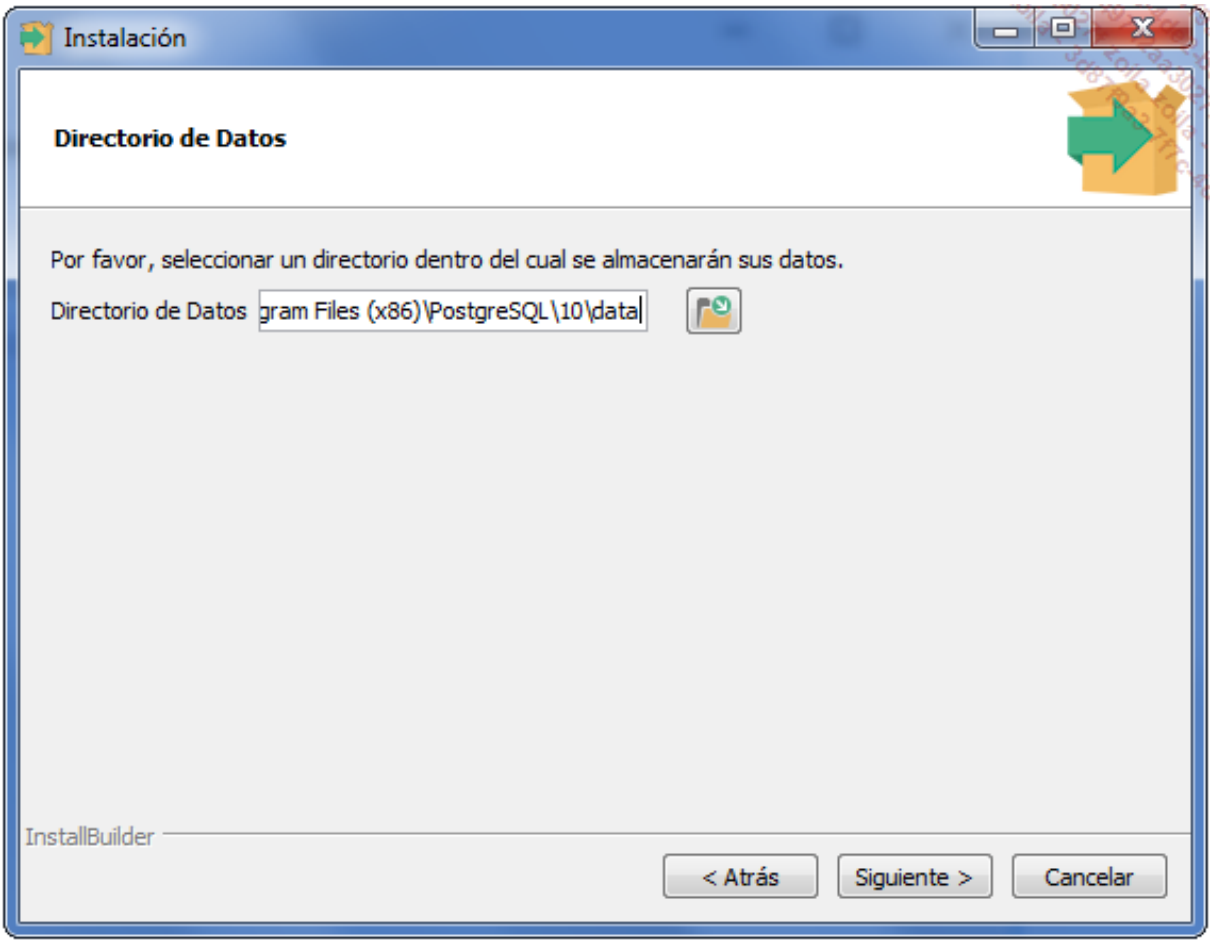
La primera opción permite seleccionar la ubicación de la instalación de los programas de PostgreSQL. No es obligatorio modificar la opción por defecto:



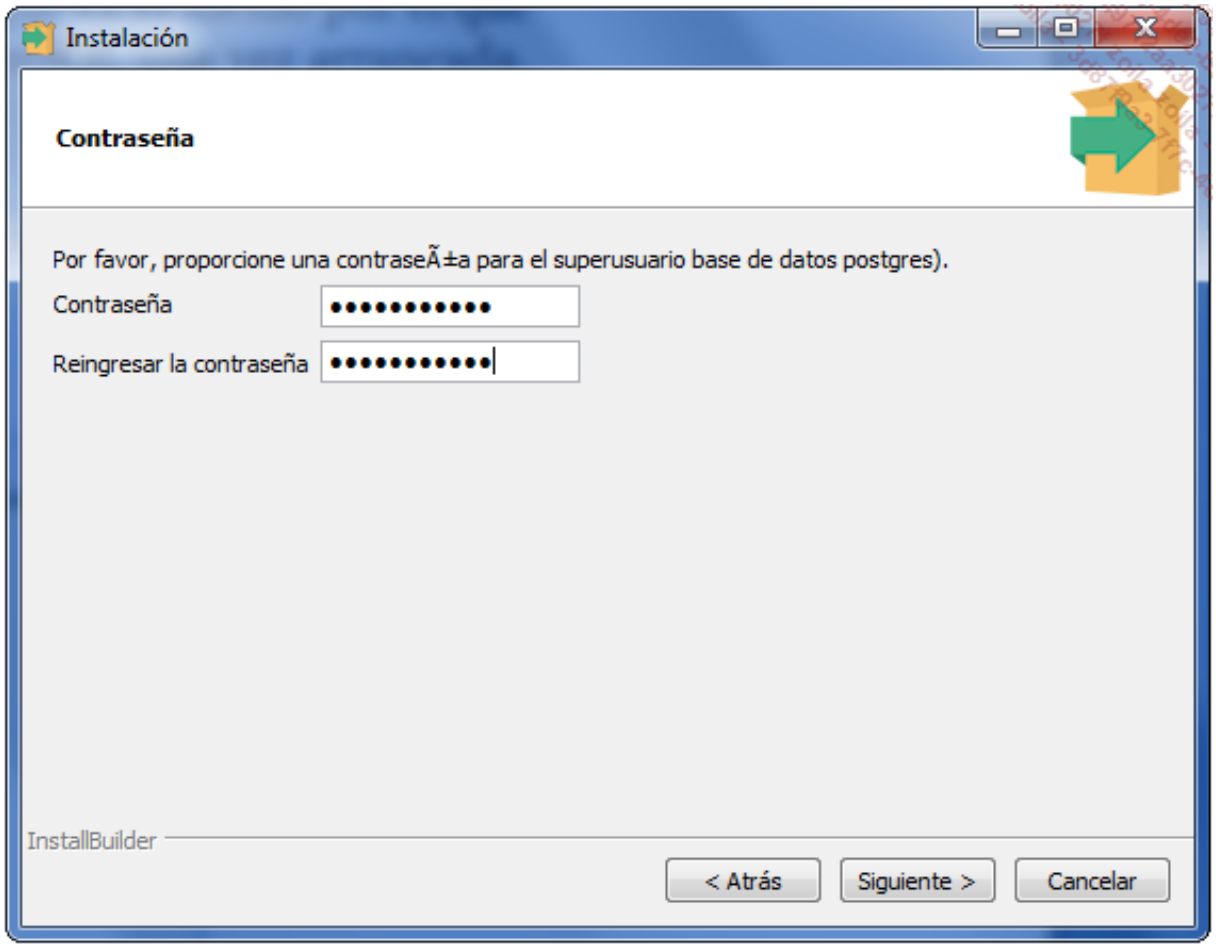
La siguiente etapa permite seleccionar los componentes que se van a instalar. Las herramientas en línea de comandos permiten acceder a una instancia PostgreSQL remota. La herramienta Stack Builder permite añadir herramientas en el lado servidor; por ejemplo, PgBouncer.



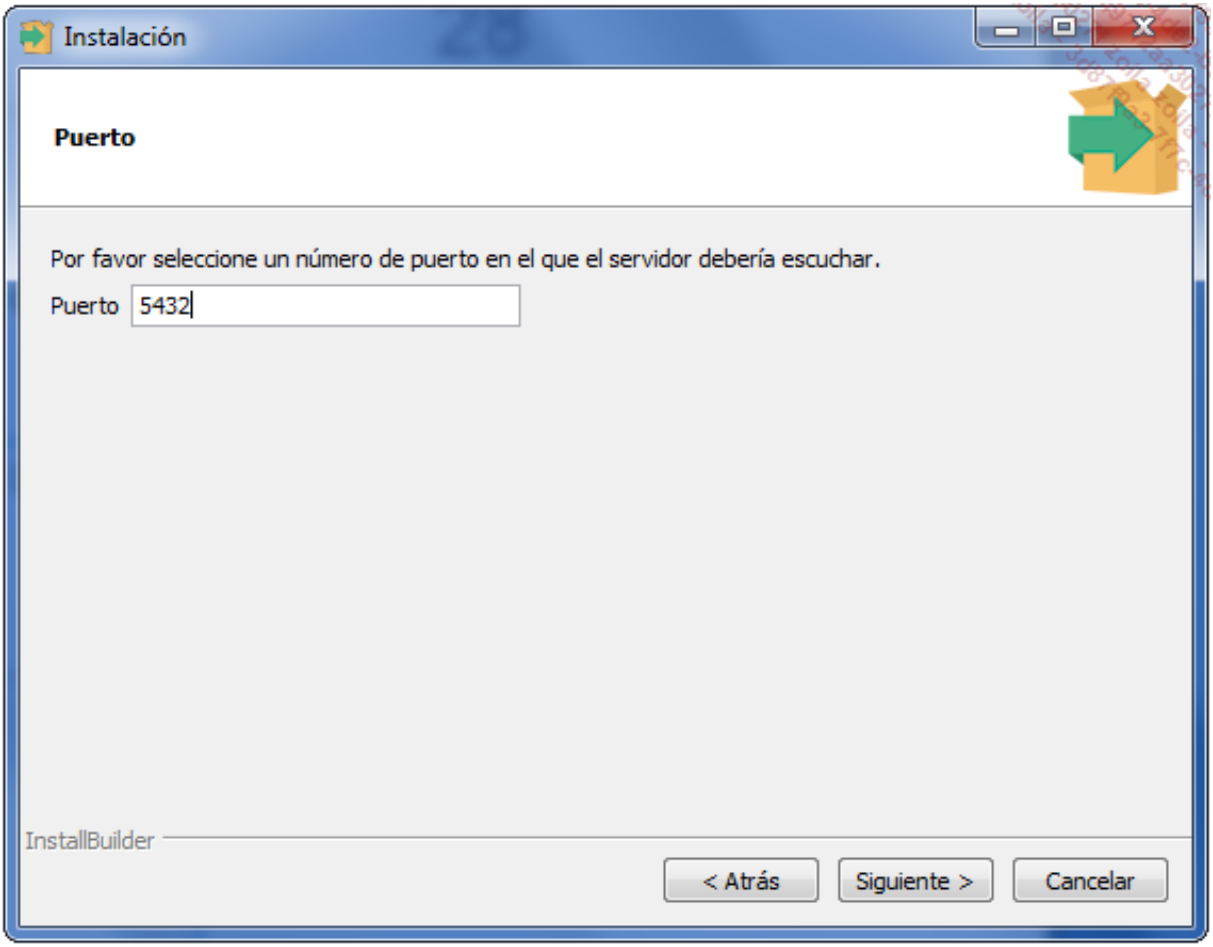
La siguiente etapa afecta a la elección del directorio de los datos. Por defecto, el instalador propone ubicar los datos en Program Files, lo que no es una buena práctica. Conviene seleccionar el directorio y el disco duro o subsistema de disco que recibirá los datos y que deberá contar con suficiente espacio libre, correspondiente a las necesidades de rendimiento requeridas.



La contraseña del usuario 'postgres' es importante porque es el medio por el que el administrador se puede conectar a la instancia PostgreSQL una vez arrancada para crear objetos como roles o bases de datos.

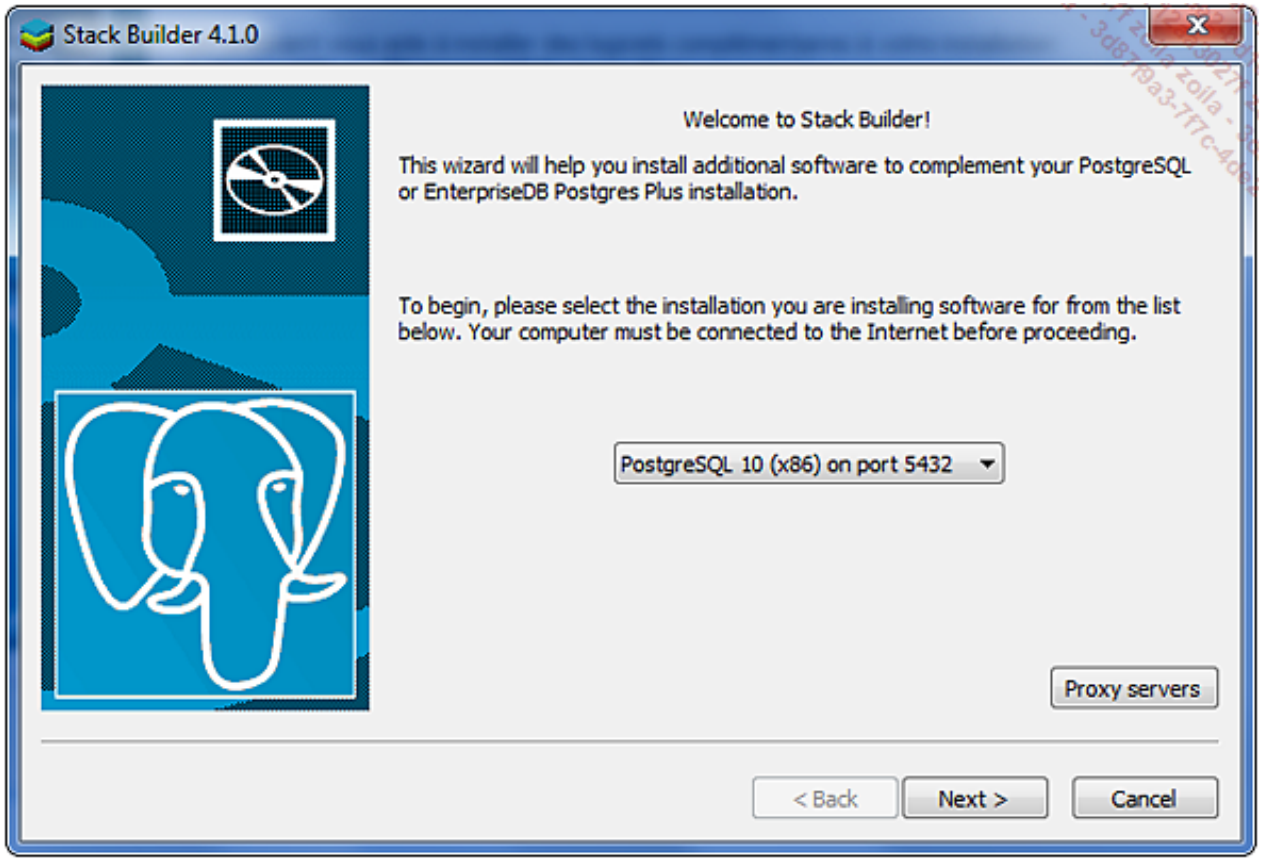


El puerto por defecto de PostgreSQL es 5432; se puede cambiar más adelante en el archivo de configuración postgresql.conf.

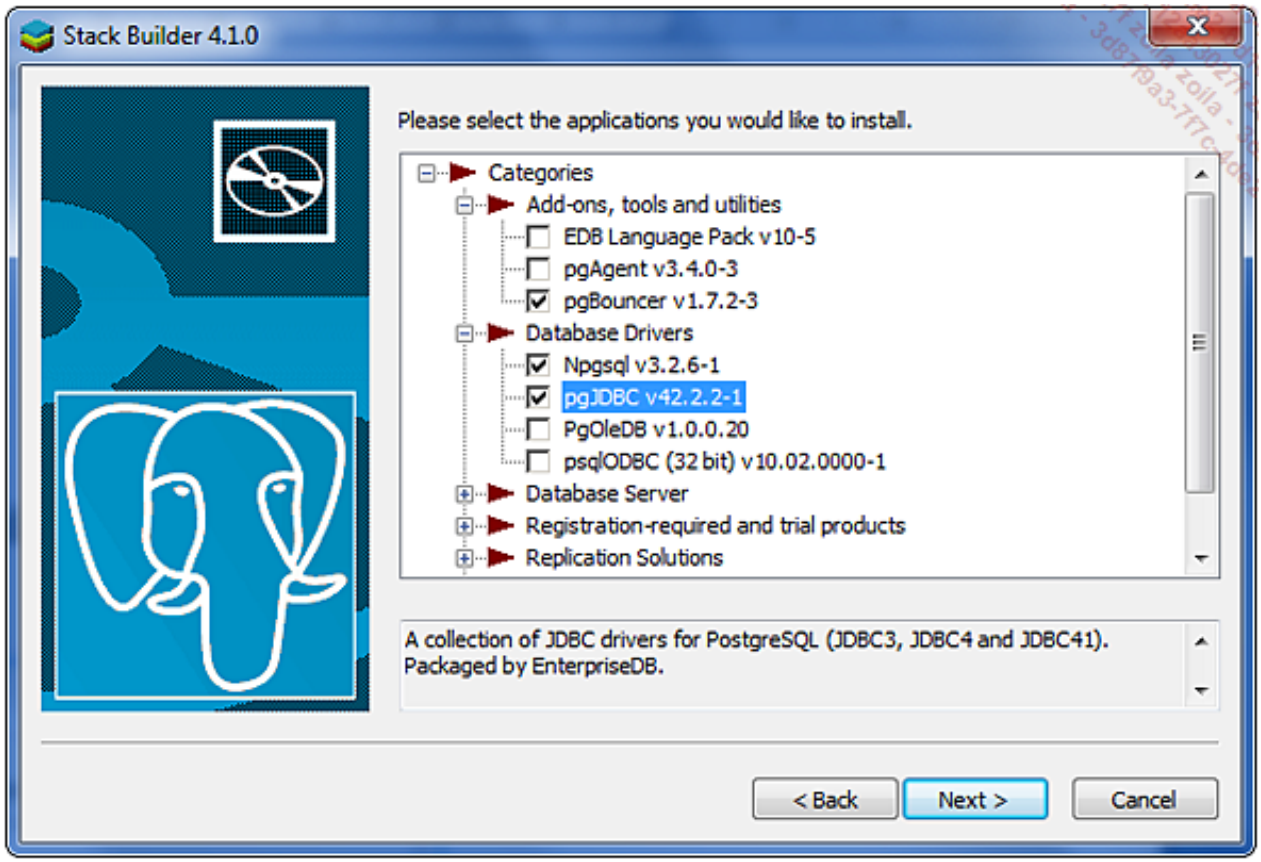


Un clic en **Terminar** finaliza la instalación de PostgreSQL. Los archivos se crean en el directorio seleccionado y se crean también los datos de la instancia.

La herramienta Stack Builder se puede lanzar para instalar herramientas adicionales. La opción correspondiente se marca por defecto.

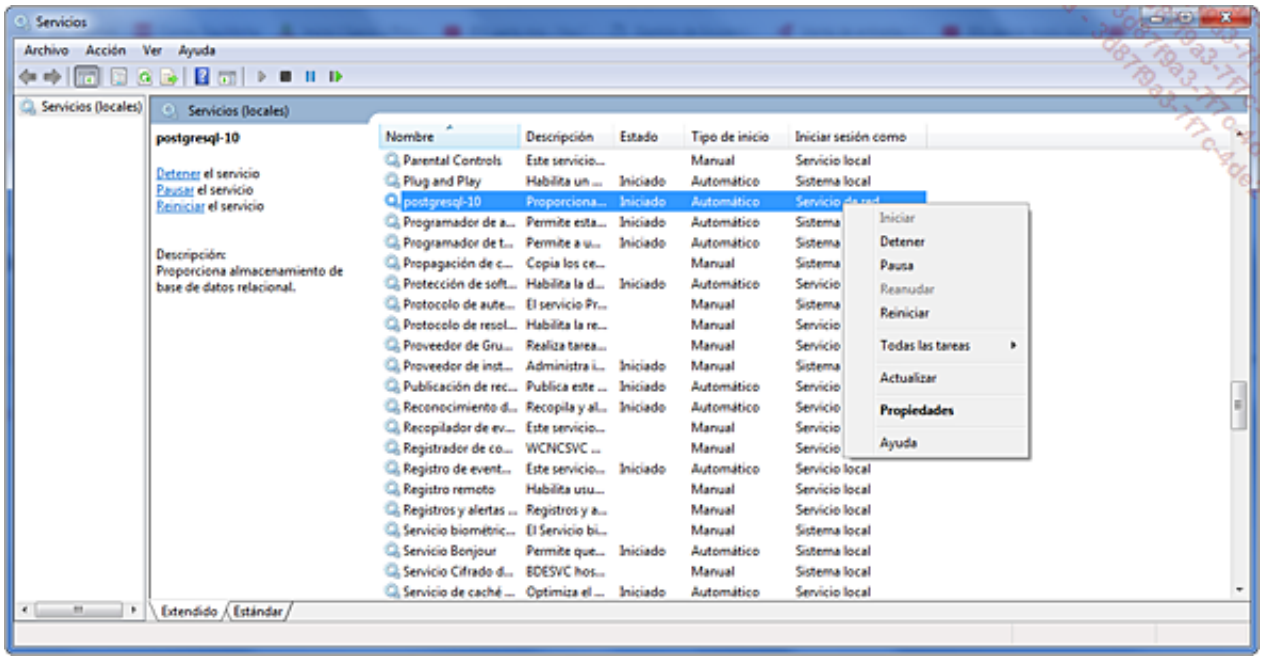


En función de la instancia de PostgreSQL elegida, se ofrece una arborescencia de software adicional, entre pgBouncer o Npgsql.



Es suficiente con marcar las casillas del software deseado y Stack Builder lanzará las instalaciones solicitadas.

Una vez que termina la instalación, se puede controlar PostgreSQL como cualquier servicio con la herramienta Servicios de Windows.



Instalación silenciosa

La herramienta de instalación para Windows también se puede utilizar en modo silencioso (no se solicita ninguna interacción al usuario). Todos los argumentos son modificables pasando las opciones a este script de instalación. Para esto, es suficiente con ubicarlo en línea de comandos o en un script y la instalación se hace de manera totalmente transparente para el usuario.

El instalador se puede llamar con las siguientes opciones:

- `--help`: lista de opciones de instalación.
- `--version`: versión del instalador.
- `--optionfile <optionfile>`: archivo que contiene las opciones de instalación.
- `-- installer-language <en|es|fr>`: lenguaje del instalador, entre `en`, `es` y `fr`, por defecto `en`.
- `--extract-only <yes|no>`: extracción solo de los archivos. Desactivado (0) por defecto.
- `--disable-stackbuilder <yes|no>`: desactiva la herramienta Stack Builder. Por defecto, la herramienta está activada.
- `--mode <qt|gtk|xwindow|text|unattended>`: modo de instalación, entre `qt`, `gtk`, `xwindow`, `text` y `unattended`, por defecto `qt`.
- `--unattendedmodeui <none|minimal|minimalWithDialogs>`: modo de instalación, entre `none`, `minimal`, `minimalWithDialogs`, por defecto `minimal`.
- `--prefix <prefix>`: ruta de instalación de los programas. El valor por defecto es: `C:\Program Files (x86)\PostgreSQL\10`.
- `--datadir <datadir>`: directorio de los datos de la instancia PostgreSQL. El valor por defecto es: `C:\Program Files (x86)\PostgreSQL\10\data`.
- `--superaccount <superaccount>`: nombre del superusuario de la instancia PostgreSQL. El valor por defecto es `postgres`.
- `--superpassword <password>`: contraseña del superusuario de la instancia PostgreSQL.
- `--serverport <serverport>`: puerto TCP de la instancia PostgreSQL. El valor por defecto es 5432.
- `--locale <locale>`: argumento de localización de la instancia PostgreSQL. Por defecto, el instalador utiliza el argumento del sistema operativo.
- `--servicename <servicename>`: nombre del servicio Windows de PostgreSQL.
- `--serviceaccount <serviceaccount>`: nombre del usuario del sistema operativo que ejecuta la instancia PostgreSQL. El valor por defecto es `postgres`.
- `--servicepassword <servicepassword>`: contraseña del usuario Windows que ejecuta la instancia PostgreSQL. El valor por defecto es la contraseña del superusuario de la instancia PostgreSQL.
- `--install_runtimes <install_runtimes>`: instala o no las dependencias para Microsoft Visual C++ antes de la instalación de PostgreSQL. Por defecto, se instalan las librerías.
- `--install_plpgsql <yes|no>`: instala o no el lenguaje de procedimientos almacenados PL/pgSQL. El valor por defecto es yes.
- `--create_shortcuts <yes|no>`: crea las entradas en el menú Windows. Activado por defecto.

- `--debuglevel <debuglevel>`: nivel de depuración del instalador entre 0 y 4, 2 por defecto.
- `--debugtrace <debugtrace>`: archivo de depuración.

Ejemplo

El siguiente ejemplo muestra una instalación silenciosa sin interfaz gráfica, modificando el directorio de los datos por defecto:

```
postgresql-10.0-windows-x64.exe --servicename postgresql10
--datadir C:\PostgreSQL\10\data --superpassword contraseña
--mode unattended --unattendedmodeui none
```

De la misma manera, es posible ubicar sus opciones en un archivo de configuración, simplificando el registro de comandos. El archivo option.ini se puede escribir como en el siguiente ejemplo:

```
mundo=unattended
unattendedmodeui=none
servicename=postgresql10
datatadir=C:\PostgreSQL\10\data
superpassword=contraseña
```

Una vez creado el archivo, llamamos al ejecutable de instalación de la siguiente manera:

```
postgresql-10.0-windows-x64.exe -optionfile option.ini
```

Descargar el instalador BigSQL

Desde la página https://www.openscg.com/bigsql/package-manager/, es posible instalar el comando `pgc`, que a continuación va a permitir instalar PostgreSQL en el sistema. Desde una línea de comandos Windows (cmd.exe), el siguiente comando instala el comando `pgc`:

```
@powershell -NoProfile -ExecutionPolicy unrestricted -Command "iex
((new-object
net.webclient).DownloadString('http://s3.amazonaws.com.sabidi.urv.cat/pgcentral/
install.ps1'))"
```

Es conveniente seleccionar la ubicación desde la que se lanzará este comando, porque esta ubicación es la utilizada para las instalaciones posteriores, incluido el almacenamiento de los datos. Una vez que se ha lanzado el script de instalación, los siguientes comandos permiten instalar y lanzar una instancia de PostgreSQL:

```
cd bigsql
pgc install pg10
pgc init pg10
pgc start pg10
```

La inicialización crea una instancia en un directorio pg10, situada ella misma en el directorio bigsql. En esta ubicación es donde se almacenan los datos. Por lo tanto, es importante seleccionarla bien. Es posible mover esta ubicación cuando la instancia PostgreSQL se detiene.

Además, la herramienta `pgc` permite instalar mucho software del ecosistema PostgreSQL, listado en esta página: https://www.openscg.com/bigsql/components/, entre los que están pgBackRest, pgBouncer e incluso pgBadger.

A pesar de no existir un asistente gráfico y de que el uso del registro de comandos puede parecer poco habitual para un usuario de un sistema MS-Windows, este método es eficaz y rápido.

Instalación de extensiones

El ecosistema de PostgreSQL ofrece numerosas extensiones para PostgreSQL, que permiten enriquecer su comportamiento y ayudar al administrador de bases de datos a explotar las instancias. Estas extensiones están reunidas en el seno de una red de extensiones: PostgreSQL eXtension Network: https://pgxn.org/

Algunas extensiones de esta red se instalan a través de los almacenes de paquetes para Debian o RedHat, proporcionados por los desarrolladores de PostgreSQL. Se trata de una alternativa pertinente porque los paquetes se proporcionan en forma de programas o librerías binarias ya compiladas, mientras que la utilización de la red PGXN hace necesaria la compilación de las extensiones.

Cuando la extensión deseada no está disponible en los almacenes, la red PGXN propone descargar y compilar automáticamente la extensión. Para esto, hay que instalar el comando `pgxn` y los archivos de encabezado de PostgreSQL. Los mecanismos de dependencias de las distribuciones Debian o Redhat instalan el resto de las dependencias necesarias. En un sistema Debian, la instalación se hace con el siguiente comando:

```
apt-get install pgxnclient postgresql-server-dev-10
```

En un sistema Redhat, la instalación se hace con el siguiente comando:

```
yum install pgxnclient postgresql10-libs
```

El comando `pgxn` permite buscar entre las extensiones e instalarlas. Por ejemplo, para instalar la extensión que permite utilizar el tipo de datos `tinyint`, es suficiente con el siguiente comando:

```
pgxn install tinyint
```

A continuación, hay que declararla en la base de datos:

```
psql -Upostgres clients
clients=# create extension tinyint;
CREATE EXTENSION
```

Por lo tanto, es posible eliminar la instalación de las extensiones.

Inicialización del sistema de archivos

Introducción

Una de las etapas más importantes de la instalación de PostgreSQL consiste en crear un directorio en el que escribirá el servidor y leerá los datos de las bases de datos. Esta etapa es un paso previo necesario para el arranque de PostgreSQL.

La elección del directorio que se debe utilizar no es una cuestión menor; todos los datos se almacenarán en él, salvo excepciones. Por lo tanto, es importante seleccionar un directorio situado en un subsistema de disco rápido, si es posible dedicado a esta tarea para optimizar las lecturas y escrituras de datos.

Describir la elección del subsistema de disco excede el marco de este libro; sin embargo, se pueden listar varios tipos de estos subsistemas, como por ejemplo:

- Controlador RAID con discos SAS o SSD, configurados en RAID 10.
- Base de almacenamiento SAN con conexiones Fibre Channel.
- SSD, tarjeta FusionIO o simplemente RAID software.

Se debe tener cuidado con esta elección. La adquisición de hardware específico es frecuente y se debe hacer en función de objetivos concretos: rendimiento esperado, durabilidad, costes, capacidad de alojamiento y administración, etc. Esta lista no es exhaustiva.

Un segundo aspecto importante en este estado es la elección del sistema de archivos utilizado. En función del sistema operativo elegido, se pueden usar diferentes sistemas de archivos y la elección puede ser determinante en lo que respecta al rendimiento, la durabilidad de los datos o el mantenimiento. En lo que respecta a los sistemas operativos GNU/Linux, entre las numerosas elecciones disponibles, Ext4 y XFS normalmente son las preferidas por los administradores de bases de datos. Las opciones de montaje de estos sistemas de archivos se pueden adaptar al uso que hace PostgreSQL, lo que hace adecuada la utilización de un sistema de archivos dedicado para los datos de PostgreSQL.

El directorio seleccionado debe ser propiedad del usuario que ejecuta el servidor PostgreSQL y que se ha creado durante la instalación. Por defecto, este usuario es `postgres`, aunque no importa el nombre elegido.

Por ejemplo, si el directorio en el que se almacenarán las bases de datos es `/var/lib/postgres/data`, los siguientes comandos muestran su creación y la modificación de sus atributos en Linux:

```
[root]# mkdir -p /var/lib/postgres/data
[root]# chown -R postgres:postgres /var/lib/postgres
[root]# chmod -R 700 /var/lib/postgres
```

Además de este primer directorio, hay otros que se pueden utilizar para almacenar los archivos de traza binarios y otros espacios de tablas. Estas dos nociones se explicarán en los siguientes capítulos.

Inicialización de una instancia

A cada directorio creado le corresponde una instancia de PostgreSQL, es decir, un servidor que por, lo tanto, tiene su propio espacio de almacenamiento, que acepta conexiones entrantes y dispone de sus propios archivos de configuración.

También se puede llamar a un directorio grupo de bases de datos e incluso **clúster**. El término anglosajón **clúster** algunas veces puede llevar a confusión según el contexto en que se use. Se utiliza con frecuencia en el contexto de las bases de datos y significa simplemente **grupo**. Aquí se emplea en el sentido de grupo de bases de datos.

La instancia y el grupo de bases de datos se corresponden con las mismas bases de datos, pero desde dos puntos de vista diferentes: la instancia se corresponde con una ejecución del servidor PostgreSQL, y un grupo de bases de datos, con un contenedor de bases de datos, en forma de directorios y archivos.

La inicialización de una instancia se realiza con el comando `initdb`. Situando correctamente las opciones y sus valores, es posible personalizar la instancia. Estas opciones y valores tienen una influencia importante en la manera en la que se comporta el servidor, por ejemplo durante la creación de bases de datos en la instancia, durante la creación de índices o durante la búsqueda de datos. Por lo tanto, es conveniente configurar correctamente esta creación.

1. Opciones del comando

El conjunto de opciones del comando `initdb` se corresponde con la siguiente lista:

- `-A,--auth=METHOD`: método de autenticación por defecto.
- `-D,--pgdata=DATADIR`: ruta del directorio de los datos.
- `-X,--xlogdir=XLOGDIR`: define la ubicación de las trazas de transacciones.
- `-k,--data-checksums`: activa la utilización de las sumas de control de los datos.
- `-E,--encoding=ENCODING`: codifica por defecto las bases de datos.
- `--locale=LOCALE`: argumento regional por defecto de las bases de datos.
- `--lc-collate`, `--lc-ctype`, `--lc-messages`, `--lc-monetary`, `--lc-numeric`, `--lc-time`: detalles de los argumentos regionales por defecto. Por defecto se utilizan los ajustes del sistema operativo.
- `--no-locale`: utilización del argumento regional C, equivalente a `--locale=C`.
- `-T, --text-search-config=CFG`: configuración por defecto de la búsqueda de texto completo.
- `-U, --username=NAME`: superusuario de la instancia.
- `-W, --pwprompt`: solicita interactivamente la contraseña del super-usuario.
- `--pwfile=FILE`: ruta del archivo que contiene la contraseña del super-usuario.
- `-L DIRECTORY`: indica la ubicación de los archivos de entrada.
- `-n, --noclean`: en caso de error, no elimina el directorio de los datos. Servirá para los bloqueos.
- `-N, --nosync`: no solicitar escrituras síncronas de los datos durante la creación.
- `-S, --sync-only`: solo solicitar una sincronización para el directorio de los datos.
- `-s, --show`: muestra los ajustes internos.

a. Opciones esenciales

Entre la lista presentada, es esencial entender y seleccionar correctamente algunas opciones. En particular, algunas de ellas no se pueden modificar posteriormente en el directorio de datos así creado.

La opción `-D` permite seleccionar la ubicación de los datos. Esta ubicación es importante y, mientras que la instancia no se arranque, siempre es posible mover los datos. De cualquier manera, lo que importa es la ubicación final de los datos de la instancia.

La opción `-A` permite seleccionar la política de seguridad por defecto, es decir, el contenido del archivo `pg_hba.conf` (ver capítulo Conexiones - sección Sesión). Esta política por defecto puede permitir una inicialización de las aplicaciones (bases de datos y roles) y se puede modificar más adelante, una vez que la instancia se arranque. Los valores posibles son los aceptados en el archivo `pg_hba.conf`, por ejemplo: `trust`, `md5`, `ident`.

La opción `-X` permite seleccionar la ubicación de los archivos de traza de transacciones (archivos WAL). Esta ubicación es importante; las transacciones inicialmente se escriben en estos archivos y después se sincronizan en el sistema de archivos, antes de presentarse realmente en las tablas. La elección de una ubicación diferente permite optimizar el rendimiento para estos archivos particulares. Esta ubicación no se puede modificar una vez arrancada la instancia.

La opción `-k` permite activar el control de la calidad de los datos escritos, usando las sumas de control, calculadas durante la escritura y verificadas en la lectura. Estas sumas de control no están activas por defecto y pueden provocar una ralentización en las escrituras y lecturas de los datos.

b. Elección del juego de caracteres

Un argumento adicional relativo a la elección del juego de caracteres se puede integrar en el comando `initdb`.

Un juego de caracteres determina una correspondencia entre un carácter y la manera en la que se almacena, en forma de campo de bits. Con PostgreSQL, cada base de datos tiene su propio juego de caracteres. Este argumento determina la manera en la que los caracteres se escriben en el disco.

La creación de la instancia crea una base de datos modelo que sirve para la creación de las bases de datos de las aplicaciones. La elección del juego de caracteres durante la creación de la instancia se corresponde de hecho con el juego de caracteres de la base de datos modelo `template1`.

El valor de esta opción se puede determinar en función de las variables de entorno del sistema operativo. Si esto no es posible, entonces el valor por defecto es `SQL_ASCII`. Este valor no se corresponde generalmente con las necesidades de las aplicaciones que utilizarán las bases de datos. Las aplicaciones normalmente usan juegos de caracteres como `UTF-8` o `LATIN1`, que permiten almacenar caracteres acentuados.

Por lo tanto, es pertinente seleccionar correctamente el juego de caracteres inicial, lo que simplifica la creación posterior de las bases de datos.

La tabla siguiente reúne los juegos de caracteres que PostgreSQL conoce, limitado a los que se utilizan generalmente en Europa occidental:

Nombre	Descripción	Idioma	Bytes/Carácter	Alias
SQL_ASCII	No especificado	Todos los idiomas	1	
UTF8	Unicode, 8-bit	Todos los idiomas	1-4	Unicode
LATIN1	ISO 8859-1, ECMA 94	Europa occidental	1	ISO88591
LATIN9	ISO 8859-15	LATIN1 con el Euro	1	ISO885915
WIN1252	Windows CP1252	Europa occidental	1	

El juego de caracteres `SQL_ASCII` se corresponde con la tabla ASCII. Todos los caracteres superiores a 127 insertados en una tabla no se interpretan, de tal manera que no importa el carácter que acepte. Este comportamiento impide la conversión y validación de los caracteres, lo que hace que los datos almacenados sean difíciles de explotar.

El juego de caracteres UTF8 se corresponde con la implementación del estándar Unicode, que es un intento de reunir en un único juego de caracteres todos los alfabetos conocidos. De esta manera, todos las caracteres se identifican de manera única.

Los juegos de caracteres LATIN1, LATIN9 y WIN1252 permiten almacenar los caracteres correspondientes a los alfabetos de los idiomas de Europa occidental. Por lo tanto, su alcance no es tan extendido con el Unicode.

La opción del comando initdb, que permite seleccionar el juego de caracteres por defecto, es la siguiente:

```
-E codage, --encoding=codage
```

Donde codage se corresponde con el juego de caracteres seleccionado.

Es preferible ajustar correctamente el sistema operativo con los argumentos de localización deseados.

c. Ajustes de los argumentos locales

También hay disponibles otros argumentos, llamados argumentos locales.

Los argumentos locales permiten a PostgreSQL modificar su comportamiento en las ordenaciones, los formateos de cifras o fechas y los mensajes. En efecto, en función del idioma utilizado y de la región de origen, las reglas de ordenación o el formateado de una fecha pueden ser diferentes y PostgreSQL debe tener esto en cuenta durante la presentación de los datos o para generar un índice.

Para esto, PostgreSQL se basa en los argumentos locales del sistema operativo. En los sistemas operativos Linux, el comando locale -a permite conocer los argumentos locales disponibles. Las variables de entorno LC_ALL, después LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC,LC_TIME y para terminar LANG se utilizan con el comando initdb para inicializar el grupo de bases de datos.

Si ninguna de estas variables de entorno está disponible, entonces los argumentos locales se posicionarán en C, que se corresponde con el estándar ISO C.

El uso de argumentos locales específicos modifica el comportamiento de PostgreSQL, limitando por ejemplo la utilización de índices con la palabra clave LIKE o teniendo un impacto negativo sobre el rendimiento.

Algunos de estos argumentos tienen una influencia directa sobre la manera en la que PostgreSQL va a escribir un índice.

Por lo tanto, es initdb el que va a inicializar los argumentos locales con las siguientes opciones:

```
--locale=locale
```

O incluso, de manera separada:

```
--lc-collate=locale, --lc-ctype=locale, --lc-messages=locale,
--lc-monetary=locale, --lc-numeric=locale --lc-time=locale
```

Donde locale se corresponde con un código de idioma y región. Por ejemplo, para el idioma castellano de España, el código es es_ES, y para el idioma inglés de Estados Unidos, el código es en_US.

2. Ejecución del comando

El comando `initdb` se ejecuta habitualmente con la cuenta de usuario propietario de los archivos. En el ejemplo, se trata de la cuenta `postgres`. Los siguientes comandos permiten conectarse como `postgres` al sistema y lanzar el comando `initdb`:

```
[root]# su - postgres
[postgres]$ initdb -A trust -E UTF8 -locale=es_ES -D
/var/lib/postgresql/10/data -X /pgxlog/10/wals
```

En este ejemplo, la política de autenticación es `trust` (sin contraseña), pero, por defecto, la instancia no permite conexiones locales. El juego de caracteres es `UTF8`, el argumento regional es `es_ES`, la ruta de los datos es `/var/lib/postgresql/10/data` y los archivos de traza de transacciones están en el directorio `/pgxlog/10/wals`.

El comando no puede estar en las rutas por defecto (variable de entorno `PATH`). Este es el caso cuando PostgreSQL se instala por un sistema de paquetes, como en los sistemas Debian o Red Hat.

En un sistema Debian, la ruta completa del comando es `/usr/lib/postgresql/10/bin/initdb`. En un sistema Red Hat y relacionados, la ruta es `/usr/lib/pgsql-10/bin/initdb`. Ver los sistemas de paquetes en el capítulo Instalación.

En Windows, es necesario arrancar un intérprete de comandos actuando como `postgres`. Esto se puede hacer lanzando el comando:

```
> runas /user:postgres cmd.exe
```

Este comando crea una arborescencia de directorios y archivos. El siguiente ejemplo muestra un extracto del contenido del directorio `data`. Este directorio contiene algunos archivos y directorios.

Directorios creados

- `base`: contiene el conjunto de las bases de datos y, por lo tanto, las tablas y los índices. Se trata del espacio de tablas por defecto.
- `global`: contiene las tablas globales, accesibles en la instancia desde todas las bases de datos. Se trata de un espacio de tablas particular.
- `pg_xact`: contiene los archivos utilizados para conocer el estado de las transacciones en curso cuando la instancia PostgreSQL está activa.
- `pg_dynshmem`: contiene los archivos utilizados por el sistema de asignación dinámica de memoria.
- `pg_logical`: contiene los datos sobre la descodificación lógica de los datos replicados.
- `pg_multixact`: contiene los datos de las multitransacciones.
- `pg_notify`: contiene los datos de las notificaciones asíncronas (comandos `LISTEN/NOTIFY`).
- `pg_replslot`: contiene los datos de los slots de replicación.
- `pg_serial`: contiene los datos de las transacciones serializables.
- `pg_snapshots`: contiene los snapshots exportados.
- `pg_stat`: contiene los archivos permanentes de estadísticas.
- `pg_stat_tmp`: contiene los archivos temporales de estadísticas.

- pg_subtrans: contiene el estado de las subtransacciones.
- pg_tblspc: contiene los enlaces simbólicos a los espacios de tablas creados.
- pg_twophase: contiene los archivos de transacciones preparados.
- pg_wal: contiene los archivos de traza de transacciones. Cuando la opción -X de initdb designa a otro directorio, no se trata de un directorio sino de un enlace simbólico.

El directorio que contiene la mayor parte de los datos siempre será el directorio base. El resto de los directorios solo contienen datos temporales o con bajo volumen.

Archivos creados

- PG_VERSION: archivo de texto que contiene la versión principal de PostgreSQL, por ejemplo 10.
- postgresql.auto.conf: archivo de configuración en el que se escribe el comando SQL ALTER SYSTEM. Este archivo siempre se lee en último lugar.
- postgresql.conf: archivo de configuración.
- pg_hba.conf: archivo de autenticación.
- pg_ident.conf: archivo de correspondencias de los usuarios.

Ahora están presentes todos los elementos necesarios para el arranque del servidor.

3. Creación de instancias adicionales

En un mismo sistema, es posible crear varias instancias de PostgreSQL utilizando los mismos programas y creando simplemente otro directorio de grupo de bases de datos con el comando initdb. Estas instancias distintas tienen sus propios archivos de configuración, bases de datos diferentes y aceptan conexiones diferentes.

Separado del directorio de datos, es importante distinguir otro recurso: el puerto TCP. Las conexiones al servidor pasarán por una conexión TCP sobre un puerto concreto. Por defecto, el puerto TCP utilizado por PostgreSQL es 5432. Para que varias instancias de PostgreSQL funcionen al mismo tiempo, hay que indicar a PostgreSQL que reciba las conexiones entrantes en un puerto diferente al puerto por defecto.

Es posible cambiar este puerto por defecto modificando una opción pasada al programa postmaster; pero el método más sencillo y más seguro consiste en modificar el archivo de configuración postgresql.conf, presente en el directorio del grupo de bases de datos y propio de cada instancia.

postgresql.conf es un archivo de texto modificable con cualquier editor de texto. El argumento que se ha de modificar es port. Aquí es suficiente con corregir el valor por defecto y eventualmente retirar el carácter # presente al inicio del registro.

El nuevo valor que se debe indicar tiene que ser un número de puerto TCP no utilizado en el sistema. Es posible verificar los puertos utilizados con el comando netstat con los sistemas Linux o los sistemas Windows.

Para los sistemas Linux:

```
[root]# netstat -ltn
```

Para los sistemas Windows:

```
> netstat -a
```

Es suficiente con seleccionar un puerto no utilizado, incrementando simplemente el valor por defecto:

```
port = 5433
```

Una vez guardado el archivo, la instancia está lista para arrancar y puede recibir bases de datos distintas de otra instancia.

Parada y arranque del servidor

Como indica el mensaje del resultado del comando `initdb`, existe un método sencillo para arrancar el servidor PostgreSQL:

```
[postgres]$ pg_ctl -D data -l journaltrace start
```

El comando `postgres` se corresponde con el programa del servidor. Este es el programa que aceptará las conexiones entrantes una vez cargado. Es posible arrancar el servidor de esta manera, pero este no es el mejor método.

El comando `pg_ctl` está dedicado al control del servidor PostgreSQL y, por lo tanto, al arranque del servidor `postgres`. Por consiguiente, es posible arrancar PostgreSQL como se indica; después de detener, volver a arrancar o cargar de nuevo el servidor. La opción `-D` permite indicar el directorio que contiene los datos y los archivos de configuración. La opción `-l` permite redirigir los mensajes a un archivo de trazas y la opción `start` permite indicar la acción que se debe realizar.

Las opciones `start`, `stop`, `restart` y `reload` permiten respectivamente arrancar, detener, volver a arrancar y volver a cargar el servidor.

El comando `pg_ctl` se utiliza por el script de arranque del sistema de arranque de los sistemas GNU/Linux para detener y arrancar el servidor PostgreSQL. También permite guardar PostgreSQL como servicio en los sistemas Windows.

Dos opciones permiten guardar y retirar PostgreSQL del panel de servicio de los sistemas Windows. Este registro se realiza implícitamente durante la instalación de PostgreSQL en un sistema Windows, pero puede ser útil durante la creación de otra instancia.

La opción `register` dispone de opciones específicas que permiten indicar el nombre del servicio, tal y como aparece en el panel de servicio (`-N`), el nombre (`-U`) y la contraseña (`-P`) del usuario del sistema y, para terminar, la ubicación del grupo de base de datos (`-D`). Por ejemplo, el siguiente comando permite guardar una nueva instancia de PostgreSQL. Este comando se debe ejecutar como cuenta con permisos de sistema:

```
> pg_ctl register -N pgsql10-2 -U postgres -P postgres -D
c:\PostgreSQL\data2\
```

Para retirar una instancia, se puede utilizar la opción `unregister`:

```
> pg_ctl unregister -N pgsql10-2
```

Este comando desactiva la instancia en el panel de servicio. Esta instancia siempre aparece, pero se considera desactivada. Solo desaparecerá durante el siguiente arranque del sistema.

Scripts Debian

Los paquetes fabricados para integrar PostgreSQL en la distribución Debian añadirán algunos scripts que simplifican la administración de las versiones y de los grupos de bases de datos en un sistema.

En los sistemas Debian y Ubuntu, hay disponibles dos paquetes específicos, `postgresql-common` y `postgresql-client-common`, que aportan estas funcionalidades.

El paquete `postgresql-common` ofrece los scripts siguientes:

- `/usr/bin/pg_lsclusters`: lista las instancias existentes con su configuración y la ubicación de los datos.
- `/usr/bin/pg_createcluster`: crea una instancia; este script utiliza `initdb` y, como argumento, la instancia en función de la versión solicitada y del resto de las instancias existantes.
- `/usr/bin/pg_ctlcluster`: controla las instancias utilizando el comando `pg_ctl`.
- `/usr/bin/pg_upgradecluster`: permite actualizar una instancia de otra versión de PostgreSQL; por ejemplo, de 9.6 a 10.
- `/usr/bin/pg_dropcluster`: elimina una instancia de PostgreSQL.
- `/usr/sbin/pg_maintenance`: permite realizar algunas tareas de mantenimiento sobre las instancias existentes.

El paquete `postgresql-client-common` aporta el script `pg_wrapper`, que permite utilizar un comando cliente específico, en función de la instancia sobre la que el usuario se desea conectar. Por lo tanto, este script permite utilizar siempre la versión correcta de un comando en función de la instancia.

1. El script pg_lsclusters

El siguiente ejemplo muestra el comando `pg_lsclusters`, que lista las instancias de PostgreSQL que funcionan en un sistema Debian o Ubuntu. Cada línea del resultado del comando se corresponde con una instancia y cada columna indica una propiedad de la instancia:

```
[root]# pg_lsclusters
Ver Cluster Port Status Owner    Data directory Log file
10 main    5432 online postgres /var/lib/postgresql/10/main
/var/log/postgresql/postgresql-10-main.log
```

La primera columna indica la versión de PostgreSQL utilizada para la instancia, seguida en la segunda columna por el nombre utilizado para esta instancia.

La tercera columna indica el puerto TCP utilizado por la instancia.

La cuarta columna indica el estado de la instancia: `online` significa que la instancia está arrancada y `down` que está parada.

Las siguientes columnas indican respectivamente el usuario propietario de la instancia, el directorio en el que se almacenan los datos y el archivo de trazas.

2. El script pg_ctlcluster

El comando `pg_ctlcluster` permite detener y arrancar una instancia de PostgreSQL. Los tres argumentos del comando son respectivamente la versión de PostgreSQL utilizada para la instancia, el nombre de la instancia y la acción que se va a realizar. Esta puede ser `start`, `stop`, `restart`, `reload` y `promote`.

El siguiente ejemplo muestra la parada de la instancia:

```
[root]# pg_ctlcluster 10 main stop
```

El siguiente ejemplo muestra el rearranque de la instancia:

```
[root]# pg_ctlcluster 10 main restart
```

Las opciones `start`, `stop` y `restart` permiten arrancar, detener y volver a arrancar la instancia destino. Sin embargo, con la utilización de Systemd en la mayor parte de las distribuciones actuales, es preferible arrancar y detener las instancias PostgreSQL con este y el comando `systemctl`.

La opción `reload` permite volver a leer la configuración de la instancia, aplicando las modificaciones añadidas a los archivos de configuración, tal y como se describe en el capítulo Explotación.

La opción `promote` permite bascular una instancia de un modo `standby` a un modo `primary`, tal y como se describe en el capítulo Replicación.

3. El script pg_createcluster

El comando `pg_createcluster` permite inicializar nuevas instancias de PostgreSQL, indicando la versión de PostgreSQL deseada. Existen algunas opciones que permiten personalizar la instancia:

- `-u <user>`, `-g <group>`: nombre y grupo del propietario, por defecto `postgres`.
- `-d <dir>`: directorio de los datos, por defecto `/var/lib/postgresql/<versión>/<nombre>`.
- `-l <ruta>`: ruta del archivo de trazas, por defecto `/var/log/ postgresql/postgresql-<versión>-<nombre>.log`.
- `-e <encoding>`: juego de caracteres por defecto de la instancia.
- `-s <dir>`: directorio del socket Unix, por defecto `/var/run/postgresql/`.
- `--locale=locale`, `--lc-collate=locale`, `--lc-ctype=locale`, `--lc-messages=locale`, `--lc-monetary=locale`, `--lc-numeric =locale`, `--lc-time=locale`: argumentos regionales, ya sea el único argumento `-locale` o bien un subconjunto de argumentos detallados. Por defecto, se utilizan los argumentos del sistema operativo.
- `-p <port>`: número del puerto TCP; por defecto toma el primer valor libre después de 5432.
- `--start`: arranca la instancia después de la inicialización.
- `--start-conf auto|manual|disabled`: define el comportamiento de la instancia al arranque del sistema operativo, por ejemplo `-o work_mem=4MB`.
- `-o guc=value`: modifica el archivo `postgresql.conf` con el argumento indicado.

- `--createclusterconf=file`, `--environment=file`: define las alternativas a los archivos presentes en `/etc/postgresql-common/`.
- `-- <initdb opciones>`: los dobles guiones al final de la línea de comandos permiten pasar opciones directamente al comando `initdb`, por ejemplo `-A trust`.

Después de las opciones, es necesario indicar la versión principal de PostgreSQL utilizada para la instancia, seguido del nombre de la instancia, nombre que se usará en las rutas y que se devuelve como resultado del comando `pg_lsclusters`.

El siguiente ejemplo inicializa una instancia para la versión 10:

```
[root]# pg_createcluster -e UTF8 -o work_mem=4MB --start 10 test1
-- -A trust
Creating new cluster (configuration:
/etc/postgresql/10/test1, data:
/var/lib/postgresql/10/test1)...
Moving configuration file
/var/lib/postgresql/10/test1/pg_hba.conf to
/etc/postgresql/10/test1...
Moving configuration file
/var/lib/postgresql/10/test1/pg_ident.conf to
/etc/postgresql/10/test1...
Moving configuration file
/var/lib/postgresql/10/test1/postgresql.conf to
/etc/postgresql/10/test1...
Configuring postgresql.conf to use port 5438...
```

Se arranca la instancia y queda lista para recibir conexiones:

```
[root]# pg_lsclusters
Version Cluster   Port Status Owner    Data directory
Log file
10      test1     5438 online postgres
/var/lib/postgresql/10/test1
/var/log/postgresql/postgresql-10-test1.log
```

A partir de ahora, los archivos de configuración de la instancia están presentes en el directorio `/etc/postgresql/10/test1/`, y los datos, en el directorio indicado.

Archivos de configuración

El archivo de configuración `/etc/postgresql-common/createcluster.conf` proporcionado por el paquete `postgresql-common`, o su alternativa identificada por la opción `--createclusterconf`, permite controlar la creación de un nuevo clúster posicionando los ajustes, no en la llamada del comando, sino en un archivo permanente. Los argumentos de este archivo son:

- `create_main_cluster = <bool>`: permite la creación de un clúster durante la instalación de un nuevo paquete servidor, por ejemplo una nueva versión principal de PostgreSQL. Por defecto, esta creación está activada (`true`).
- `start_conf = auto|manual|disabled`: controla el arranque de la instancia durante el arranque del sistema operativo. Los valores posibles son `auto`, `manual` o `disabled` y se escriben en el archivo `start.conf` de cada instancia. Por defecto, las instancias se arrancan automáticamente (`auto`).

- `data_directory = <dir>`: directorio de los datos, por defecto `/var/lib/postgresql/%v/%c`, donde `%v` y `%c` son las variables correspondientes a la versión y a la instancia.
- `xlogdir = <dir>`: directorio de los archivos de traza de transacciones. Ignorado por defecto, permite utilizar automáticamente un directorio alternativo; idéntico a la opción `-X` de `initdb`.
- `initdb_options`: opciones pasadas a `initdb`.
- Otras opciones: es posible añadir al final del archivo las opciones que se utilizarán en la configuración de PostgreSQL, en el archivo `postgresql.conf`, descrito en el capítulo Explotación.

4. El script pg_dropcluster

El comando `pg_dropcluster` permite eliminar una instancia existente. Se eliminarán todos los archivos de la instancia, incluido el directorio de los datos y los archivos de configuración.

Los argumentos del comando son la opción `--stop`, que detiene el servidor antes de eliminar los archivos, seguido de la versión de PostgreSQL utilizada por la instancia y, para terminar, el nombre de la instancia. El ejemplo que sigue elimina la instancia `test1`:

```
[root]# pg_dropcluster --stop 10 test1
```

5. El script pg_upgradecluster

El comando `pg_upgradecluster` permite modificar la versión de PostgreSQL utilizada para una instancia, pasando por ejemplo de la versión 9.6 a la versión 10. Cuando una instancia utiliza la versión 10, es suficiente con lanzar este comando que automatiza la copia de los datos o la modificación de los datos para utilizar una versión más reciente.

Este script puede usar dos métodos para proceder a esta actualización:

- La utilización de `pg_dump` para extraer los datos e inyectarlos en la nueva versión.
- La utilización de `pg_upgrade`, modificando los datos en disco.

Estos dos comandos se explican en detalle en el capítulo Explotación.

Esta actualización implica la no disponibilidad del servicio durante todo el tiempo de la operación, independientemente del método elegido.

Las opciones del script son las siguientes:

- `-v newversion`: define la versión de la nueva instancia. Por defecto, se trata de la última versión disponible.
- `--logfile file`: define el archivo de trazas de la nueva instancia.
- `--locale=locale`: define el argumento regional de la nueva instancia. Por defecto, el valor es el de la antigua instancia.
- `-m, --method=dump|upgrade`: define el método que se debe utilizar y, por lo tanto, la herramienta `pg_dump` o `pg_upgrade`. Por defecto, el método es `dump`.
- `-k, --link`: con el método `upgrade`, se utilizan enlaces físicos en lugar de copiar los archivos. Esta opción se pasa directamente a `pg_upgrade`.

Ahora existe una nueva instancia creada con el mismo resultado que con la llamada del comando `pg_createcluster`, pero con los datos de la antigua instancia, ahora presentes en la nueva.

Este sencillo procedimiento no corrige las diferencias funcionales que pudieran existir entre diferentes versiones de PostgreSQL. Es conveniente comprobar correctamente el comportamiento de esta nueva instancia.

Conexiones

Introducción

PostgreSQL es un servidor de bases de datos: espera conexiones desde el software cliente. Este software cliente puede abrir una conexión con el servidor siguiendo dos métodos: el protocolo TCP/IP o un socket UNIX. Con los sistemas Windows o cuando el cliente se sitúa en una máquina host diferente a la del servidor, solo es posible la conexión por TCP/IP.

Por convención, el número de puerto TCP utilizado por defecto es el 5432. Se pueden utilizar otros números de puerto, según la configuración del servidor.

Sesión

Una sesión, abierta desde la conexión de un software cliente, solo afecta a una única base de datos en el servidor y a una única cuenta de usuario. De esta manera, cuando una aplicación desee utilizar varias bases de datos, debe abrir otras conexiones. De la misma manera, si una aplicación desea abrir una sesión con otra cuenta de usuario, se debe abrir otra conexión.

Una conexión entre un software cliente y el servidor PostgreSQL necesita la existencia de dos objetos del lado del servidor: una cuenta de usuario y una base de datos. La primera conexión se puede realizar con una cuenta de usuario y una base de datos creada durante la inicialización del grupo de bases de datos.

El lado servidor (pg_hba.conf)

La apertura de una conexión está controlada por PostgreSQL, siguiendo las reglas establecidas en un archivo de configuración: `pg_hba.conf`. Este archivo se encuentra en el directorio del grupo de bases de datos. Contiene las reglas que definen los permisos y restricciones de acceso en función del nombre de usuario, la base de datos, el origen y el método utilizado.

Cada línea del archivo se corresponde con una regla. Durante un intento de conexión, PostgreSQL recorre las reglas y las compara con los argumentos de la conexión para determinar la apertura o no de la conexión.

Una regla está compuesta por cuatro o cinco elementos:

- El tipo de conexión.
- El nombre de la base de datos.
- El nombre del usuario.
- El identificador en la red.
- El método de autenticación.

El tipo de conexión puede ser:

- `local`: este tipo de conexión identifica las conexiones desde un socket UNIX.
- `host`: este tipo de conexión identifica las conexiones desde una red TCP/IP. Las conexiones pueden utilizar o no el cifrado SSL.
- `hostssl` o `hostnossl`: de la misma manera, estos dos tipos de conexiones identifican las conexiones TCP/IP, pero hacen explícita la utilización o no del cifrado SSL de la conexión.

Si falta un tipo de conexión en el archivo, la conexión es imposible. Por ejemplo, si el tipo de conexión que falta es `local`, no se puede realizar ninguna conexión utilizando un socket Unix.

El nombre de la base de datos indica la base de datos implicada por la regla. Se pueden indicar varios nombres, separados por comillas, y se pueden utilizar las palabras clave `all`, `sameuser` y `samerole`:

- La palabra clave `all` identifica cualquier base de datos.
- La palabra clave `sameuser` significa que la base de datos tiene el mismo nombre que el usuario de la conexión.
- La palabra clave `samerole` significa que el usuario que abre la conexión debe ser miembro del rol que tiene el mismo nombre que el de la base de datos indicado en la conexión.

El nombre de usuario identifica la cuenta utilizada para la conexión. La palabra clave `all` identifica todas las cuentas de usuario y, cuando el nombre está directamente seguido por el carácter +, este campo identifica todas las cuentas miembro del rol indicado.

El identificador en la red permite identificar el origen de la conexión. La notación puede tener varias formas:

- La notación CIDR, con la dirección del host o de la red, el carácter / después de la máscara CIDR; por ejemplo: 192.168.1.0/24
- La notación clásica, con el nombre del host o de la red, un carácter [Espacio] después de la máscara; por ejemplo: 192.168.1.0 255.255.255.0
- Un nombre de host: se hace una búsqueda DNS inversa con la dirección IP de la conexión, para poder validar la correspondencia. Se puede utilizar una notación relativa, empezando por un punto. Por ejemplo, si se indica `.example.com`, entonces `pepe.example.com` es válida.

Estas notaciones solo afectan a los tipos de conexiones `host`, `hostssl` y `hostnossl`.

Para terminar, el método de autenticación determina la manera en la que se puede abrir la conexión:

- `trust`: conexión sin condición. Este método permite abrir una conexión sin proporcionar la contraseña. Se debe utilizar con precaución.
- `md5`: este método solicita una contraseña cifrada para abrir una conexión y es preferible. Sustituye a los métodos `password` y `crypt`, que se corresponden con antiguas versiones de PostgreSQL.
- `scram-sha-256`: desde la versión 10, este método permite un intercambio seguro de datos, más seguro que md5 y que responde a los estándares actuales en términos de seguridad informática.
- `cert`: conexión por certificado SSL. Similar a las conexiones SSH, este método permite autenticar el cliente sin proporcionar la contraseña.
- `reject`: este método rechaza cualquier intento de conexión correspondiente con la regla.
- Otros destinos que permiten utilizar métodos de autenticación se basan en un servicio externo de PostgreSQL: gssapi (Kerberos), sspi, radius y ldap.

Los siguientes ejemplos ilustran la manera de escribir este archivo de configuración:

```
local all postgres trust
host all postgres 192.168.0.0/24 md5
host clients clientes 192.168.1.2/32 trust
host clients clientes 0.0.0.0/0 md5
```

La primera línea autoriza al usuario `postgres` a conectarse a todas las bases de datos sin dar la contraseña, pero solo desde un socket Unix.

La segunda línea autoriza al usuario `postgres` a conectarse desde cualquier host de la red local `192.168.0.0/24`, proporcionando la contraseña.

La tercera línea autoriza al usuario `clientes` a conectarse a la base de datos clientes sin proporcionar la contraseña, pero solo desde el host 192.168.1.2.

Para terminar, la cuarta línea autoriza al usuario `clientes` a conectarse desde cualquier host a la base de datos `clientes`, proporcionando la contraseña.

El orden de los registros es importante. En efecto, si la tercera y la cuarta línea se cambian, entonces el usuario `clientes` nunca se podrá conectar desde el host 192.168.1.2 sin proporcionar la contraseña, porque la correspondencia se hará con el registro anterior, indicando una conexión desde cualquier host.

La escritura de este archivo se debe hacer siguiendo una metodología, escribiendo simplemente las reglas necesarias y respetando las reglas de lectura de PostgreSQL: las reglas más concretas se deben escribir en primer lugar, seguidas de las reglas genéricas.

Clientes

Se proporcionan varias herramientas destinadas a los administradores de bases de datos; algunas para el servidor, otras como proyectos externos a PostgreSQL. En todos los casos, estas herramientas emplean el mismo protocolo cliente/servidor utilizado para las aplicaciones. Este protocolo se implementa en una librería proporcionada por PostgreSQL: libpq. Algunos lenguajes ofrecen una implementación nativa del protocolo por medio de un controlador, como el controlador JDBC del lenguaje Java.

1. Las opciones de conexión

Las opciones de conexión a una instancia PostgreSQL son comunes para todos los clientes, porque dependen del protocolo cliente/servidor. Estas opciones son:

- El nombre del host o la dirección IP de la instancia PostgreSQL. Este argumento también puede ser la ruta del socket Unix.
- El puerto TCP de la instancia PostgreSQL. El valor por defecto es 5432.
- El nombre de la base de datos en la instancia a la que queremos conectarnos. En efecto, nos conectamos siempre a una base de datos de una instancia y no a una única instancia.
- El rol interno de la instancia PostgreSQL.

Las siguientes opciones se pueden utilizar para abrir una conexión con un servidor:

- `-h nombrehost`, `--host nombrehost`: indica el nombre del host o la dirección IP del servidor. Por lo tanto, esta opción permite conectarse a un servidor remoto, es decir, diferente del host en el que se ejecuta el cliente.
- `-p port`, `--port port`: indica el puerto TCP sobre el que abrir la conexión. Por lo tanto, este es el puerto utilizado por el servidor. El puerto por defecto es 5432, pero se puede usar otro valor, según la configuración del servidor.
- `-U nombreusuario`, `--username nombreusuario`: indica el nombre del rol utilizado para abrir la conexión.
- `-W`, `--password`: indica al cliente que solicite, de manera interactiva, una contraseña al usuario. Es posible almacenar la contraseña en la variable de entorno `PGPASSWORD` o utilizar el archivo de contraseñas para evitar tener que escribir la contraseña.
- `-d <database>`: indica la base de datos a la que conectarse.

El siguiente ejemplo abre una conexión con la cuenta de usuario `postgres`, en la base de datos `postgres`, sin proporcionar la contraseña y sin indicar la dirección del servidor. Implícitamente, `psql` abrirá una conexión pasando por el socket Unix. Por lo tanto, en el mismo host:

```
[postgres]$ psql -U postgres -d postgres
postgres=#
```

La opción `-U` permite indicar el nombre de usuario existente en la base de datos. Sin esta opción, se utiliza el nombre de usuario del sistema operativo que lanza `psql`, como en el siguiente ejemplo:

```
[postgres]$ psql -d postgres
postgres=#
```

El último argumento es el nombre de la base de datos. Cuando este argumento no se proporciona, `psql` utiliza el nombre del usuario como nombre de base de datos. En el siguiente ejemplo se crea una conexión como usuario `postgres` a la base de datos `postgres`, pasando por el socket Unix:

```
[postgres]$ psql
postgres=#
```

Estos ejemplos permiten conectarse a una instancia del mismo sistema y utilizando implícitamente el socket Unix creado por la instancia.

a. Variables de entorno

Estos argumentos se pueden especificar por medio de variables de entorno. Es suficiente con guardar estas variables en el intérprete de comandos del sistema donde se lanza `psql` para que se interpreten:

- `PGDATABASE`: nombre de la base de datos a la que conectarse.
- `PGHOST` y `PGHOSTADDR`: nombre y dirección IP del servidor. `PGHOST` puede comenzar por una barra oblicua y en este caso se interpreta como el directorio del socket Unix.
- `PGPORT`: número del puerto TCP del servidor.
- `PGUSER`: nombre del usuario.
- `PGPASSWORD`: contraseña del usuario.

b. Cadena de conexión

Igual que sucede en diferentes lenguajes de programación, es posible utilizar una cadena de conexión con los clientes en línea de comandos. Esta cadena de conexión utiliza las palabras clave que se corresponden con las opciones en línea de comandos.

Además, algunas opciones de conexión solo están disponibles en este modo. Las opciones de conexión son:

- `host` y `hostaddr`: nombre de host o dirección IP.
- `port`: número de puerto TCP.
- `dbname`: nombre de la base de datos.
- `user`: nombre de rol.
- `password`: contraseña.
- `passfile`: archivo de contraseñas si deseamos utilizar un archivo diferente del especificado por defecto.

- `sslmode`: define el modo de seguridad SSL de la conexión. Se aceptan los siguientes valores:
  - `disable`: sin conexión SSL.
  - `allow`: intento de conexión no SSL y después, a no ser que se indique lo contrario, una conexión SSL.
  - `prefer`: intento de una conexión SSL y después, a no ser que se indique lo contrario, una conexión no SSL. Se trata del comportamiento por defecto.
  - `require`: intento de una única conexión SSL.
- `application_name`: etiqueta que identifica la aplicación.
- `client_encoding`: juego de carácter del cliente. El valor `auto` permite determinar el juego de caracteres en función del entorno del cliente.

Ahora se puede utilizar la cadena para abrir una conexión, como en el siguiente ejemplo:

```
psql "host=localhost port=5432 user=postgres dbname=postgres"
postgres=#
```

También es posible definir la cadena en forma de URI:

```
postgresql://user:password@host:port/dbname?param=value
```

Por ejemplo:

```
psql "postgresql://postgres@localhost:5432/postgres"
```

c. Archivo de servicio

El archivo de servicio permite guardar los argumentos anteriores en una de sus secciones del archivo de servicio y utilizar este servicio desde el cliente para abrir una conexión. El archivo de servicio se define en el sistema desde el que se conectan las aplicaciones. Este archivo puede ser global, válido para todos los usuarios del sistema, o local, para un usuario dado.

El nombre del archivo «usuario» es: `.pg_service.conf` en el directorio usuario. El archivo global se define durante la compilación y, por lo tanto, depende de la manera en la que se instala la parte cliente de PostgreSQL, en este caso concreto. En un sistema Debian, el archivo debe ser `/etc/postgresql-common/pg_service.conf`, y en un sistema RedHat o Centos `/etc/sysconfig/pgsql/pg_service.conf`.

El archivo puede contener varias secciones identificadas con un nombre utilizando una sintaxis similar a los archivos INI. Por ejemplo, el siguiente extracto permite conectarse a la instancia local:

```
[cnxlocal]
user=postgres
dbname=postgres
host=localhost
port=5432
```

Entonces, el cliente utiliza el argumento service para conectarse:

```
psql "service=cnxlocal"
```

También es posible definir el servicio en la variable de entorno `PGSERVICE`.

d. Host múltiples

Desde la versión 10 de PostgreSQL, es posible definir varios hosts durante la conexión del cliente, lo que permite paliar la no disponibilidad de un host, pasando al host siguiente de la lista. Es posible definir los nombres de hosts, las direcciones IP y los números de puerto, por ejemplo:

```
psql "host=server1,server2,server3 user=postgres dbname=postgres"
```

Siguiendo el orden de la lista, se utiliza el primer host disponible.

El argumento `target_session_attrs` con valor `read-write` permite indicar que la conexión que se debe utilizar debe estar en modo lectura/escritura. En el caso de red de instancias PostgreSQL en replicación (ver capítulo Replicación), es posible listar todas las instancias replicadas y el cliente prueba, durante la apertura de la sesión, el estado de la instancia para determinar y después utilizar la instancia principal. Esta técnica permite no tener que gestionar el cambio de topología del punto de vista del cliente, incluso si tiene el inconveniente de prolongar el tiempo de apertura de la conexión.

e. Archivo de contraseñas

Es posible guardar las contraseñas en un archivo dedicado para evitar tener que indicarlas sistemáticamente y también que las contraseñas se escriban en el código de las aplicaciones. Este archivo se sitúa en el directorio personal del usuario del sistema operativo que lanza `psql`.

Con un sistema Unix, el archivo se llama `.pgpass` y se sitúa en el directorio del usuario, accesible con el acceso directo ~, es decir `~/.pgpass`.

Con un sistema Windows, el archivo se llama `pgpass.conf` y se encuentra en el directorio `%APPDATA%\postgresql`.

Este archivo contiene tantas líneas como combinaciones de conexión existan. El modelo de una línea es el siguiente:

```
nombrehost:puerto:database:nombreusuario:contraseña
```

Es posible escribir los cuatro primeros argumentos con el carácter genérico *. Es suficiente con escribir los registros más específicos primero, seguidos por las líneas más genéricas para evitar cualquier ambigüedad en la interpretación del contenido. Por ejemplo, la siguiente línea indica la contraseña (`passpg`) del usuario `postgres` para cualquier servidor PostgreSQL:

```
*:*:*:postgres:passpg
```

Si la contraseña es diferente para un servidor particular, entonces la siguiente línea se debe posicionar antes:

```
192.168.0.2:5432:*:postgres:pgpass
```

Por razones de seguridad, este archivo debe tener el acceso en modo lectura y escritura restringido al propietario. Esto se puede realizar con el siguiente comando bajo Unix:

```
chmod 600 ~/.pgpass
```

Sin estas restricciones, el archivo no se utilizará por `psql`.

2. Las herramientas cliente

a. La herramienta psql

La herramienta `psql` se proporciona con las herramientas del servidor PostgreSQL desde el momento en el que el servidor se instala correctamente; está disponible siempre. Se trata de una herramienta en línea de comandos que funciona en modo interactivo, es decir, que permite una interacción con un usuario, o en modo no interactivo, por ejemplo interpretando el contenido de un archivo.

`psql` tiene dos funciones:

- Permite enviar instrucciones al servidor, por ejemplo sentencias SQL o comandos de administración (creación de tablas, de usuarios, etc.).
- Permite interpretar las instrucciones especiales, que completan el juego de instrucciones SQL del servidor.

`psql` es una herramienta cliente: se conecta al servidor como cualquier cliente, es decir, proporcionando un nombre de usuario, una contraseña si es necesaria y el nombre de una base de datos. A partir de estos datos y del host desde el que se ejecuta `psql`, el servidor acepta la conexión o no.

- `-l`, `--list`: lista las bases de datos existentes en el servidor.
- `-e`: muestra todas las consultas SQL enviadas al servidor.
- `-E`: muestra todas las consultas SQL realizadas por los comandos `\d` enviadas al servidor.
- `-f nombrearchivo`, `--file nombrearchivo`: interpreta las instrucciones del archivo `nombrearchivo`, después sale del programa.
- `-c <consulta>`: ejecuta una consulta SQL.
- `-L <archivo>`: escribe el resultado de las consultas en un archivo, además de en la salida estándar.
- `-o`: redirige el resultado de las consultas a un archivo, como el comando interno `\o`.
- `-s`: ejecuta las consultas una a una, pidiendo al usuario que confirme la ejecución de cada consulta durante la utilización en modo no interactivo.
- `-A`: desactiva la alineación de los datos en la salida. Es equivalente al comando interno `\a`.
- `-t`: desactiva la generación de los encabezados en el resultado de los datos. Es equivalente al comando `\t`.
- `-X`, `--no-psqlrc`: no lee el archivo `.psqlrc`.
- `-s`, `--single-step`: ejecuta el script en modo paso a paso, solicitando confirmación al usuario.
- `-1`, `--single-transaction`: ejecuta el script en una única transacción.

b. Uso en modo interactivo

Una vez conectado en modo interactivo, el usuario obtiene un intérprete que permite rellenar comandos y leer los resultados. Los comandos se dividen en dos familias:

- Las consultas SQL, que se interpretan por el servidor y se pueden utilizar desde cualquier cliente.
- Los comandos especiales, que se interpretan por `psql`.

Las consultas SQL son, por ejemplo, consultas `SELECT` o `UPDATE` o sentencias `CREATE` o `DROP`. No tienen nada específico de `psql`.

Los comandos especiales son accesos directos que empiezan por la barra oblicua inversa (\) y permiten mejorar la interactividad de `psql`. Por ejemplo, el siguiente comando especial muestra el conjunto de espacios de nombres disponibles en la base de datos actual:

```
postgres=# \dn
        Lista de los esquemas
       Nombre       | Propietario
————————————————————+————————————
information_schema  | postgres
pg_catalog          | postgres
pg_toast            | postgres
public              | postgres
(4 registros)
```

Las siguientes líneas indican algunos comandos útiles.

El carácter +, cuando se especifica, permite visualizar los permisos asociados al objeto o las descripciones adicionales. El término modelo se corresponde con un nombre de objeto; es posible utilizar los caracteres genéricos (*, ?).

- `\c [nombredb]`: abre una conexión con otra base de datos.
- `\conninfo`: muestra la información de la conexión actual.
- `\l[+]`: lista las bases de datos del servidor.
- `\du [modelo]`, `\dg [modelo]`: lista todos los roles y grupos.
- `\h [comando]`: muestra la ayuda del comando SQL.
- `\?`: muestra la ayuda de los comandos especiales.
- `\d[+] [modelo]`: muestra el detalle de las relaciones descritas por el modelo.
- `\db[+] [modelo]`: lista todos los espacios de tablas.
- `\dd [modelo]`: muestra las descripciones de los objetos. El modelo permite filtrar por el nombre de los objetos.
- `\dp [modelo]`: lista los permisos.
- `\df[antw][+] [modelo]`: lista las funciones, con sus argumentos y el tipo de retorno. Los caracteres adicionales se pueden limitar respectivamente a los agregados, funciones normales, funciones desencadenadoras y funciones «window».
- `\d[istvmS] [modelo]`: lista de los objetos, según el carácter utilizado:
  - `i`: índice,
  - `s`: secuencia,
  - `t`: tabla,
  - `v`: vista,

- `m`: vista materializada,
- `S`: objeto de sistema.

Es posible utilizar varios caracteres. El carácter `S` permite visualizar solo los objetos de sistema; sin este carácter, solo se verán los objetos que no son de sistema.

- `\de[tsuw][+] [PATTERN]`: lista respectivamente las tablas extranjeras, los servidores, las correspondencias de usuario y los «wrappers».
- `\dn[+] [modelo]`: lista todos los esquemas o espacios de nombres disponibles.
- `\dp [modelo]`: lista los permisos de las tablas, vistas y secuencia identificadas por el modelo.
- `\timing`: muestra el tiempo de ejecución de las consultas. Este comando activa o desactiva la visualización, según el estado actual.
- `\watch [N]`: ejecuta la última consulta cada N segundos.
- `\g [FILE]` o `\gset [PREFIX]` o`;`: ejecuta la consulta SQL y, en los casos de `\g` y `\gset`, escribe el resultado en un archivo o una variable.
- `\echo [STRING]`: escribe la cadena en la salida estándar.
- `\i archivo`: ejecuta los comandos desde un archivo.
- `\ir archivo`: como `\i`, pero relativa al script actual.
- `\o [FILE]`: escribe el conjunto de los resultados de las consultas en el archivo.
- `\qecho [STRING]`: escribe la cadena en el flujo de salida, utilizable con `\o`.
- `\e [archivo] [línea]`: edita la última consulta o el archivo indicado con un editor externo.
- `\ef [función [línea]]`: edita la función indicada en un editor externo.
- `\ev [vista [línea]]`: edita la vista identificada en un editor externo.
- `\p`: muestra la última consulta.
- `\g [|comando]` o `[archivo]`: ejecuta la consulta actual o la última consulta y escribe el resultado en un PIPE en un comando o archivo.
- `\gexec`: ejecuta la consulta actual o la última consulta y trata cada valor del resultado como una consulta, que se debe ejecutar. El resultado debe contener sentencias SQL válidas.
- `\r`: vacía la zona de memoria que contiene la última consulta.
- `\s [archivo]`: muestra el histórico de las consultas o, cuando se indica un archivo, registra el histórico en este archivo.
- `\w archivo`: registra la zona de memoria que contiene la última consulta en un archivo.
- `\copy ...`: ejecuta el comando `copy` hacia y desde el cliente y no a un archivo en el lado servidor.
- `\a`: cambia la alineación de visualización, activado por defecto. Es equivalente al argumento `-A` del comando.
- `\t`: cambia la visualización de los encabezados (nombres de los atributos). Es equivalente al argumento `-t` del comando.

- `\f [cadena]`: muestra o define el separador de campos para el modo no alineado.
- `\H`: cambia el modo de visualización HTML, desactivado por defecto.
- `\T [cadena]`: define los atributos del marcador `<tabla>` en la visualización HTML.
- `\pset [{format|border|expanded|fieldsep|fieldsep_cero|footer|null|numeri tuples_only|title|tableattr|pager} [on|off]]`: define las opciones de formateo indicadas.
- `\set [nombre [valor]]`: define una variable interna de `psql`. Si el comando se lanza sin argumentos, muestra las variables ya definidas.
- `\ef [función [línea]]`: edita la función identificada en un editor externo.
- `\if`, `\elif`, `\else`, `\end`: permite definir bloques de instrucciones condicionales utilizando las variables internas definidas por `\set` como expresiones.
- `\x [on|off|auto]`: cambia la visualización extendida.
- `\q`: sale de `psql`.

c. Uso en modo no interactivo

La herramienta `psql` también se puede utilizar en modo no interactivo; es decir, que las instrucciones que se deben interpretar se leen desde un archivo o un flujo, sin intervención del usuario. Este modo es útil durante la utilización de scripts de automatización de tareas, que funcionan sin la intervención del usuario.

Una primera opción útil es `-f`. Cuando se pasa como argumento de `psql`, las instrucciones se leen desde un archivo:

```
$ psql -f archivo.sql
```

El equivalente de este comando podría ser:

```
$ psql < archivo.sql
```

También es posible leer la salida de otro comando, utilizando la redirección de flujo:

```
$ echo 'select 42' | psql -f -
```

o:

```
$ echo 'select 42' | psql
```

Las dos secuencias son equivalentes, pero `psql` se comporta de manera diferente: en el caso de que se use la opción `-f`, seguida de un guión (`-`), `psql` muestra los mensajes de errores con los números de registros.

Las instrucciones se interpretan de la misma manera.

La opción `-c` permite ejecutar solo un comando SQL, sin tener que leerlo desde un archivo o en la entrada estándar, utilizando los argumentos de control de salida, por ejemplo:

```
$ psql -d clientes -At  -c 'select count(*) from clientes'
42
```

d. Archivo de configuración

La herramienta `psql` se puede configurar por medio del archivo `.psqlrc`, ubicado en la raíz del directorio personal del usuario de sistema. Contiene diferentes instrucciones. El principio es el mismo que para el archivo `.bashrc` del shell bash. Los ajustes permiten personalizar el comportamiento interactivo de `psql`. Por ejemplo, es posible modificar el registro de comandos ajustando los argumentos `PROMPT1` y `PROMPT2`, modificar el histórico de los comandos SQL o definir comandos SQL en una variable.

La línea de comandos se puede ajustar con la línea por defecto `PROMPT1` o `PROMPT2`, que es la que se obtiene durante la entrada de un comando SQL en varias líneas, antes del final del comando (generalmente un punto y coma).

Por ejemplo, la siguiente definición de `PROMPT1` permite visualizar en orden: el nombre de usuario (`%n`), el nombre del host (`%m`), el puerto TCP (`%>`) y la base de datos actual (`%~`):

```
\set PROMPT1  '%n@%m:%>/%~%x%# '
```

El símbolo `%x` agrega un asterisco al prompt cuando se abre una transacción con el comando `BEGIN`. El símbolo `%#` muestra el carácter # cuando el usuario de la sesión es un superusuario, y el carácter >, en el resto de los casos.

Es posible modificar el comportamiento del histórico cambiando su tamaño, ignorando los comandos que empiezan por un espacio o los comandos duplicados (o los dos):

```
HISTSIZE=2000
HISTCONTROL= {ignoredups | ignorespace | ignoreboth}
```

También es posible definir comandos SQL para beneficiarse de la variable con objeto de, por ejemplo, llamar de manera rápida al comando en modo interactivo:

```
\set locks 'select n.nspname, c.relname, l.pid, l.mode, l.granted
from pg_locks l join pg_class c on l.relation=c.oid join
pg_namespace n on c.relnamespace=n.oid;'
```

A continuación, es suficiente con llamar en línea de comandos a la variable: `locks`.

Permisos de acceso

La cuenta de usuario creada inicialmente es una cuenta con permisos: tiene todos los permisos sobre los objetos (bases de datos, tablas, usuarios, etc.) de la instancia. Se trata del administrador del servidor. Es el equivalente de la cuenta root de un sistema Unix. El nombre de esta cuenta depende de las opciones elegidas durante la instalación y la inicialización de PostgreSQL y la instancia. Esta cuenta generalmente tiene el nombre postgres como nombre de cuenta del usuario de sistema.

1. Administración de los roles: usuarios y grupos

Una cuenta de usuario también se llama rol. Un rol es un objeto global, es decir, que es válido para toda la instancia. Un rol tiene permisos sobre los objetos; permite abrir conexiones, forma parte del resto de los roles o contiene otros roles. Un rol reúne las nociones de usuarios y grupos.

El comando que sirve para crear roles es `create role`. Sustituye a los comandos `create user` y `create group` de las versiones anteriores (por razones de compatibilidad, estos comandos siempre están disponibles).

Este comando también es accesible desde un programa cliente, `createuser`, que permite crear roles desde el intérprete de comandos del sistema operativo, por ejemplo en un script.

La sinopsis del comando SQL es la siguiente:

```
CREATE ROLE nombrerol [ [ WITH ] option [ ... ] ]
```

La sinopsis del comando de sistema es:

```
[postgres]$ createuser [opción...] [nombrerol]
```

Las opciones son las siguientes, con la opción del comando SQL seguida del equivalente para el comando de sistema.

- `LOGIN`, `-l`: indica si el rol se puede conectar al servidor. El rol se convierte en el equivalente de una cuenta de usuario.
- `CONNECTION LIMIT límite_conexión`, `-c number`: indica el número máximo de conexiones simultáneas de un rol. El valor por defecto es (`-1`), que no fija ningún límite.
- `[ ENCRYPTED | UNENCRYPTED ] PASSWORD 'contraseña'`, `-P`, `-E`, `-N`: define la contraseña de un rol. Las palabras clave `ENCRYPTED`, `UNENCRYPTED` indican si se debe cifrar la contraseña proporcionada o no.
- `VALIDUNTIL 'horafecha'`: indica la fecha y hora del final de validez de la contraseña. Por defecto, una contraseña es válida indefinidamente.
- `SUPERUSER`, `-s`: indica que el rol va más allá del sistema de permisos. Por lo tanto, el rol tiene todos los permisos sobre el grupo de bases de datos. Esta opción no está activada por defecto.
- `CREATEDB`, `-d`: indica que el rol puede crear bases de datos en la instancia; no es el caso por defecto.
- `CREATEROLE`, `-r`: autoriza al rol a crear otros roles. Por defecto, un rol no puede crear otros roles.
- `IN ROLE nombrerol [, ...]`: indica el rol o los roles cuyo nuevo rol es miembro.
- `ROLE nombrerol [, ...]`: indica el rol o los roles que se convierten en miembros del nuevo rol. De esta manera, este nuevo rol se convierte en el equivalente de un grupo.
- `INHERIT`, `-i`: permite al rol heredar los permisos de los roles de los que es miembro. Por defecto, un rol no hereda los permisos.
- `REPLICATION`,`--replication`: permite que el rol se utilice para poner en marcha la replicación integrada en PostgreSQL. Por defecto, un rol no puede hacer replicación, salvo un rol SUPERUSER.
- `BYPASSRLS`: permite ir más allá de las reglas de acceso a los registros de datos (ver la sección Seguridad de acceso a los registros de datos). Por defecto se aplican estas reglas, salvo para un rol SUPERUSER o cuando el rol es propietario de la tabla.

a. Definición de un rol como cuenta de usuario

La sentencia SQL `CREATE ROLE` crea una cuenta de usuario, es decir, una cuenta que permite a un usuario conectarse con el servidor. La opción `LOGIN` es un permiso que autoriza la conexión y la opción `PASSWORD` permite indicar la contraseña en los casos en los que se solicite. El siguiente ejemplo ilustra la creación de una cuenta de usuario con una contraseña, así como el permiso para crear las bases de datos:

```
postgres=# CREATE ROLE slardiere LOGIN PASSWORD 'password'
CREATEDB;
```

Una vez que se crea el rol, se le pueden asignar permisos con los comandos `GRANT` y `REVOKE`.

b. Definición de un rol como grupo

La creación de un rol sin el permiso `LOGIN` provoca la creación de un grupo. En realidad, un rol que tiene el permiso `LOGIN` también puede ser un grupo, lo que permite una anidación de los grupos. La opción `ROLE` permite, desde la creación, indicar cuáles son los roles existentes que se han convertido en miembros de este nuevo grupo.

El siguiente ejemplo ilustra la creación de un grupo con la declaración de un rol como miembro:

```
postgres=# CREATE ROLE admingroup ROLE slardiere;
```

c. Pertenencia a un grupo y herencia

La pertenencia a un grupo no hace que se hereden los permisos del grupo. Existe una opción que permite hacer explícita esta pertenencia. El siguiente ejemplo muestra la creación de una cuenta de usuario, con una contraseña y heredando los permisos de un grupo del que es miembro:

```
postgres=# CREATE ROLE slardiere LOGIN PASSWORD
'password'  INHERIT IN ROLE admingroup;
```

d. Uso de los permisos de un grupo

Cuando esta noción de herencia no se utiliza durante la creación del rol, lo que representa el caso por defecto, los permisos del grupo no se transmiten al miembro. Es necesaria una etapa adicional explícita para obtener estos permisos: el comando `SET ROLE` permite cambiar de identidad y, por lo tanto, obtener los permisos de esta identidad.

Cuando se crea un usuario con el siguiente comando:

```
postgres=# CREATE ROLE sebl LOGIN PASSWORD
'password'  IN ROLE admingroup;
```

El usuario `sebl` no obtiene los permisos del grupo `admingroup`. Es suficiente con lanzar la siguiente sentencia para obtenir los permisos del grupo:

```
sebl=> SET ROLE admingroup;
```

Los comandos `SET ROLE NONE` y `RESET ROLE` permiten volver al estado inicial, es decir, únicamente con los permisos del usuario.

e. Modificación de un rol

La modificación de un rol se puede realizar con la sentencia SQL `ALTER ROLE`. Todas las opciones de la sentencia `CREATE ROLE` son modificables con la sentencia `ALTER ROLE`, con la condición de tener los permisos necesarios. Es suficiente con utilizar la opción que se ha de modificar para añadir el permiso o utilizar el prefijo `NO` para retirarlo. Por ejemplo, para prohibir la conexión de una cuenta de usuario, es suficiente con modificar el rol correspondiente con la opción `NOLOGIN`:

```
ALTER ROLE slardiere NOLOGIN;
```

Las siguientes líneas indican las opciones disponibles:

- `SUPERUSER` o `NOSUPERUSER`
- `CREATEDB` o `NOCREATEDB`
- `CREATEROLE` o `NOCREATEROLE`
- `INHERIT` o `NOINHERIT`
- `LOGIN` o `NOLOGIN`
- `BYPASSRLS` o `NOBYPASSRLS`
- `REPLICATION` o `NOREPLICATION`

Las siguientes opciones permiten restaurar los valores:

- `CONNECTION LIMIT límiteconexión`
- `[ ENCRYPTED | UNENCRYPTED ] PASSWORD 'contraseña'`
- `VALID UNTIL 'fechahora'`

El siguiente ejemplo ilustra el cambio de la contraseña de una cuenta de usuario:

```
ALTER ROLE slardiere PASSWORD 'nuevapassword' ;
```

También es posible renombrar una cuenta con la opción `RENAME TO`, como en el siguiente ejemplo:

```
ALTER ROLE slardiere RENAME TO sebl;
```

f. Variables de sesión

Otra función importante de la sentencia `ALTER ROLE` es la posibilidad de inicializar las variables de sesión propias del rol, de tal manera que la variable se inicialice durante la apertura de la sesión. Por ejemplo, la variable `DateStyle` permite seleccionar el formato de visualización de las fechas. El usuario puede asignar un valor concreto para encontrar el comportamiento en todas las sesiones:

```
ALTER ROLE slardiere set datestyle = 'dmy';
```

La lista de las variables está disponible con el comando `SHOW ALL`, pero no todas las variables son modifiables en una sesión.

g. Eliminación de un rol

La eliminación de un rol se puede realizar con la sentencia SQL DROP ROLE. El rol que se ha de eliminar no debe ser propietario de un objeto del grupo de bases de datos. Por lo tanto, es necesario cambiar inicialmente la pertenencia de los objetos. El siguiente ejemplo muestra la eliminación de un rol:

```
DROP ROLE slardiere;
```

h. Administración de la pertenencia a un rol

Otro uso de los comandos GRANT y REVOKE es la adición y eliminación de un rol como miembro de otro rol:

```
GRANT role [, ...] TO nombrerol [, ...] [ WITH ADMIN OPTION ]
```

y:

```
REVOKE [ ADMIN OPTION FOR ] role [, ...] FROM nombrerol
[, ...] [ CASCADE | RESTRICT ]
```

Por lo tanto, estos comandos son adicionales al comando CREATE ROLE.

2. Asignación y revocación de permisos

Para cada rol y para cada objeto del grupo de bases de datos, se pueden añadir o eliminar permisos particulares.

La adición de permisos adicionales se realiza con el comando GRANT, y la eliminación de permisos, con el comando REVOKE.

El comando GRANT se descompone en varias partes.

Después del nombre del comando, el usuario indica la lista de permisos y a continuación, utilizando como prefijo la palabra clave ON, los objetos sobre los que se aplican estos permisos. Para terminar, el usuario indica a qué cuenta de usuario o grupo se aplican estos permisos. La lista de los permisos difiere según el objeto utilizado. La siguiente tabla resume el conjunto de los permisos que se pueden utilizar, ordenados por tipo de objeto para los comandos GRANT y REVOKE.

Tipo del objeto	Permisos	Descripción
Todos	all	Autoriza todos los permisos disponibles según el tipo del objeto.
Tabla	select	Autoriza la ejecución de consultas `SELECT` sobre la tabla.
Tabla	insert	Autoriza la ejecución de consultas `INSERT` sobre la tabla.
Tabla	update	Autoriza la ejecución de consultas `UPDATE` sobre la tabla.
Tabla	delete	Autoriza la ejecución de consultas `DELETE` sobre la tabla.
Tabla	rule	Autoriza la creación de reglas sobre la tabla.
Tabla	referencias	Autoriza la utilización de claves extranjeras desde o hacia la tabla.
Tabla	trigger	Autoriza la creación de triggers sobre la tabla.
Columna	select	Autoriza la ejecución de consultas `SELECT` sobre la columna.
Columna	insert	Autoriza la ejecución de consultas `INSERT` sobre la columna.
Columna	update	Autoriza la ejecución de consultas `UPDATE` sobre la columna.
Database	create	Autoriza la creación de esquemas en la base de datos.
Database	temporary, temp	Autoriza la creación de tablas temporales.
Function	execute	Autoriza la utilización de la función.
Language	usage	Autoriza la utilización del lenguaje procedural.
Schema	create	Autoriza la creación de tablas en el esquema.
Schema	usage	Autoriza el acceso a los objetos contenidos en el esquema.
Tablespace	create	Autoriza la creación de tablas e índices en el espacio de tablas.

Por lo tanto, la sinopsis de los comandos en su forma más sencilla es:

```
GRANT permisos [, permisos] ON tipoobjeto nombreobjeto
[, nombreobjeto] TO  { nombreusuario [, nombreusuario] | PUBLIC }
[ WITH GRANT OPTION ]
```

y:

```
REVOKE permisos [, permisos] ON tipoobjeto nombreobjeto
[, nombreobjeto] FROM { nombreusuario [, nombreusuario]
| PUBLIC }  [ CASCADE | RESTRICT ]
```

En lo que respecta al objeto TABLE, es posible identificar todas las tablas de un esquema utilizando la sentencia `ALL TABLES IN SCHEMA nombre`.

La palabra clave `PUBLIC`, en lugar del nombre de cuenta de usuario, se corresponde con todas las cuentas de usuario, incluidas las creadas después de la asignación de los permisos.

En el caso del comando `GRANT` con la opción `WITH GRANT OPTION`, el que recibe los permisos y los puede transmitir a su vez, lo que permite delegar la administración de los permisos en un objeto. Por lo tanto, un usuario A puede acordar los permisos de un usuario B, dándole el permiso de transmitirlos. A su vez, el usuario B puede transmitir estos permisos a un usuario C, sin que el usuario C tenga el permiso de transmitirlos.

En los casos de asignación de permisos a una o varias columnas de una tabla, la sintaxis hace que sea necesario poner entre paréntesis a continuación de la lista de las columnas de los permisos asignados, antes de la sentencia `ON` que indica el nombre de la tabla.

A la inversa, en caso del comando `REVOKE`, un permiso solo se puede retirar por el usuario que lo asignó. La opción `CASCADE` permite a un usuario retirar todos los permisos sobre un objeto, incluido a un usuario al que no fue asignado el permiso. En el ejemplo anterior, el usuario A podría retirar sus permisos al usuario B y al usuario C, añadiendo la palabra clave `CASCADE` en el comando `REVOKE`.

El siguiente ejemplo muestra la asignación de permisos al usuario `sebl` sobre las tablas de la base de datos de clientes:

```
clientes=# GRANT SELECT, INSERT, UPDATE,
DELETE ON TABLE clientes, contactos, facturas,
registros_facturas, prestaciones TO sebl;
```

El siguiente ejemplo muestra la retirada del permiso `SELECT` sobre la tabla `facturas` al usuario `sebl`:

```
clientes=# REVOKE SELECT ON TABLE clientes FROM sebl;
```

3. Definición de los permisos por defecto

Es posible definir permisos por defecto a los objetos que se crean. La sentencia `ALTER DEFAULT PRIVILEGES` permite guardar los permisos deseados y, de esta manera, simplificar la creación de los objetos. Es posible guardar los permisos por defecto para las tablas, secuencias, funciones, tipos y esquemas. Este ajuste por defecto se guarda para un usuario dado y puede identificar un esquema concreto.

La sinopsis del comando es la siguiente:

```
ALTER DEFAULT PRIVILEGES
   [ FOR { ROLE | USER } role [, ...] ]
   [ IN SCHEMA schema [, ...] ]
   grant_or_revoke
```

Si no se indica el rol, se utiliza el rol actual. El siguiente ejemplo muestra el registro de los permisos `SELECT`, `INSERT`, `UPDATE` y `DELETE` para las futuras tablas del rol `sebl`:

```
ALTER DEFAULT PRIVILEGES GRANT SELECT, INSERT, UPDATE,DELETE
ON TABLES TO sebl;
```

4. Seguridad de acceso a los registros de datos

Desde la versión 9.5 de PostgreSQL, es posible seleccionar los registros devueltos por una consulta en función del rol que ejecuta esta consulta. Por lo tanto, es posible prohibir el acceso a algunos registros de una tabla a un usuario dado. Al contrario de lo que sucedía con los permisos establecidos por los comandos `GRANT` y `REVOKE`, la consulta no provoca ningún error, sino que devuelve los registros autorizados por las reglas establecidas.

Estas reglas se añaden a los permisos dados por el comando `GRANT`. Por lo tanto, es necesario haber establecido los permisos necesarios antes de implementar estas reglas.

Las reglas no se aplican a los roles que se benefician de los permisos `SUPERUSER` o `BYPASSRLS`. Por defecto, las reglas están desactivadas para todas las tablas.

a. Activación

Para cada una de las tablas para las que queremos utilizar esta funcionalidad, es necesario activar la aplicación de las reglas con el comando `ALTER TABLE`. El siguiente ejemplo activa esta funcionalidad para la tabla `prestaciones`:

```
ALTER TABLE prestaciones ENABLE ROW LEVEL SECURITY;
```

b. Creación de las reglas de acceso

Es posible definir varias reglas por tabla en función de las sentencias y los roles. Además, las reglas se pueden combinar entre ellas. La sinopsis del comando `CREATE POLICY` es la siguiente:

```
CREATE POLICY nombre ON table
   [ AS { PERMISSIVE | RESTRICTIVE } ]
   [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
   [ TO { role | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
   [ USING ( using_expression ) ]
   [ WITH CHECK ( check_expression ) ]
```

El nombre debe ser único para una tabla dada.

Por defecto, las reglas son `PERMISSIVE`, como si se combinaran con el operador lógico `OR`. Cuando las reglas son `RESTRICTIVE`, se combinan como con el operador `AND`, lo que limita los registros devueltos.

Por defecto, todas las sentencias están afectadas por la regla, pero es posible limitar la aplicación de la regla a una sentencia deseada. También por defecto, la regla se aplica a todos los roles, incluso si es posible listar los roles a los que se aplica.

La sentencia `USING` permite indicar una expresión booleana que implementa la lógica de la regla. La expresión se añade a las consultas que utilizan la tabla. Se devolverán los registros para los que la expresión es verdadera. El resto de los registros se ignoran.

La sentencia `WITH CHECK` permite controlar los valores utilizados con las sentencias `INSERT` y `UPDATE`. Solo se pueden insertar o actualizar los registros que validan la expresión. Se produce un error si la expresión es falsa.

c. Ejemplo de creación de las reglas de acceso

El siguiente ejemplo permite limitar el acceso a los registros de la tabla `prestaciones`, en función de los identificadores de prestaciones asignados en la tabla `intervenciones`.

Se crea la tabla `intervenciones` y se alimenta con las siguientes consultas:

```
create table intervenciones
(
 login text,
 prest_id integer referencias prestaciones (prest_id),
 primary key ( login, prest_id )
);
insert into intervenciones select 'slardiere', prest_id from
prestaciones where cl_nombre = 'SPLFBC';
```

La sentencia `GRANT` permite dar permisos en modo lectura al usuario `slardiere`:

```
grant select on table prestaciones,intervenciones to slardiere;
```

Para terminar, la creación de la regla utiliza la tabla `intervenciones`, con una sentencia que hace la llamada a la tabla `prestaciones` implícitamente:

```
CREATE POLICY intervenciones_p
ON prestaciones
FOR SELECT
USING ((cl_nombre) IN ( SELECT cl_nombre FROM intervenciones i
       where i.prest_id=prestaciones.prest_id
         and i.login = current_user ));
```

Una vez que se ha establecido la regla, las consultas que utilizan la tabla `prestaciones` aplican este filtro, de tal manera que el rol `slardiere` solo puede ver las prestaciones indicadas en la tabla `intervenciones`:

```
slardiere$ select cl_nombre, prest_id, prest_tipo, prest_nombre
from prestaciones order by prest_fecha_inicio desc;
cl_nombre | prest_id | prest_tipo | prest_nombre
----------+----------+------------+----------------
SPLFBC    |    99415 | C          | Consulta
SPLFBC    |    99414 | O          | Otro
SPLFBC    |    99413 | E          | Eliminación
SPLFBC    |    99412 | E          | Entrega
SPLFBC    |    99411 | M          | Mantenimiento
SPLFBC    |    99375 | C          | Consulta
SPLFBC    |    99374 | O          | Otro
SPLFBC    |    99373 | E          | Eliminación
SPLFBC    |    99372 | E          | Entrega
[...]
```

Definición de los datos

Introducción

El lenguaje de definición de los datos se basa en tres sentencias: `CREATE`, `ALTER` y `DROP`, que se aplican a cada objeto de la base de datos. Cada objeto se puede corresponder con un objeto físico, es decir, un archivo en los sistemas de archivos, o a un objeto lógico, es decir, una definición almacenada en el catálogo de la base de datos.

La ejecución de estas sentencias implica necesariamente un bloqueo exclusivo de los objetos afectados. Con este bloqueo exclusivo, no se puede atender ningún otro bloqueo en modo lectura o escritura. Esto no supone ningún problema durante la creación de un objeto, porque ninguna sesión actual lo puede haber reclamado, pero durante la eliminación de un objeto hay que esperar a que todos los bloqueos existentes se liberen. En lo que respecta a una base de datos, por ejemplo, no debe haber sesiones conectadas a esta base de datos.

El impacto más importante se produce durante la modificación de un objeto porque el bloqueo permanece durante todo el tiempo de ejecución del comando y, por lo tanto, esto puede tener un impacto en la ejecución de consultas concurrentes.

En consecuencia, las sentencias de definición de los datos se deben ejecutar con precaución, eligiendo si es posible una ventana de tiempo oportuna.

Los espacios de tablas

Un espacio de tablas es un directorio de un sistema de archivos, en el que PostgreSQL escribe los archivos de las tablas y de los índices. Por defecto, PostgreSQL dispone de un espacio de tablas ubicado en el directorio del grupo de bases de datos. Es posible crear otros espacios de tablas, que permiten al administrador seleccionar de esta manera la ubicación del almacenamiento de una tabla o de un índice.

Hay varios motivos que pueden hacer que un administrador cree un espacio de tablas:

- La partición ya no dispone de suficiente espacio libre. En este caso, los espacios de tablas permiten utilizar varios discos duros diferentes para un mismo grupo de base de datos, sin utilizar sistemas de volúmenes lógicos, como LVM en los sistemas GNU/Linux.
- La utilización de las tablas y de los índices provoca una saturación de las escrituras y lecturas del subsistema de disco en el que se ubica el espacio de tablas por defecto. Incluso en los sistemas de alto rendimiento, el escalado de una aplicación puede hacer que el administrador cree otros espacios de tablas en los subsistemas de discos diferentes. De esta manera, las escrituras y lecturas de archivos de tablas e índices se reparten en varios soportes físicos, mejorando el rendimiento.

Por lo tanto, un espacio de tablas es una herramienta que se puede utilizar por el administrador del servidor de bases de datos, que permite intervenir sobre la ubicación física del almacenamiento de las tablas y de los índices. Esto significa que el administrador puede elegir, tabla por tabla e índice por índice, independientemente de la base de datos, la ubicación de un archivo, optimizando de esta manera los volúmenes utilizados, las escrituras y las lecturas.

Un espacio de tablas no es específico a una base de datos. Forma parte de un grupo de bases de datos por defecto, se utiliza desde todas las bases de datos. Una administración fina de los permisos sobre los espacios de tablas permite al administrador controlar el reparto de los archivos, en función de las bases de datos y los roles utilizados.

La primera etapa antes de inicializar el espacio de tablas desde PostgreSQL es crear un directorio para este uso. Según las necesidades, se puede tratar de un directorio sobre un nuevo subsistema de discos o en una partición de un sub-sistema de discos ya presente. Las siguientes etapas permiten inicializar este directorio y asignar los permisos necesarios para que el usuario `postgres` sea el propietario:

```
[root]# mkdir -p /data2/postgres/tblspc2/
[root]# chown postgres:postgres /data2/postgres/tblspc2/
[root]# chmod 700 /data2/postgres/tblspc2/
```

Si el directorio ya existe, se debe vaciar y pertence al usuario del sistema operativo, que ejecuta la instancia de PostgreSQL.

Cuando se crea el directorio, es suficiente con guardarlo en la instancia de PostgreSQL, dándole un nombre que permita identificarlo. La sinopsis del comando es la siguiente:

```
postgres= # CREATE TABLESPACE nombretblspc [ OWNER nombrerol ]
LOCATION 'directorio';
```

Por defecto, el espacio de tablas creado de esta manera pertenece al usuario que ejecuta el comando. Solo un superusuario puede crear un espacio de tablas, pero puede transmitir la pertenencia a otro usuario por medio de la opción `OWNER`.

Una vez que existe el espacio de tablas, se puede utilizar cuando se crean o modifican las tablas e índices.

1. Modificación de un espacio de tablas

Es posible modificar un espacio de tablas existente. Hay dos argumentos modificables: el nombre y el propietario. El comando `ALTER TABLESPACE` permite hacer estas dos modificaciones, como en la siguiente sinopsis:

```
postgres=# ALTER TABLESPACE nombretblspc1 RENAME TO nombretblspc2;
postgres=# ALTER TABLESPACE nombretblspc2 OWNER TO nombrerol;
```

En función de las características del subsistema de discos utilizados por el espacio de tabla, es posible modificar las constantes de coste `seq_page_cost` y `random_page_cost` del planificador de consultas, modificando el coste de lectura de una página en disco:

```
postgres=# ALTER TABLESPACE nombretblspc2 set (seq_page_cost =
3, random_page_cost = 1 );
postgres=# ALTER TABLESPACE nombretblspc2 reset (seq_page_cost);
```

El significado de estas constantes se presenta en el capítulo Explotación.

2. Eliminación de un espacio de tablas

La eliminación de un espacio de tablas es posible si no existe ninguna tabla ni índice en este espacio de tablas, incluidos en una base de datos diferente a la de la conexión actual.

Antes de eliminar el espacio de tablas, hay que haber movido a otro espacio de tablas todos los objetos contenidos, incluidos los que no pertenezcan a la base de datos actual. Una vez que el espacio de tablas está vacío, la sentencia `DROP TABLESPACE` permite esta eliminación, como muestra la siguiente sinopsis:

```
postgres=# DROP TABLESPACE [ IF EXISTS ] nombretblspc;
```

Una vez que el espacio de tablas se elimina en la instancia de PostgreSQL, el directorio del sistema de archivos ya no es útil y se puede eliminar. La opción `IF EXISTS` evita que se provoque un error durante la eliminación si el espacio de tablas no existe.

Las bases de datos

En una instancia de PostgreSQL, una base de datos es un contenedor. Contiene los esquemas, las tablas, los índices y todos los objetos útiles para una aplicación. También recibe las conexiones desde las aplicaciones cliente. En efecto, cuando se abre una conexión en una base de datos particular, no es posible utilizar directamente los objetos creados en otras bases de datos.

Por lo tanto, es importante repartir correctamente los objetos y los datos de las aplicaciones en las bases de datos, principalmente utilizando la noción de esquema. La creación de una base de datos se puede realizar con la sentencia `CREATE DATABASE` o con el comando del sistema operativo `createdb`.

Algunos argumentos permiten personalizar la creación de una base de datos:

- Para crear una base de datos, es necesario ser superusuario o tener el permiso `CREATEDB`. Por el contrario, es posible transmitir la pertenencia a un usuario que no tiene permisos con la opción `OWNER`. La opción del comando `createdb` es `-O` o `--owner`.
- La base de datos `template1` sirve de modelo por defecto para la creación de otra base de datos. Para cada base de datos creada con este modelo, se hace una copia de `template1`, para que todos los objetos creados en `template1` se dupliquen en la nueva base. La base `template0` funciona de la misma manera, pero no es posible crear objetos en esta base-modelo: `template0` es una base de datos virgen. Es posible seleccionar la base de datos que sirve de modelo con la opción `TEMPLATE` y, por supuesto, es posible seleccionar cualquier base de datos modelo existente. La opción del comando `createdb` es `-T` o `--template`.
- Un argumento importante durante la creación de una base de datos es la elección del juego de caracteres. Determina la manera en la que los datos se almacenarán en las tablas y los índices. El juego de caracteres por defecto se determina durante la creación del grupo de base de datos, pero es posible indicar otro juego de caracteres con la opción `ENCODING`. No es posible modificar esta opción una vez que la base de datos se crea. Para seleccionar otra codificación diferente a la dada por defecto, es necesario utilizar la base de datos modelo `template0`, dando por hecho que el resto de las bases de datos modelo pueden contener datos incompatibles con la nueva codificación. La opción del comando `createdb` es `-E` o `--encoding`.
- Los argumentos `LC_COLLATE` y `LC_CTYPE` tienen un impacto en la localización de las ordenaciones de las cadenas de caracteres, los índices o durante la utilización de la sentencia `ORDER BY` de una consulta.
- El espacio de tablas por defecto es el que se ha creado durante la inicialización del grupo de bases de datos. Es posible crear otros espacios de tablas y asociarlos con una base de datos con la opción `TABLESPACE`. Pero esta opción solo es un argumento por defecto para la creación de las tablas e índices, que puede no servir. La opción del comando `createdb` es `-D` o `--location`.
- El último argumento, `CONNECTION LIMIT`, permite controlar el número de conexiones entrantes, que es ilimitado por defecto.

La siguiente sinopsis muestra la sentencia SQL que permite crear una base de datos:

```
postgres=# CREATE DATABASE nombre [ [ WITH ] [ OWNER [=] rol ]
[ TEMPLATE [=] modelo ] [ ENCODING [=] codificación ] [ LC_COLLATE [=]
lc_collate ] [ LC_CTYPE [=] lc_ctipo ] [ TABLESPACE [=] espaciotabla ]
[ CONNECTION LIMIT [=] límite_conexión ] ]
```

El siguiente comando permite crear una base de datos llamada `clientes` con el juego de caracteres UTF8:

```
postgres=# CREATE DATABASE clientes OWNER sebl ENCODING 'UTF8';
```

La siguiente sinopsis muestra el comando del sistema operativo:

```
[postgres]# createdb [-D tablaespacio|--tablespace=tablaespacio]
[-E codificación|--encoding=codificación] [--lc-collate=locale]
[--lc-ctype=locale] [-O owner|--owner=owner]
[-T modelo|--template= modelo] [nombrebase] [descripción]
```

Por lo tanto, la creación de la base `clientes` se puede escribir como se muestra a continuación:

```
[postgres]# createdb -O sebl clientes
```

Con el comando `createdb`, es posible conectarse a un servidor remoto utilizando las mismas opciones que el comando `psql`. Por ejemplo, el siguiente comando crea una base de datos en un servidor PostgreSQL remoto:

```
[root]# createdb -E UTF8 -h 192.168.0.3 -p 5432 -U postgres clientes
```

La elección de un juego de caracteres diferente al dado por defecto se hace utilizando la base de datos modelo `template0`. En función de los ajustes del sistema operativo y de las opciones elegidas durante la inicialización del grupo de bases de datos, puede ser necesario no conservar el valor por defecto. En todos los casos, el sistema operativo debe conocer el juego de caracteres seleccionado antes de la creación de la base de datos. El siguiente ejemplo permite crear una base de datos con el juego de caracteres UTF8, elección muy habitual desde hace algunos años:

```
postgres=# CREATE DATABASE clientes TEMPLATE template0 ENCODING
utf8 LC_COLLATE;
```

o:

```
[postgres]# createdb -T template0 -E utf8 clientes
```

Estas opciones no se pueden modificar posteriormente. Es necesario seleccionar correctamente estos valores. De la misma manera, el juego de caracteres `SQL_ASCII`, aunque esté disponible, es muy desaconsejable debido al poco control que permite sobre los datos manipulados.

Una vez que se crea la base de datos, es posible conectarse y crear objetos.

1. Modificación de una base de datos

Una vez creada, una base de datos se puede modificar con la sentencia `ALTER DATABASE`. Se pueden modificar varios datos, entre ellos el nombre, el propietario, el número límite de conexiones e incluso el espacio de tablas. Estas modificaciones solo se pueden hacer por un superusuario o por el propietario de la base de datos.

Las siguientes sinopsis muestran la utilización de la sentencia `ALTER DATABASE`:

```
postgres=# ALTER DATABASE nombre CONNECTION LIMIT límiteconexión
postgres=# ALTER DATABASE nombre RENAME TO nuevonombre
postgres=# ALTER DATABASE nombre OWNER TO rol
postgres=# ALTER DATABASE nombre SET TABLESPACE nombre_tablespace
```

El cambio de espacio de tablas implica el desplazamiento físico de las tablas e índices, salvo aquellos ya asociados a otro espacio de tablas diferente al dado por defecto (`pg_default`). Por lo tanto, este cambio tiene un impacto muy importante sobre el acceso a estos objetos, ya que se autoriza un bloqueo exclusivo todo el tiempo necesario para realizar el desplazamiento. Además, es necesario tener permisos de creación de objetos en el espacio de tablas.

Se pueden añadir otros argumentos, por ejemplo algunas variables de configuración. Estas variables normalmente se asocian a la instancia, en el archivo de configuración `postgresql.conf`, pero se pueden definir de manera diferente con otro valor para la base de datos.

```
postgres=# ALTER DATABASE nombre SET argumento { TO | = } { valor | DEFAULT }
postgres=# ALTER DATABASE nombre SET argumento FROM CURRENT
postgres=# ALTER DATABASE nombre RESET argumento
postgres=# ALTER DATABASE nombre RESET ALL
```

Todas las variables de configuración no son modificables por este sistema. Por ejemplo, es habitual definir el argumento `search_path` para la base de datos para adaptarse a los esquemas ya creados en ella, como se muestra en el siguiente ejemplo:

```
postgres=# ALTER DATABASE nombre SET search_path = public, test, $user;
```

Este argumento `search_path` toma este valor para todas las conexiones que se abran más adelante.

2. Eliminación de una base de datos

La sentencia `DROP DATABASE` permite eliminar una base de datos. Solo es posible eliminar una base de datos si no hay ningún usuario conectado a la base, incluido el usuario que lanza el comando. Todos los objetos contenidos en la base de datos se eliminan, incluidos los directorios y archivos correspondientes en el sistema de archivos. La sinopsis del comando es la siguiente:

```
postgres=# DROP DATABASE [ IF EXISTS ] nombrebase
```

La opción `IF EXISTS` evita que se provoque un error durante la eliminación si la base de datos no existe.

Esta eliminación es definitiva. No es posible volver atrás una vez que se ejecuta el comando.

Los esquemas

Los esquemas, también llamados espacios de nombres, existen implícitamente en todas las bases de datos. Se trata de una noción puramente lógica, que permite agrupar por nombre los objetos como las tablas, las vistas o las funciones, que definen de esta manera una ruta de acceso.

De esta manera, en una misma base de datos hay varios objetos que pueden tener el mismo nombre, pero ser físicamente distintos. El nombre del esquema utiliza como prefijo el nombre del objeto. En consecuencia, la distinción entre los objetos es clara.

Se puede hacer una analogía con la noción de paquete o *namespace* en diferentes lenguajes de programación. Objetos con nombres idénticos (atributos, métodos...) pero con una implementación diferente se pueden distinguir por el nombre del paquete. En PostgreSQL, la noción es idéntica, considerando el hecho de que una clase se corresponde con una tabla y la implementación, con los datos. Por lo tanto, el paquete se corresponde con un esquema. A pesar de esta analogía, observe una diferencia importante: solo existe un nivel de profundidad con los esquemas. En PostgreSQL, no es posible crear esquemas en un esquema.

Inicialmente existen varios esquemas en una base de datos:

- El esquema `pg_catalog` contiene las tablas y vistas de sistema, que permiten a PostgreSQL almacenar la información relativa a su funcionamiento.
- El esquema `public` es un esquema inicialmente creado con la base de datos y considerado como esquema por defecto.
- El esquema `information_schema` contiene la información de los objetos de la base de datos actual.

La creación de un esquema se realiza con la sentencia `CREATE SCHEMA`, como se muestra en la siguiente sinopsis:

```
postgres=# CREATE SCHEMA [ nombresquema | AUTHORIZATION nombrerol ]
[ schema_element [ ... ] ];
postgres=# CREATE SCHEMA IF NOT EXISTS [ nombresquema | AUTHORIZATION
nombrerol ];
```

Es posible indicar el nombre del esquema explícitamente o deduciéndolo del nombre del rol de la sentencia `AUTHORIZATION`. Cuando el nombre del esquema es explícito, el propietario del esquema se puede indicar explícitamente por la sentencia `AUTHORIZATION`.

El siguiente ejemplo crea el esquema `formacion` con dos métodos diferentes:

- Llamando explícitamente el esquema:

```
postgres=# CREATE SCHEMA formacion;
```

- O utilizando el rol `formacion` existente:

```
postgres=# CREATE SCHEMA AUTHORIZATION formacion;
```

Es posible crear objetos simultáneamente en el esquema, añadiendo las sentencias de creación en el comando para crear al mismo tiempo una tabla, una vista o una función:

```
postgres=# CREATE SCHEMA formacion CREATE TABLE sesiones (id int,
inicio date);
```

Una vez que se crea el esquema, es posible acceder al contenido (tablas, vistas, etc.) utilizando el nombre cualificado del objeto, es decir, utilizando como prefijo el nombre del esquema, como en el siguiente ejemplo:

```
user=> SELECT * FROM formacion.sesiones;
```

Cuando el usuario no cualifica el nombre de la tabla, el intérprete de consultas utiliza el contenido de la variable `search_path` para buscar los objetos. Por lo tanto, modificando el contenido de esta variable es posible indicar cuál debe ser el esquema implícito para buscar en él los objetos.

El siguiente comando muestra el contenido por defecto de la variable:

```
user=> show search_path;
search_path
————————
$user,public
```

Por lo tanto, el intérprete de consultas busca en un esquema que tiene el mismo nombre que el usuario y después, en el esquema `public`.

Es posible modificar el contenido de esta variable con el comando `set`:

```
user=> set search_path = formacion, public, $user;
```

El intérprete de consultas busca los objetos en el primer esquema, después pasa al siguiente si no se han encontrado. Por lo tanto, la sentencia de los nombres de los esquemas en la variable tiene su importancia.

La sentencia `IF NOT EXISTS` permite no generar ningún error si el esquema ya existe. La utilización de esta sentencia no permite crear objetos simultáneamente, de tal manera que el comando solo genere un mensaje `NOTICE`.

1. Modificación de un esquema

Es posible modificar el nombre o el propietario de un esquema utilizando la sentencia `ALTER SCHEMA`, como se muestra en la siguiente sinopsis:

```
postgres=# ALTER SCHEMA formacion RENAME TO nuevonombre;
postgres=# ALTER SCHEMA formacion OWNER TO nuevopropietario;
```

Estas modificaciones no implican ningún cambio en los ajustes como la variable `search_path` o una consulta SQL que utilice un nombre cualificado, de tal manera que se pueden producir errores como consecuencia de un renombramiento. Por lo tanto, conviene ser prudente utilizando estos comandos.

2. Eliminación de un esquema

Si se utiliza la opción `CASCADE`, la eliminación de un esquema implica la eliminación de todos los objetos contenidos (tablas, vistas, etc.). Por defecto, la opción `RESTRICT` impide la eliminación del esquema si siempre contiene objetos. Solo el propietario o un superusuario pueden hacer esta eliminación. La sinopsis de este comando es la siguiente:

```
postgres=# DROP SCHEMA [ IF EXISTS ] nombresquema [, ...]
[ CASCADE | RESTRICT ];
```

La opción `IF EXISTS` evita que se provoque un error durante la eliminación si el esquema no existe.

Las tablas

Una tabla es el elemento básico de un servidor de bases de datos. En ella es donde se almacenan los datos, y la organización de las tablas determina la calidad de la base de datos.

La elección de los tipos de datos y los enlaces entre las tablas forman parte de la etapa de diseño de la base de datos para que tenga buen rendimiento y sea duradera.

La sentencia `CREATE TABLE` permite poner en marcha las elecciones de este diseño.

La sinopsis mínima de este comando es la siguiente:

```
CREATE [ TEMP | UNLOGGED ] TABLE [ IF NOT EXISTS ] nombretabla ( [
 { nombrecol tipo [ COLLATE collation ] [ restriccioncolumna[...] ]
   | restriccionтabla
   | LIKE source_table [ like_option ... ]
 } [,...] ] )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] )
 | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
```

Previo a la definición de los atributos de una tabla, se puede indicar un determinado número de características adicionales para modificar el comportamiento de una tabla en la base de datos.

En primer lugar, los modificadores `TEMP` y `UNLOGGED` permiten cambiar la naturaleza de una tabla:

- Una tabla temporal creada con el modificador `TEMP` solo es visible en el contexto de la sesión actual, es decir, ninguna otra sesión concurrente tiene acceso a esta tabla y esta deja de existir cuando termina la sesión. La sentencia `ON COMMIT` modifica la visibilidad de la tabla temporal a nivel de la transacción. Por defecto, los registros se conservan hasta el final de la sesión (`PRESERVE ROWS`), pero es posible eliminarlos (`DELETE ROWS`) o eliminar la tabla temporal al final de la transacción (`DROP`). Los índices asociados a esta tabla también son temporales y, por lo tanto, se elimina en el mismo momento que la tabla. Una tabla temporal puede tener el mismo nombre que una tabla permanente. En este caso, será la única visible mientras dure la sesión.
- Una tabla creada con el modificador `UNLOGGED` no ve sus datos guardados en los archivos de traza de transacciones. Este mecanismo que implementa la durabilidad de datos puede provocar que haya datos que desaparezcan en casos de parada accidental de la instancia. En contrapartida, la inserción de estos datos puede ser más rápida. A partir de la versión 9.5 de PostgreSQL, es posible cambiar este estado utilizando la sentencia `ALTER TABLE`, con las clausulas `SET LOGGED` o `SET UNLOGGED`.

Las opciones `WITH OIDS` y `WITHOUT OIDS` permiten respectivamente agregar a la tabla una columna que contiene identificadores de registros o no. Si estos argumentos no están presentes, el valor por defecto depende del argumento `default_with_oids` de la configuración, que por defecto desactiva los `OIDS`. Generalmente se desaconseja utilizar los OIDS y en particular depender de ellos en una aplicación porque, cuando los datos se utilizan en otra instancia PostgreSQL, por ejemplo como consecuencia de una migración o la utilización de la replicación, el valor del OID puede estar utilizado en otra instancia PostgreSQL. Un valor OID es único en una instancia PostgreSQL dada.

La opción `TABLESPACE` permite indicar en qué espacio de tablas se debe crear la tabla.

Algunos argumentos de configuración se pueden indicar por la sentencia `WITH`. Estos argumentos tienen un valor global que viene del archivo de configuración de la instancia. En función de lo que es deseable para cada tabla, se pueden indicar los siguientes argumentos:

- El argumento `fillfactor`, expresado en porcentaje, permite modificar el comportamiento del servidor durante el almacenamiento físico de los datos. Por defecto, el es 100 %, lo que se corresponde con la utilización de la totalidad de una página de datos en el sistema de almacenamientos para los datos de la tabla. Cuando un valor más bajo se indica, el servidor no utilizará toda la página, lo que permitirá a los comandos de actualización usar este espacio que se ha dejado libre. Por lo tanto, esta opción permite optimizar las escrituras en el sistema de almacenamiento en caso de una tabla que se actualiza regularmente.
- Los argumentos `autovacuum_*` definen el comportamiento del agente de mantenimiento y se detallan en el capítulo Explotación.

1. Atributos

a. Definición de un atributo

Una tabla se puede definir como un conjunto ordenado de atributos y tuplas. Un atributo se puede definir como la asociación entre un nombre y un tipo de datos.

Para cada atributo es posible añadir un valor por defecto y restricciones. El valor por defecto se debe corresponder con el tipo de datos utilizado y se indica con la palabra clave `DEFAULT`. Sin indicación particular, el valor por defecto es `NULL`.

Cada tabla puede tener un máximo de 1 600 atributos.

b. Restricciones

Las restricciones sobre las columnas permiten declarar una columna como clave primaria o clave extranjera, e incluso forzar una columna a tener un valor no nulo o único.

Asignar un nombre a una restricción permite encontrarla fácilmente, en particular cuando se trata de manipularla, como por ejemplo durante la eliminación.

En el caso de una restricción de unicidad o clave primaria, se crea automáticamente un índice. Es posible seleccionar el espacio de tablas donde crear el índice con la opción `USING INDEX TABLESPACE`, así como los argumentos de almacenamiento del índice, por ejemplo `fillfactor`.

La sentencia de verificación `CHECK` espera una expresión booleana antes de una inserción o actualización. Si el resultado es `TRUE` o `UNKNOWN`, entonces se aceptan los datos; en caso contrario, el valor `FALSE` implica el rechazo de la inserción o la actualización.

En el caso de una clave extranjera, es posible señalar con la sentencia `REFERENCES` la indicación de la tabla y la columna referenciada, así como la precisión de la concordancia de los valores: por defecto, cuando se inserta un valor nulo se acepta, independientemente del estado de las columnas referenciadas. Es el comportamiento debido a la opción `MATCH SIMPLE`. Con la opción `MATCH FULL`, es posible prohibir la utilización de un valor nulo, salvo si existe en las columnas referenciadas.

Siempre con las claves extranjeras, en función de las modificaciones de los valores en las columnas referenciadas, se pueden realizar distintas acciones sobre dos eventos diferentes: las actualizaciones (`ON UPDATE`) y las eliminaciones (`ON DELETE`):

- `NO ACTION`, `RESTRICT`: se genera un error señalando que el dato se está utilizando. En el caso de `NO ACTION`, si la restricción es diferida, entonces el error solo se producirá durante la verificación.
- `CASCADE`: el dato se modifica según el cambio realizado en la columna referenciada.

- SET NULL: el valor de la columna vale NULL.
- SET DEFAULT: a la columna se le asigna el valor por defecto.

Para terminar, el último argumento permite diferir la verificación de las claves extranjeras al final de una transacción con la opción DEFERRABLE, para evitar bloquear su desarrollo. Por defecto, las verificaciones no son diferidas (NOT DEFERRABLE).

Cuando las verificaciones sobre las claves se declaran como diferidas, las opciones INITIALLY DEFERRED e INITIALLY IMMEDIATE permiten indicar el momento en que se hace la verificación: la opción INITIALLY DEFERRED retrasa la verificación hasta el final de la transacción, mientras que la opción INITIALLY IMMEDIATE desencadena la verificación al final de la instrucción. El interés es poder modificar el momento de la verificación de una restricción diferida con la sentencia SET CONSTRAINT.

La siguiente sinopsis muestra las diferentes posibilidades de una restricción sobre una columna:

```
[ CONSTRAINT nombre_restricción ]
{ NOT NULL | NULL |
 CHECK ( expresión ) [ NO INHERIT ] |
 DEFAULT expr |
 UNIQUE [ WITH ( storage_parameter [= value] [, ... ] ) ]
           [ USING INDEX TABLESPACE espaciotabla ] |
 PRIMARY KEY [ WITH (argumento [= valor] [, ... ] ) ]
           [ USING INDEX TABLESPACE espaciotabla ] |
 REFERENCES tabla [ ( columna ) ]
       [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
       [ ON DELETE acción ] [ ON UPDATE acción ] }
[ DEFERRABLE | NOT DEFERRABLE ]
[ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

2. Restricciones de tablas

También es posible declarar las restricciones válidas para toda la tabla y no únicamente para un atributo. Esto es útil cuando las restricciones utilizan varias columnas, como en la siguiente sinopsis:

```
[ CONSTRAINT nombre_restricción ]
{
   UNIQUE ( nombre_columna [, ... ] ) [ USING INDEX
TABLESPACE espaciológico ] |
   PRIMARY KEY ( nombre_columna [, ... ] ) [ USING INDEX
TABLESPACE espaciológico ] |
   CHECK ( expresión ) |

   FOREIGN KEY ( nombre_columna [, ... ] ) REFERENCES
table_reference [ ( columna_referencia [, ... ] ) ]
       [ MATCH FULL  | MATCH SIMPLE ]
       [ ON DELETE acción ] [ ON UPDATE acción ]
}
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED |
INITIALLY IMMEDIATE ]
```

3. Ejemplos

Los siguientes ejemplos muestran la creación de diferentes tablas utilizadas para una aplicación minimalista de administración de facturas y prestaciones. El objetivo es poder reservar las prestaciones con antelación y formalizar una factura cuando se realicen.

La base de datos se divide en cinco tablas:

- Una tabla `clientes`, que contiene las referencias de los clientes.
- Una tabla `contactos`, que contiene los diferentes contactos de los clientes.
- Una tabla `prestaciones`, que contiene todas las prestaciones reservadas, confirmadas y terminadas.
- Una tabla `facturas`, que contiene las fechas y los números de las facturas.
- Una tabla `registros_facturas`, que relaciona las prestaciones con las facturas.

La tabla `clientes` es muy sencilla. Solo contiene dos campos, uno es la clave primaria a la que PostgreSQL añade automáticamente un índice.

El campo `cl_direccion` es simplemente un campo de tipo de datos `text`, que permite almacenar de gran cantidad de texto:

```
CREATE TABLE clientes (
   cl_nombre varchar(20) PRIMARY KEY ,
   cl_direccion text
);
```

La tabla `contactos` permite almacenar la información de un contacto, asociándolo con un cliente gracias a la palabra clave `references`. Esta definición solo utiliza el nombre de la tabla que sirve de referencia. PostgreSQL elegirá implícitamente el campo que tiene el mismo nombre (`cl_nombre`) en la tabla referenciada.

```
CREATE TABLE contactos (
   ct_nombre varchar(20) ,
   ct_apellidos varchar(20),
   ct_telefono varchar(20),
   ct_email varchar(35),
   ct_funcion varchar(30),
   cl_nombre varchar(20) REFERENCES clientes
);
```

La tabla `prestaciones` contiene diferentes campos que permiten guardar la reserva de una prestación, con una fecha de inicio y una fecha de fin, y después confirmarla.

```
CREATE TABLE prestaciones (
   prest_id integer PRIMARY KEY ,
   prest_nombre VARCHAR(20),
   prest_desc TEXT ,
   prest_tipo VARCHAR(30),
   prest_fecha_inicio TIMESTAMP WITH TIME ZONE ,
   prest_fecha_fin TIMESTAMP WITH TIME ZONE ,
   prest_confirm bool,
    cl_nombre varchar(20) REFERENCES clientes
);
```

La tabla `facturas` permite guardar una factura, con una fecha de facturación, una fecha de pago y una referencia a un cliente:

```
CREATE table facturas (
   fct_num varchar(30) PRIMARY KEY ,
   fct_fecha TIMESTAMP WITH TIME ZONE ,
   fct_fecha_pago TIMESTAMP WITH TIME ZONE ,
   fct_medio_pago varchar(20),
   cl_nombre varchar(20) REFERENCES clientes
);
```

La tabla `registros_facturas` hace referencia al mismo tiempo a la tabla `prestaciones` y a la tabla `facturas`, integrando la cantidad de la facturación de una prestación para calcular la cantidad total de una factura.

```
CREATE TABLE registros_facturas (
   fct_num varchar(30) REFERENCES facturas ( fct_num ) ,
   prest_id integer UNIQUE REFERENCES prestaciones,
   registros_fct_nombre varchar(20),
   registros_fct_cantidad decimal(10,3),
   registros_fct_cnt float
);
```

Una vez que se escriben estas sentencias SQL en un archivo, por ejemplo clientes-ddl.sql, es posible ejecutar simplemente este archivo con el comando `\i` de `psql`:

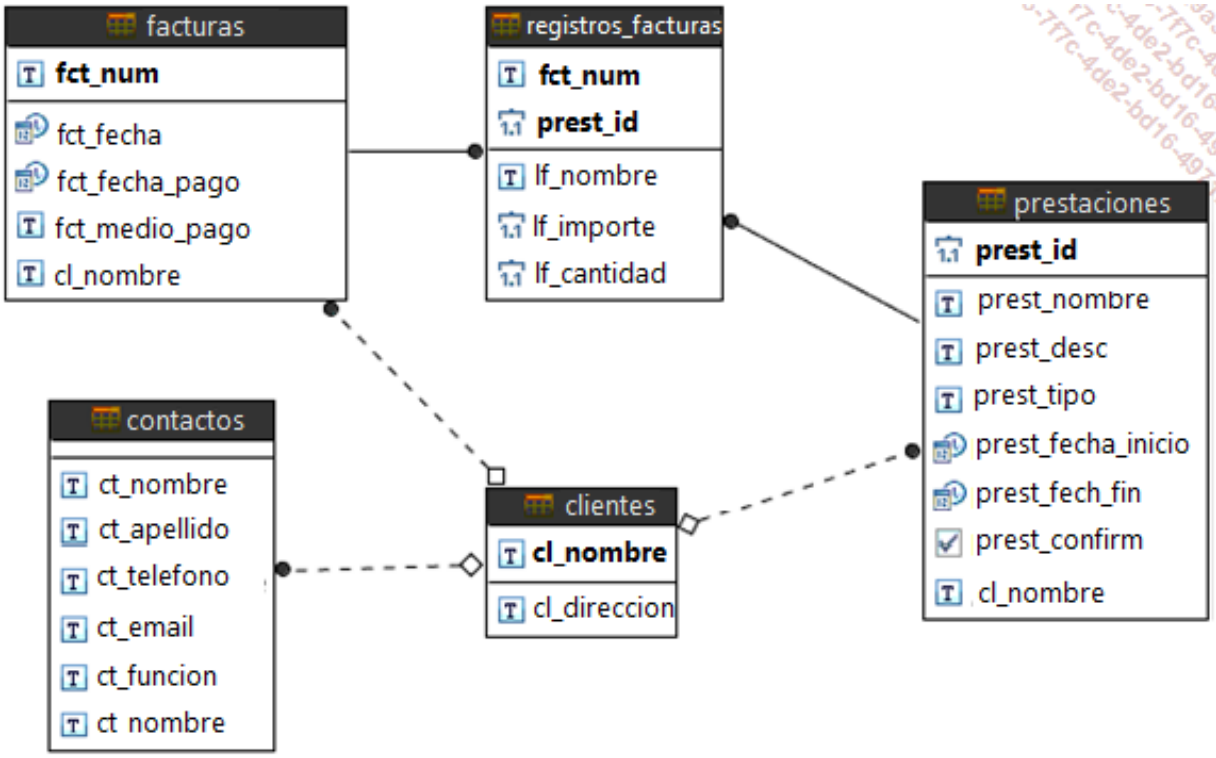
```
postgres=# \i clientes-ddl.sql
```

o utilizando la opción `-f` de `psql`:

```
[postgres] $ psql -f clientes-ddl.sql
```

También es posible ejecutar estas sentencias SQL desde cualquier editor de código SQL conectado a la instancia PostgreSQL.

El siguiente diagrama muestra el esquema de los datos tal y como se puede visualizar con el editor DBeaver, una vez que las tablas están creadas:



4. Modificación de una tabla

Sobre una tabla es posible realizar numerosas acciones, principalmente el control de la tabla en la base de datos: su nombre, su esquema y su ubicación física.

En todos los casos, la sentencia `IF EXISTS` justo después de `ALTER TABLE` evita que se provoque un error durante la ejecución. La sentencia `ONLY` permite no propagar las modificaciones a las tablas heredadas.

La modificación de la ubicación física de la tabla se realiza modificando el espacio de tablas en el que se sitúa. Por supuesto, el nuevo espacio de tablas debe existir antes de ejecutar este comando:

```
postgres=# ALTER TABLE nombretabla SET TABLESPACE espaciotabla;
```

Los siguientes comandos permiten modificar:

- El espacio de nombres desde el que es accesible la tabla:

```
postgres=# ALTER TABLE nombretabla SET SCHEMA esquema;
```

- El propietario de la tabla:

```
postgres=# ALTER TABLE nombretabla OWNER TO rol;
```

- El nombre de la tabla:

```
postgres=# ALTER TABLE nombretabla RENAME TO nvnombretabla;
```

Los siguientes comandos permiten renombrar un atributo o una columna:

```
postgres=# ALTER TABLE name RENAME nombrecolumna TO nv_nombre;
postgres=# ALTER TABLE name RENAME CONSTRAINT nombre_restricción
TO nv_nombre;
```

Los otros comandos permiten realizar modificaciones sobre el contenido, por ejemplo la adición, la eliminación o el cambio de nombre de columnas:

```
postgres=# ALTER TABLE nombretabla ADD COLUMN IF NOT EXISTS columna tipo
[ restricción_columna [ ... ] ]
postgres=# ALTER TABLE nombretabla DROP columna [ RESTRICT | CASCADE ]
```

Desde la versión 9.6 de PostgreSQL, es posible utilizar la sentencia `IF NOT EXISTS` durante la adición de una columna.

Es posible modificar el tipo de una columna y su valor por defecto:

```
postgres=# ALTER TABLE nombretabla ALTER columna TYPE tipo
postgres=# ALTER TABLE nombretabla ALTER columna SET DEFAULT expresion
postgres=# ALTER TABLE nombretabla ALTER columna DROP DEFAULT
```

Algunos comandos también permiten la eliminación o adición de restricciones. Cuando se asigna un nombre a una restricción automáticamente, es necesario encontrar el nombre generado por el servidor para eliminarla:

```
postgres=# ALTER TABLE nombretabla ADD restricción_tabla
postgres=# ALTER TABLE nombretabla DROP CONSTRAINT nombre_restricción
[ RESTRICT | CASCADE ]
postgres=# ALTER TABLE nombretabla ALTER columna { SET | DROP } NOT NULL
```

Las diferentes modificaciones mencionadas se pueden acumular para, por ejemplo, añadir una columna y una restricción en la misma operación.

Algunas modificaciones añadidas necesitan una reescritura física de la tabla, por ejemplo durante un cambio de tipo de datos o durante la adición de una columna con un valor por defecto. Esta reescritura puede plantear problemas, porque la sentencia `ALTER TABLE` debe obtener un bloqueo exclusivo sobre la tabla durante la duración de toda la operación. Durante este periodo, ninguna otra consulta concurrente puede utilizar la tabla. Por lo tanto, cuando el número de registros es importante, la reescritura puede ser muy larga.

Para evitar bloquear durante mucho tiempo una tabla, algunas veces es necesario separar una consulta `ALTER TABLE` en varias consultas.

Por ejemplo, la adición de una columna que tiene como restricción un valor por defecto se puede declarar de la siguiente manera:

```
ALTER TABLE nombretabla add column pruebas int default 0;
```

Esta consulta funciona correctamente, pero solo libera el bloqueo exclusivo cuando termina la reescritura de la tabla. El siguiente procedimiento consigue el mismo resultado obteniendo en cada sentencia `ALTER TABLE` un bloqueo exclusivo sobre la tabla, pero la reescritura provocada por la sentencia `UPDATE`, siendo la más larga, no impone bloqueo exclusivo sobre la tabla, sino solamente sobre cada registro de la tabla:

```
ALTER TABLE nombretabla add column pruebas;
update nombretabla set pruebas = 0;
ALTER TABLE nombretabla alter column pruebas set default = 0;
```

Sin embargo, la tabla reescrita ocupa en el sistema de almacenamiento dos veces su volumen útil. Por lo tanto, es necesario realizar un mantenimiento sobre esta tabla usando el comando `VACUUM`. Este comando se detalla en el capítulo Explotación:

```
VACUUM nombretabla;
```

El siguiente comando permite pasar de un tipo de datos numéricos `integer` sobre 32 bits al tipo de datos `bigint` sobre 64 bits:

```
ALTER TABLE nombretabla alter column  id set tipo bigint;
```

Los valores no cambian, pero el almacenamiento es diferente. Por lo tanto, es necesaria una reescritura de la tabla.

El siguiente procedimiento permite una reescritura de la tabla minimizando los bloqueos exclusivos:

```
ALTER TABLE nombretabla add column id_alt bingint;
UPDATE nombretabla set id_alt = id;
ALTER TABLE nombretabla drop column id;
ALTER TABLE nombretabla rename id_alt to id;
```

Como en el ejemplo anterior, es necesaria una operación de mantenimiento. Además, hay que observar un cambio en la sentencia de las columnas, ya que la nueva columna creada se encuentra necesariamente al final de la lista.

Los siguientes comandos permiten modificar las propiedades de una tabla, el valor de un argumento de configuración para esta tabla...:

```
postgres=# ALTER TABLE nombretabla DISABLE TRIGGER [ trigger_name | ALL | USER ]
postgres=# ALTER TABLE nombretabla ENABLE TRIGGER [ trigger_name | ALL | USER ]
postgres=# ALTER TABLE nombretabla ENABLE REPLICA TRIGGER trigger_name
postgres=# ALTER TABLE nombretabla ENABLE ALWAYS TRIGGER trigger_name
```

```
postgres=# ALTER TABLE nombretabla DISABLE RULE rewrite_rule_name
postgres=# ALTER TABLE nombretabla ENABLE RULE rewrite_rule_name
postgres=# ALTER TABLE nombretabla ENABLE REPLICA RULE rewrite_rule_name
postgres=# ALTER TABLE nombretabla ENABLE ALWAYS RULE rewrite_rule_name
```

```
postgres=# ALTER TABLE nombretabla CLUSTER ON index_name
postgres=# ALTER TABLE nombretabla SET WITHOUT CLUSTER
```

```
postgres=# ALTER TABLE nombretabla SET WITH OIDS
postgres=# ALTER TABLE nombretabla SET WITHOUT OIDS
```

```
postgres=# ALTER TABLE nombretabla SET ( storage_parameter = value [, ... ] )
postgres=# ALTER TABLE nombretabla RESET ( storage_parameter [, ... ] )
```

```
postgres=# ALTER TABLE nombretabla INHERIT parent_table
postgres=# ALTER TABLE nombretabla NO INHERIT parent_table
```

El siguiente comando permite modificar el factor de relleno de una tabla:

```
postgres=# ALTER TABLE nombretabla SET FILLFACTOR = valor
```

Algunos de estos comandos implican cambios demasiado profundos en los archivos utilizados. Por ejemplo, la adición de una columna con un valor no nulo por defecto implica la reconstrucción completa de la tabla, que necesita el doble del volumen de esta, con un tiempo de ejecución no despreciable.

5. Eliminación de una tabla

La eliminación se realiza simplemente utilizando la sentencia `DROP TABLE`. Después de haber determinado la tabla que se ha de eliminar, la única opción, `CASCADE`, consiste en indicar si los objetos dependientes de esta tabla, como una vista por ejemplo, también se deben eliminar:

```
postgres=# DROP  [ IF EXISTS ] TABLE nombre [, ...]
[ CASCADE | RESTRICT ]
```

La opción `IF EXISTS` evita que se provoque un error durante la eliminación si la tabla no existe.

El comportamiento por defecto (`RESTRICT`) no elimina la tabla si hay otros objetos que dependen de ella.

Es posible eliminar varias tablas separando los nombres por una coma.

6. Creación de una tabla desde una consulta

Existen dos mecanismos para crear una tabla según el resultado de una consulta `SELECT`:

- `CREATE TABLE AS`
- `SELECT INTO`

Estos dos mecanismos producen un resultado equivalente. La diferencia es que existe una sintaxis diferente e incompatible de `SELECT INTO` en el lenguaje de procedimiento almacenado Pl/PgSQL. Por lo tanto, la sentencia `CREATE TABLE AS` es una elección con permisos.

La sentencia `CREATE TABLE AS` deriva de la sentencia `CREATE TABLE`, añadiendo una consulta `SELECT`, que recupera un conjunto de tuplas (es decir, una relación) insertadas como contenido durante la creación de la tabla. Este comportamiento puede resultar parecido a una vista, pero, en realidad, en este caso los datos se copian. Por lo tanto, las modificaciones realizadas en las tablas originales no se repercuten en la nueva tabla.

La sinopsis de la sentencia `CREATE TABLE AS` es la siguiente:

```
CREATE  [ TEMP ] TABLE nombretabla [ (nombre_columna [, ...] ) ]
AS SELECT [ columnas ] FROM [ nombretabla , ... ]
```

El siguiente ejemplo muestra la creación de una tabla `facturacion_t` utilizando una consulta `SELECT`:

```
CREATE TABLE facturacion_t AS
SELECT f.fct_num, f.fct_fecha, f.fct_fecha_pago,
 f.cl_nombre,
 sum(l.registros_fct_importe*l.registros_fct_cnt) AS importe_BRT,
 sum(l.registros_fct_importe*l.registros_fct_cnt) AS importe_BRT,
 sum(l.registros_fct_importe*l.registros_fct_cnt)*1.21 AS importe_TOTAL
 FROM facturas f NATURAL JOIN registros_facturas l LEFT
JOIN prestaciones p on l.prest_id=p.prest_id group BY
f.fct_num, f.fct_fecha, f.fct_fecha_pago, f.cl_nombre
 ORDER BY substring(fct_num from '..$') ASC;
```

La sinopsis de la sentencia `SELECT INTO` es muy parecida a la de la sentencia `SELECT`. Es la siguiente:

```
SELECT ( * | columnas ) INTO [ TEMP ]  nuevatabla
FROM nombretabla [ , nombretabla ]
```

El siguiente ejemplo produce el mismo resultado que la sentencia `CREATE TABLE AS`:

```
SELECT f.fct_num, f.fct_fecha, f.fct_fecha_pago,f.cl_nombre,
 sum(l.registros_fct_importe*l.registros_fct_cnt) AS importe_BRT,
 sum(l.registros_fct_importe*l.registros_fct_cnt)*1.21 AS importe_TOTAL
 INTO facturacion_t
 FROM facturas f NATURAL JOIN registros_facturas
l LEFT JOIN prestaciones p on l.prest_id=p.prest_id
 GROUP BY f.fct_num, f.fct_fecha, f.fct_fecha_pago,
f.cl_nombre
 ORDER BY substring(fct_num from '..$') ASC;
```

7. Partición declarativa

La partición de los datos se corresponde con la separación física de los datos en objetos que permiten manipularlos, en función de una clave de partición definida.

La partición de los datos es una respuesta al aumento del volumen de datos en las tablas. Cuando este volumen sobrepasa un tamaño crítico, por ejemplo varios centenares de millones, incluso mil millones de registros en una tabla, es muy costoso en tiempo y en recursos materiales recorrer estos datos, en particular cuando solo es útil una parte significativamente reducida de ellos.

Por ejemplo, el interés de un dato cuya característica es el tiempo (con un atributo de tipo `timestamp`) disminuirá con el tiempo: una traza de actividad de un servidor HTTP, una transacción bancaria o un pequeño anuncio de hace varios años no tiene valor en la actualidad. El valor añadido ya se ha extraído: estadísticas de un sitio web, saldo de una cuenta u objeto vendido. Por lo tanto, se puede archivar.

Por el contrario, los datos actuales tienen un interés directo. Se pueden leer en la actualidad. Por lo tanto, en la misma tabla hay datos archivables y otros útiles. En estos dos casos, cualquier acción va a tener un coste cada vez más importante: un `DELETE` en una tabla de tamaño importante puede ser largo y necesitar operaciones sobre el soporte del almacenamiento que sobrepasan en mucho el tamaño de los datos que se han de archivar. La misma lógica se aplica a un `SELECT`.

Este tipo de problema hace que el administrador particione los datos. Desde la versión 10, PostgreSQL propone un mecanismo de partición integrado. En lo que respecta a las versiones anteriores, siempre están disponibles las extensiones de PostgreSQL que facilitan la partición basada en herencia: la más avanzada es `pg_partman`: http://pgxn.org/dist/pg_partman/doc/pg_partman.html y el autor de este libro ha escrito la extensión partmgr: https://github.com/slardiere/PartMgr.

La declaración de la partición se basa en la sentencia `CREATE TABLE`, que permite crear la tabla principal y las particiones. La creación de la tabla principal utiliza la palabra clave `PARTITION BY`, seguida de la opción `RANGE` o `LIST`, según el tipo de partición elegida. Una partición de tipo `RANGE` implica que los datos se reparten en las particiones según rangos de valores, mientras que una partición de tipo `LIST` implica que los datos se reparten según las listas de valores. La clave de partición es un atributo que existe en la tabla o una expresión sobre un atributo. En el caso de la partición de tipo `RANGE`, es posible asociar hasta 32 atributos o expresiones, mientras que una partición de tipo `LIST` solo permite utilizar un único atributo o expresión. La tabla principal no puede definir una clave primaria o única, incluso si es posible definir tales restricciones en las particiones.

A partir de esta clave de partición, durante la inserción de datos, PostgreSQL puede determinar la partición en la que insertar de manera efectiva el dato. Es necesario que la partición exista efectivamente, sin que se produzca un error. En efecto, no existe tabla por defecto en la que los datos «huérfanos» se puedan insertar. La creación de las particiones se hace utilizando la sentencia `CREATE TABLE` con la palabra clave `PARTITION OF`. Con esta palabra clave, se hace referencia a la tabla principal y al rango de valores o lista de valores que la partición puede aceptar, según el tipo de partición.

El siguiente ejemplo utiliza la tabla `prestaciones`. La clave de partición elegida es el tiempo:

```
CREATE TABLE prestaciones (
   prest_id integer,
   prest_nombre VARCHAR(20),
   prest_desc TEXT ,
   prest_tipo VARCHAR(30),
   prest_fecha_inicio TIMESTAMP WITH TIME ZONE ,
   prest_fecha_fin TIMESTAMP WITH TIME ZONE ,
   prest_confirm bool,
    cl_nombre varchar(20) REFERENCES clientes
) PARTITION BY RANGE ( prest_fecha_inicio );
```

Después, elegimos crear las particiones por año:

```
CREATE TABLE prestaciones_2016
PARTITION OF prestaciones
FOR VALUES FROM ('2016-01-01') TO ('2017-01-01');

CREATE TABLE prestaciones_2017
PARTITION OF prestaciones
FOR VALUES FROM ('2017-01-01') TO ('2018-01-01');

CREATE TABLE prestaciones_2018
PARTITION OF prestaciones
FOR VALUES FROM ('2018-01-01') TO ('2019-01-01');
```

Las sentencias de creación de las tablas no retoman la lista de atributos; en efecto, las particiones tienen obligatoriamente los mismos atributos. Una vez que se crean las tablas, los datos se pueden insertar y estarán presentes automáticamente en la partición cuya fecha `prest_fecha_inicio` se corresponda con el rango de fechas utilizado. La tabla principal permanece siempre vacía, incluso si la partición correspondiente al dato que se debe insertar no existe: en ese caso, la inserción produce un error.

Es posible añadir varios niveles de partición, creando de esta manera subparticiones. En los siguientes ejemplos, las particiones por año están a su vez subdivididas utilizando una expresión sobre el atributo `prest_id`. Este reparto es útil para evitar que la partición por año no sea demasiado grande. La expresión utilizada, el módulo (`%`), permite garantizar un reparto equitativo de los datos en las tablas:

```
CREATE TABLE IF NOT EXISTS prestaciones_2017
   PARTITION OF prestaciones
     FOR VALUES FROM ('2017-01-01') TO (  '2018-01-01'  )
     PARTITION BY LIST ( ( prest_id % 4) );

CREATE TABLE IF NOT EXISTS prestaciones_2017_0
   PARTITION OF prestaciones_2017
     FOR VALUES IN ( 0 );

CREATE TABLE IF NOT EXISTS prestaciones_2017_1
   PARTITION OF prestaciones_2017
     FOR VALUES IN ( 1 );

CREATE TABLE IF NOT EXISTS prestaciones_2017_2
   PARTITION OF prestaciones_2017
     FOR VALUES IN ( 2 );

CREATE TABLE IF NOT EXISTS prestaciones_2017_3
   PARTITION OF prestaciones_2017
     FOR VALUES IN ( 3 );
```

Entonces, la tabla correspondiente a la partición anual se particiona y no contiene datos.

Cuando la tabla principal aparece en una sentencia `SELECT`, los valores indicados en los rangos de valores, como en las listas, se utilizan para determinar en qué particiones se encuentran los datos buscados. Entonces es interesante utilizar las claves de partición en los filtros de búsqueda de las consultas para recorrer solo las particiones efectivamente útiles, lo que constituye una optimización del rendimiento muy importante.

Cuando los datos de una partición no son útiles, por ejemplo cuando son demasido antiguos o ya se han tratado, es posible eliminarlos simplemente con un coste muy inferior al que supondría la sentencia `DELETE`.

Antes de eliminar la partición con la sentencia `DROP TABLE`, es necesario desconectar la partición con la sentencia `ALTER TABLE`:

```
ALTER TABLE prestaciones DETACH PARTITION prestaciones_2016;
DROP TABLE prestaciones_2016;
```

Las vistas

Una vista es el equivalente de una consulta `SELECT`, pero almacenada en forma de una relación equivalente a una tabla, aunque en modo solo lectura. Los datos mostrados por una vista no son modificables y la consulta `SELECT` subyacente se lanza en cada llamada a la vista. El interés de una vista es presentar los datos de una determinada manera y estar siempre disponible.

La sentencia `CREATE VIEW` permite crear una vista basándose en una consulta `SELECT`, como en la siguiente sinopsis:

```
CREATE [ OR REPLACE ] [ TEMP ] VIEW nombre [ ( columna, ... ) ]
 WITH ( nombreopción = valor] [, ... ] )
 AS consulta
 WITH [ CASCADED | LOCAL ] CHECK OPTION;
```

Por defecto, el nombre de las columnas de la vista se corresponden con el nombre de las columnas devueltas por la consulta SELECT. Es posible modificar estos nombres de columna indicándolos entre paréntesis, después del nombre de la tabla.

Además, es posible sustituir una vista existente, añadiendo las palabras clave OR REPLACE después de CREATE, con la condición de que las columnas de la vista sean identicas en número y tipo.

La sentencia TEMP modifica el alcance de la vista, haciéndola accesible únicamente durante la sesión actual y solo durante el tiempo de la sesión. Por lo tanto, la vista se elimina al final de la sesión. Cuando una vista temporal tiene el mismo nombre que un objeto existente, es prioritaria sobre este objeto durante la sesión.

Es posible actualizar los datos presentados por una vista con algunas condiciones concretas. La sentencia WITH CHECK OPTION permite verificar si las modificaciones añadidas a los datos se corresponden con las cláusulas de la vista, es decir, que el dato modificado formará parte de la vista. Los modificadores LOCAL y CASCADED permiten indicar el alcance de la vista. El valor por defecto CASCADED permite propagar la verificación a todas las vistas intermedias, hasta la tabla en la que se almacena el dato. El valor LOCAL solo hace la verificación sobre la vista actual. Estas modificaciones también se pueden indicar en la sentencia WITH, por ejemplo:

```
CREATE VIEW name WITH ( check_option = local ) ...
```

Para que los datos de una vista se puedan actualizar, es necesario respetar algunas condiciones:

- La vista solo debe tener una única tabla o vista listada en FROM.
- La vista no debe utilizar las palabras clave: WITH, DISTINCT, GROUP BY, HAVING, LIMIT u OFFSET al más alto nivel.
- La vista no opera sobre los conjuntos: UNION, INTERSECT o EXCEPT al más alto nivel.
- Las columnas de la vista no se deben calcular como resultado de cálculos agregados, funciones de ventana o que devuelvan un conjunto.

El siguiente ejemplo permite visualizar las cantidades de las facturas después de buscar los datos en las tablas facturas, registros_facturas y prestaciones. Por lo tanto, la vista tiene seis columnas, de las cuales dos son campos calculados, es decir, que los valores mostrados no se corresponden directamente con un dato de una tabla utilizada:

```
CREATE OR REPLACE VIEW facturacion AS
SELECT f.fct_num, f.fct_fecha, f.fct_fecha_pago, f.cl_nombre,
 sum(l.registros_fct_importe*l.registros_fct_cnt) AS importe_BRT,
 sum(l.registros_fct_importe*l.registros_fct_cnt)*1.196 AS importe_TOTAL
FROM facturas f NATURAL JOIN registros_facturas l
     LEFT JOIN prestaciones p ON l.prest_id=p.prest_id
GROUP BY f.fct_num, f.fct_fecha, f.fct_fecha_pago, f.cl_nombre
ORDER BY substring(fct_num from '..$') ASC;
```

Una vez creada la vista, es posible utilizarla con la sentencia SELECT como una tabla. En este ejemplo, no es posible actualizar los datos de la vista.

El siguiente ejemplo es una vista cuyos datos se pueden actualizar. El uso principal de esta funcionalidad es limitar los datos visibles y utilizables durante una actualización. Seleccionando correctamente los permisos asignados sobre la vista, es posible crear una partición limpia de los datos en función de un nombre de usuario:

```
CREATE VIEW prestaciones_2014
AS SELECT prest_id, prest_nombre, prest_desc, prest_tipo
  FROM prestaciones
 WHERE prest_fecha_inicio between '2014-01-01 00:00:00'
   AND '2014-12-31 23:59:59'
WITH LOCAL CHECK OPTION;
```

Por lo tanto, durante un `UPDATE` o un `INSERT` posterior sobre esta vista, los datos manipulados deben corresponder al año 2014 y podrán ser visibles en esta misma vista.

1. Modificación de una vista

Una vista se puede modificar para cambiar su nombre, su esquema o su propietario:

```
ALTER VIEW nombre OWNER TO new_owner
ALTER VIEW nombre RENAME TO new_name
ALTER VIEW nombre SET SCHEMA new_schema
```

En todos los casos, se puede añadir la sentencia `IF EXISTS` antes del nombre para no provocar un error cuando la vista no exista.

Se pueden definir valores por defecto para los atributos. Durante la inserción de los datos de la vista, se utiliza este valor por defecto incluso si este no está definido en la tabla donde realmente se almacena el valor:

```
ALTER VIEW nombre ALTER [ COLUMN ] column_name SET DEFAULT expresión
ALTER VIEW nombre ALTER [ COLUMN ] column_name DROP DEFAULT
```

Se pueden modificar las mismas opciones que las utilizadas para la creación de la vista:

```
ALTER VIEW nombre SET ( nombreopción [= valor] [, ... ] )
ALTER VIEW nombre RESET ( nombreopción [, ... ] )
```

2. Eliminación de una vista

La sentencia `DROP VIEW` permite eliminar una vista. Por supuesto, ningún dato se puede eliminar con este comando. Con la opción `CASCADE`, se pueden eliminar al mismo tiempo otras vistas dependientes. Este no es el caso por defecto. La siguiente sinopsis muestra la eliminación de una vista:

```
DROP VIEW [ IF EXISTS ] nombre [, ...] [ CASCADE | RESTRICT ]
```

La opción `IF EXISTS` evita que se provoque un error durante la eliminación si la vista no existe.

3. Vistas materializadas

Una vista materializada reúne las propiedades de una tabla y de una vista en un único objeto: los datos devueltos por la consulta se almacenan en una tabla, de tal manera que la consulta no se ejecute durante cada llamada de la vista.

Esto crea una especie de memoria RAM, que permite que los resultados se entreguen sin necesidad de ser recalculados.

a. Creación de una vista materializada

La sintaxis de creación de una vista materializada es la siguiente:

```
CREATE MATERIALIZED VIEW nombretabla [ (columna [, ...] ) ]
   [ WITH ( argumento [= valor] [, ... ] ) ]
   [ TABLESPACE espaciotabla ]
   AS consulta
   [ WITH [ NO ] DATA ];
```

Los argumentos de almacenamiento son los mismos que durante la creación de una tabla. Por defecto, la consulta se ejecuta durante la creación de la vista y los datos se almacenan en la tabla creada de esta manera. La opción `WITH NO DATA` permite no ejecutar la consulta; los datos solo se almacenan durante el siguiente refresco de la vista.

b. Actualización de los datos de una vista materializada

La actualización o refresco de los datos de una vista permite ejecutar de nuevo la consulta, almacenada en la definición de la vista, para disponer de datos coherentes con los datos de las tablas utilizadas en la consulta. La sintaxis de actualización es:

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] nombre
   [ WITH [ NO ] DATA ];
```

La sentencia `CONCURRENTLY` permite no bloquear los datos de la vista materializada durante la ejecución de la vista y, por lo tanto, durante la reubicación de los datos. Una actualización concurrente de los datos es más lenta y no se puede incluir en una transacción, pero esto permite continuar utilizando los antiguos datos de la vista materializada.

La utilización de la sentencia `WITH NO DATA` permite eliminar el conjunto de datos de la vista, haciéndola no utilizable.

El sistema de reglas

En PostgreSQL existe un sistema de reescritura de consultas, llamado reglas de reescritura, que permite modificar en profundidad la interpretación de las sentencias SQL. El comportamiento es muy cercano al de los triggers, pero, mientras que un trigger permite acciones antes o después de una consulta, una regla permite devolver una acción para realizarla en su lugar.

La sinopsis de la sentencia de creación de una regla es la siguiente:

```
CREATE [ OR REPLACE ] RULE nombre AS ON evento
   TO table [ WHERE condición ]
   DO [ ALSO | INSTEAD ] { NOTHING | comando |
( comando; comando ... ) }
```

Las palabras clave `OR REPLACE` permiten sustituir una regla existente, por lo tanto con el mismo nombre.

El evento debe ser una de las cuatro sentencias de manipulación de los datos (`SELECT`, `UPDATE`, `INSERT` y `DELETE`).

La condición permite filtrar la reescritura, en función de los valores del registro seleccionado, con la palabra clave `OLD` que designa al registro.

La palabra clave ALSO indica que se realiza la consulta que desencadena la reescritura, mientras que la palabra clave INSTEAD indica que esta consulta no se ejecutará más.

Para terminar, se indica la acción que se ha de realizar o de lo contrario se indica la palabra clave NOTHING. Es posible realizar varias acciones indicándolas entre paréntesis.

1. Eliminación de una regla

La eliminación de una regla se hace con la sentencia DROP RULE, indicando el nombre de la regla y de la tabla:

```
DROP RULE nombre ON table
```

2. Ejemplo

Por ejemplo, la tabla clientes contiene la lista de todos los clientes. Un cliente podría dejar de serlo y se debería eliminar de la tabla. Pero esto podría provocar problemas, principalmente durante la relectura de antiguas facturas. En lugar de la eliminación de un cliente, las reglas permiten modificar la tupla afectada.

Para esto, el siguiente comando permite añadir un campo activo que indica si un cliente está activo o no:

```
clientes=# ALTER TABLE clientes ADD COLUMN activo BOOL DEFAULT
true;
```

A continuación, una regla permite no eliminar un cliente, modificando simplemente el valor booleano del campo activo:

```
clientes=# CREATE RULE supr_clientes AS ON DELETE TO clientes
DO INSTEAD UPDATE clientes SET activo = false WHERE cl_nombre =
OLD.cl_nombre;
```

Cada vez que se llama la sentencia DELETE, la regla modifica esta llamada y ejecuta en su lugar una actualización, teniendo en cuenta el nombre del cliente modificado, gracias a la estructura OLD correspondiente al antiguo valor del registro. También existe una estructura NEW, correspondiente a los nuevos valores. La estructura OLD es válida para las sentencias DELETE y UPDATE, y la estructura NEW es válida para las sentencias UPDATE y INSERT.

La herencia

PostgreSQL propone un sistema de herencia para las tablas, lo que permite a una tabla hija heredar las columnas de la tabla madre y, además, tener sus propias columnas. Cuando se inserta una tupla en la tabla hija, los datos también son visibles desde la tabla madre. Solo se almacenan físicamente en esta tabla las columnas propias de la tabla hija.

Por ejemplo, la tabla prestaciones de la base de datos clientes permite almacenar información genérica de las prestaciones. Creando una tabla formaciones, que hereda de la tabla prestaciones, es posible especializar esta tabla, añadiendo campos, como en el siguiente ejemplo:

```
CREATE TABLE formaciones  (
  num_becarios int,
  plandeestudios varchar(25),
  tipo varchar(5) check ( tipo in ('intra','inter') ))
INHERITS (prestaciones);
```

La descripción de la tabla por el comando `\d` muestra que las columnas de la tabla madre forman efectivamente parte de esta nueva tabla:

```
clientes=# \d formaciones
                  Tabla "public.formaciones"
    Columna        |          Tipo             | Modificadores
-------------------+---------------------------+--------------
prest_id           | integer                   | not null
prest_nombre       | character varying(20)     |
prest_desc         | text                      |
prest_tipo         | character varying(30)     |
prest_fecha_inicio | timestamp with time zone  |
prest_fecha_fin    | timestamp with time zone  |
prest_confirm      | boolean                   |
cl_nombre          | character varying(20)     |
num_becarios       | integer                   |
plandeestudios     | character varying(25)     |
tipo               | character varying(5)      |
restricciones:
   "formaciones_tipo_check" CHECK ("type"::text =
'intra'::text OR "type"::text = 'inter'::text)
Hereda de: prestaciones
```

Durante la inserción de datos, los datos de las columnas heredadas están visibles en las dos tablas y, por lo tanto, son utilizables en ambas. De la misma manera, es posible crear otras tablas heredando de la tabla `prestaciones`, por ejemplo una tabla `consejo` que añadiría columnas específicas para este tipo de prestaciones.

Este mecanismo se utiliza para poner en marcha la partición de los datos.

Administración de datos externos

PostgreSQL dispone de mecanismos que permiten acceder a los datos situados en el exterior de una instancia. La implementación responde al estándar SQL/MED.

Esta implementación permite utilizar un componente externo que define los métodos útiles para conectarse a un servicio, como un servidor de base de datos, o abrir un recurso, como un archivo CSV. Estos componentes no se proporcionan con PostgreSQL, excepto el conector hacia PostgreSQL, sino por los proyectos externos. Estas implementaciones se llaman «wrappers».

Si bien este mecanismo permite utilizar datos externos en modo lectura y escritura, no todos los componentes específicos implementan la escritura de datos.

Una vez que se instala el componente externo, es posible definir una conexión hacia un servicio externo y establecer una correspondencia entre un usuario local de la instancia PostgreSQL con un usuario autorizado para conectarse al servicio externo.

Para terminar, una tabla extranjera se crea para hacer la correspondencia entre una entidad del servicio externo y una relación local a la instancia PostgreSQL.

1. Wrappers

a. Lista de wrappers disponibles

Los wrappers disponibles para la instalación no se entregan con PostgreSQL, excepto los wrappers `postgres_fdw` y `file_fdw`. Según las necesidades, se han desarrollado diferentes wrappers y generalmente están disponibles en forma de código fuente, como software libre.

La wiki de PostgreSQL hace referencia a los proyectos existentes en la siguiente dirección: https://wiki.postgresql.org/wiki/Foreign_data_wrappers y el almacén pgxn integra algunos: http://pgxn.org/tag/fdw/

La siguiente tabla resume las características de los principales wrappers:

nombre	lectura/escritura	pxgn
`oracle_fdw`	Sí	Sí
`mysql_fdw`	Sí	Sí
`tds_fdw (MS-SQL Server, Sybase)`	No	Sí
`sqlite_fdw`	No	No
`redis_fdw`	No	Sí
`couchdb_fdw`	No	Sí
`ldap_fdw`	No	Sí
`hadoop_fdw`	No	No

b. Creación de un wrapper

Este comando hace que el conector externo esté disponible en la base de datos actual. Por defecto, no hay conector instalado. Por lo tanto, es necesario instalar el wrapper como extensión con la sentencia SQL `CREATE EXTENSION` que, en la mayor parte de los casos, induce el comando de creación. La sinopsis es la siguiente:

```
CREATE FOREIGN DATA WRAPPER nombre [ HANDLER handler_función | NO
HANDLER ] [ VALIDATOR validator_función | NO VALIDATOR ]
[ OPTIONS ( opción 'valor' [, ... ] ) ]
```

La opción `HANDLER` define la función llamada durante la creación de una tabla extranjera. La opción `VALIDATOR` define una función utilizada durante la creación posterior de una tabla, un servidor o un usuario. Las otras opciones se definen por los wrappers en sí mismos.

c. Modificación de un wrapper

El comando `ALTER FOREIGN DATA WRAPPER` permite modificar la definición del wrapper según la siguiente sinopsis:

```
CREATE FOREIGN DATA WRAPPER nombre
   [ HANDLER handler_function | NO HANDLER ]
   [ VALIDATOR validator_function | NO VALIDATOR ]
   [ OPTIONS ( opción 'valor' [, ... ] ) ]
```

d. Eliminación de un wrapper

El comando DROP FOREIGN DATA WRAPPER permite eliminar un wrapper según la siguiente sinopsis:

```
DROP FOREIGN DATA WRAPPER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

La opción IF EXISTS evita que se provoque un error durante la eliminación si el wrapper no existe. La opción CASCADE permite eliminar los objetos que dependen del wrapper, como un servidor.

2. Servidores

a. Creación de un servidor

Una vez que se instala el wrapper, es posible definir un servidor que define los argumentos de conexiones para una instancia del tipo del wrapper. La sinopsis del comando es la siguiente:

```
CREATE SERVER nombreservidor [ TYPE 'servidor_tipo' ] [ VERSION
'servidor_versión' ] FOREIGN DATA WRAPPER fdw_name [ OPTIONS
( opción 'valor' [, ... ] ) ]
```

b. Modificación de un servidor

El comando ALTER SERVER permite modificar la definición del servidor según la siguiente sinopsis:

```
ALTER SERVER nombreservidor [ VERSION 'versión' ] [ OPTIONS ( [ ADD |
SET | DROP ] opción ['valor'] [, ... ] ) ]
ALTER SERVER nombreservidor OWNER TO usuario
ALTER SERVER nombreservidor RENAME TO nombreservidor
```

c. Eliminación de un servidor

El comando DROP SERVER permite eliminar un wrapper según la siguiente sinopsis:

```
DROP SERVER [ IF EXISTS ] nombreservidor [ CASCADE | RESTRICT ]
```

La opción IF EXISTS evita que se provoque un error durante la eliminación si el servidor no existe. La opción CASCADE permite eliminar los objetos que dependen del servidor, como un usuario.

3. Usuarios

a. Creación de un usuario

La creación de una asociación entre un usuario existente en la instancia PostgreSQL y un usuario de un servidor externo permite encapsular la información de conexiones según la siguiente sinopsis:

```
CREATE USER MAPPING FOR { usuario | USER | CURRENT_USER |
PUBLIC } SERVER nombreservidor [ OPTIONS ( opción 'valor' [ , ... ] ) ]
```

La palabra clave PUBLIC crea una correspondencia genérica, válida por defecto para todos los usuarios de la instancia PostgreSQL.

Las opciones son específicas de cada wrapper.

b. Modificación de un usuario

El comando ALTER USER MAPPING permite modificar las opciones según la siguiente sinopsis:

```
ALTER USER MAPPING FOR { usuario | USER | CURRENT_USER |
PUBLIC } SERVER nombreservidor OPTIONS ( [ ADD | SET | DROP ] opción
['valor'] [, ... ] )
```

c. Eliminación de un usuario

El comando DROP USER MAPPING permite eliminar un usuario según la siguiente sinopsis:

```
DROP USER MAPPING [ IF EXISTS ] FOR { usuario | USER |
CURRENT_USER | PUBLIC } SERVER nombreservidor
```

La opción IF EXISTS evita que se provoque un error durante la eliminación si el usuario no existe. La opción CASCADE permite eliminar los objetos que dependen del usuario, como una tabla extranjera.

4. Tablas extranjeras

a. Creación de una tabla extranjera

La creación de una tabla extranjera es muy parecida, en la forma, a la creación de una tabla normal, porque es necesario definir localmente los nombres y tipos de datos de las columnas de la tabla remotas. La sinopsis del comando es la siguiente:

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] nombretabla (
   [ nombrecolumna tipodatos [ OPTIONS ( opción 'valor'
[, ... ] ) ] [ COLLATE collation ] [ [ CONSTRAINT nombrerestricción ]
{ NOT NULL |  NULL |  DEFAULT expr } [ ... ] ]
   [, ... ] ] )
   SERVER nombreservidor [ OPTIONS ( opción 'valor' [, ... ] ) ]
```

Se pueden definir opciones para la tabla o para la columna; por ejemplo, cuando la columna remota no tiene el mismo nombre que la columna local.

La opción IF NOT EXISTS no genera errores cuando la tabla ya existe. El nombre de la tabla puede contener un nombre de esquema existente.

b. Modificación de una tabla extranjera

Es posible modificar una tabla extranjera idéntica a una tabla normal: adición, eliminación y modificación de una columna.

La diferencia es la modificación de las opciones de columna o la tabla según el wrapper. La sinopsis del comando es la siguiente:

```
ALTER FOREIGN TABLE [ IF EXISTS ] nombretabla
[ ALTER [ COLUMN ] nombrecolumna OPTIONS ( [ ADD | SET | DROP ]
opción ['valor'] [, ... ]) ] ,
[ OPTIONS ( [ ADD | SET | DROP ] opción ['valor'] [, ... ]) ];
```

c. Eliminación de una tabla extranjera

El comando `DROP USER MAPPING` permite eliminar un usuario según la siguiente sinopsis:

```
DROP FOREIGN TABLE [ IF EXISTS ] nombretabla [, ...] [ CASCADE |
RESTRICT ]
```

La opción `IF EXISTS` evita que se provoque un error durante la eliminación si la tabla no existe. La opción `CASCADE` permite eliminar los objetos que dependen de la tabla, como una vista.

Los índices

Un índice es una buena herramienta para mejorar el rendimiento de la base de datos. La analogía con un libro permite entender el papel que juega un índice en una base de datos. Sin índices, cuando un lector busca una información, debe leer el libro hasta encontrar lo que busca. Según el tamaño del libro y el lugar en el que se sitúa la información, inicio o fin, la búsqueda puede ser larga. En la mayor parte de los libros técnicos, el editor incorpora un índice de algunas páginas, que contiene las palabras clave consideradas susceptibles de ser buscadas para que resulte más fácil encontrar el concepto. El lector no tiene más que buscar en el índice e ir a la página indicada.

Los servidores de bases de datos almacenan los datos en las tablas y deben leer las tablas cuando un usuario busca un dato. El método más sencillo es recorrer secuencialmente una tabla hasta encontrar el dato buscado. Este método funciona bien mientras la tabla tenga un volumen que haga que los tiempos para recorrerla sean correctos. Más allá de un límite que depende del tipo de datos utilizados, del número de columnas de la tabla y del número de tuplas, el recorrido secuencial de la tabla es demasiado largo para obtener un tiempo de respuesta razonable. Se hace necesario crear uno o varios índices, en función de las búsquedas realizadas sobre la tabla.

La analogía con el libro termina aquí. En efecto, un libro tiene un contenido estático, al contrario de lo que sucede con una base de datos, cuyo contenido evoluciona con el tiempo. Esto significa que los índices se deben actualizar al mismo tiempo que la tabla. Cuantos más índices asociados tenga una tabla, más tiempo lleva la actualización durante las operaciones de inserción, actualización o eliminación de los datos. El tiempo susceptible de ganarse en la lectura se pierde durante la escritura.

Por lo tanto, la creación de un índice es un delicado equilibrio entre la mejora deseada en la lectura y la bajada de rendimiento durante la escritura. El sencillo hecho de crear muchos índices sobre una misma tabla puede penalizar.

Además, un índice ocupa un volumen no despreciable en el sistema de archivos y también provoca lecturas y escrituras sobre el disco duro. Para terminar, algunas veces el servidor puede no seleccionar un índice como ruta de acceso a los datos y, por lo tanto, en estos casos un índice puede no tener ningún interés.

El hecho de crear un índice siempre tiene un impacto sobre el rendimiento, pero este impacto será diferente según el tipo de índice. Por lo tanto, es conveniente hacer una evaluación concreta de las necesidades y restricciones durante la escritura de las consultas y la elección de los índices.

1. Creación de un índice

La siguiente sinopsis muestra la creación de un índice:

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] nombre ON table
[ USING [ btree | hash | gist | spgist | gin | brin ] ]
 ( { columna | ( expresión ) }
     [ COLLATE collation ]
     [ classeop ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
   [, ...]
 )
 [ WITH ( argumento = valor [, ...]  ) ]
 [ TABLESPACE espaciotabla ]
 [ WHERE predicat ]
```

La sentencia `UNIQUE` permite verificar la presencia de datos duplicados durante la inserción y actualización de los datos en el índice.

La sentencia `CONCURRENTLY` permite crear un índice sin imponer bloqueos que prohíban la modificación de la tabla por otras transacciones. La creación concurrente de un índice es más lenta, pero preferible en un entorno de producción.

El argumento `FILLFACTOR`, como en el caso de la creación de una tabla, permite optimizar el funcionamiento del índice durante la actualización. Por defecto, el valor, expresado en porcentaje, vale 100, pero se puede bajar, lo que permite al servidor escribir las modificaciones del índice en la misma página del disco. Para cada columna, `COLLATE` permite indicar un valor de collation (que indica la ordenación que se debe realizar), lo que es útil cuando se hace una búsqueda sobre la collation indicada.

Es posible crear índices multicolumna añadiendo simplemente los nombres de las columnas separadas por comas. La elección de estas columnas se realiza observando las columnas utilizadas en las clausulas `WHERE` de las consultas `SELECT`. Es posible utilizar hasta treinta y dos columnas en un índice.

Es posible usar la expresión de una columna, es decir, el resultado de una función aplicada a una columna. El ejemplo más actual es la función `lower()`. Utilizando la expresión `lower(columna)`, son los datos en minúscula los que se indexan y esto es interesante cuando esta expresión también se utiliza en una consulta `SELECT`. Además, las clausulas `ASC` y `DESC` permiten indicar la sentencia de ordenación y `NULL` permite indicar si las tuplas nulas se deben ordenar en primer (`FIRST`) o en último (`LAST`) lugar.

La sentencia «clase de operador» permite indicar las funciones de comparación para cada tipo de dato. Generalmente, la clase del operador por defecto es suficiente. Llegado el caso, hay disponibles otras clases de operadores para usos específicos.

La opción `TABLESPACE` permite seleccionar el espacio de tablas (y, por lo tanto, de índice) utilizado para el almacenamiento físico del archivo del índice. Puede ser pertinente seleccionar un espacio de tablas diferente a la tabla utilizada para repartir las lecturas y escrituras sobre los diferentes discos duros utilizados por los espacios de tablas. En efecto, es frecuente que los índices se usen al mismo tiempo que las tablas a las que hacen referencia.

Para terminar, el predicado `WHERE` permite construir índices parciales, en función de expresiones condicionales, que definen el subconjunto de la tabla sobre la que se construye el índice.

2. Los diferentes tipos de índice

Existen diferentes tipos de índice, que se corresponden con los algoritmos utilizados para escribir y leer los datos indexados. PostgreSQL implementa cinco tipos de índice: B-Tree, Hash, Gist, SP-GiST y GIn. La sentencia USING permite indicar el tipo de índice elegido.

- B-Tree o árbol equilibrado: este tipo de índice es el que se usa por defecto. Es el que más se utiliza.
- Hash: los índices hash solo soportan el operador de igualdad (=). Se destinan a un uso restringido.

- GiST (*Generalized Search Tree*): permite las búsquedas en rangos de valores, como el solapamiento o el hecho de que un valor esté contenido en un rango de valores. Por ejemplo, la dirección IP 10.0.3.12 forma parte del rango 10.0.3.0/24. De manera interna, los rangos almacenados se organizan en forma de árboles, como un B-Tree.
- SP-GiST (*Space Partitioned GiST*): permite las búsquedas de datos particionados y no equilibrados (al contrario de lo que sucedía con GiST).
- GIn (*Generalized Inverted*): permite las búsquedas en los tipos compuestos, como las tablas, `hstore` `json` o las búsquedas de texto completo (FTS). Mientras que un índice clásico considera el conjunto del tipo compuesto como un valor, un índice GIn considera cada elemento de un tipo compuesto como un valor. Por lo tanto, es posible hacer una búsqueda indexada de un miembro de una tabla o de un documento JSON.
- BRIN para *Block Range Index*: apareció en la versión 9.5 de PostgreSQL. Este tipo de índice permite hacer búsquedas en los rangos de valores contenidos en los bloques de datos. Almacenando solo los rangos de valores, resulta un índice muy poco voluminoso respecto al volumen real de los datos, lo que hace que sea un índice adaptado para grandes volúmenes de datos. Pero el dato indexado de esta manera se debe ordenar físicamente en los bloques de datos.

Por lo tanto, la elección del tipo de índice depende de la naturaleza del dato y de la búsqueda sobre el dato, así como del impacto de la presencia del índice en la actualización de los datos de una tabla. En efecto, la actualización de un índice GiST o GIn es más lenta que para un índice B-Tree, incluso si las búsquedas en los datos indexados pueden ser más eficaces.

3. Modificación de un índice

Con la sentencia `ALTER INDEX`, se pueden realizar dos operaciones sobre un índice creado: renombrar el índice y cambiar su espacio de tablas.

La primera operación solo tiene un interés limitado, porque el nombre de un índice no es muy importante:

```
ALTER INDEX [ IF EXISTS ] nombre RENAME TO nuevonombre
```

La segunda operación es más interesante por las mismas razones que para la creación de un índice:

```
ALTER INDEX [ IF EXISTS ] nombre SET TABLESPACE espaciotabla
```

También es posible modificar los argumentos de almacenamiento, como `FILLFACTOR`:

```
ALTER INDEX [ IF EXISTS ] nombre SET ( parametro = valor [, ... ] );
ALTER INDEX [ IF EXISTS ] nombre RESET ( parametro [, ... ] );
```

La opción `IF EXISTS` evita que se provoque un error durante la eliminación si el índice no existe.

4. Eliminación de un índice

El comando `DROP INDEX` permite simplemente eliminar un índice:

```
DROP INDEX [ IF EXISTS ] nombreíndice;
```

La opción `IF EXISTS` evita que se provoque un error durante la eliminación si el índice no existe.

5. Ejemplos de índice

Hay índices que se crean implícitamente sobre las columnas utilizadas como claves primaria o única. Por lo tanto, es inútil crear un índice sobre estas columnas, incluso penaliza el rendimiento. Cuando se declara una clave primaria sobre varias columnas, el índice también lo es.

El resto de las columnas pueden ser objeto de un índice. El siguiente ejemplo muestra la indexación de la columna `prest_fecha_inicio`, de la tabla `prestaciones`:

```
CREATE INDEX prest_fecha_inicio_indice ON prestaciones (prest_fecha_inicio);
```

Un índice sobre varias columnas, optimizando por ejemplo una búsqueda realizada sobre un periodo (una fecha de inicio y una fecha de fin), se puede crear como se muestra a continuación:

```
CREATE INDEX prest_fecha_indice ON prestaciones (prest_fecha_inicio ,
prest_fecha_fin  ) ;
```

Cuando una parte de una tabla no es pertinente durante una búsqueda, es posible crear un índice parcial en función de criterios concretos. Un ejemplo es indexar las fechas de fin de prestaciones únicamente cuando la prestación se confirma:

```
CREATE INDEX prest_fecha_inicio_indice ON prestaciones
( prest_fecha_inicio ) WHERE prest_confirm;
```

Los dos ejemplos anteriores crean índices de tipo B-Tree por defecto. Los otros tipos de índice se utilizan en casos particulares según los tipos de datos utilizados y las necesidades de las consultas `SELECT`.

Por ejemplo, respecto a la tabla `prestaciones`, durante una búsqueda sobre las fechas, una de las necesidades posibles es controlar la superposición de las fechas de prestaciones o el hecho de que una fecha dada esté incluida en una prestación. El tipo de índice `Gist` se hace para esto, con el tipo de datos `datesranges`. Es posible sustituir los atributos `prest_fecha_inicio` y `prest_fecha_fin` por un único atributo y crear un índice de tipo `Gist`.

El siguiente script crea el atributo de tipo `datesranges`, que actualiza los datos desde los dos atributos fuente, los elimina y después crea el índice de tipo `Gist` utilizando la palabra clave `using`:

```
alter table prestaciones add column prest_fechas datesranges;
update prestaciones set prest_fechas =
datesranges( prest_fecha_inicio::date, prest_fecha_fin::date);
alter table prestaciones drop column prest_fecha_inicio, drop column
prest_fecha_fin;

create index on prestaciones using gist ( prest_fechas );
```

Los operadores `@>` y `<@` permiten la búsqueda en un índice de tipo `Gist` para los tipos de datos `datesranges`, así como el resto de los tipos de datos de tipo range. La siguiente consulta `SELECT` permite utilizar el índice:

```
select count(*) from prestaciones where prest_fechas @>
'2017-02-01'::date;
```

El comando EXPLAIN permite examinar el plan de la consulta. Por lo tanto, permite comprobar el uso del índice que se detalla en el capítulo Explotación - Explotación y tareas de mantenimiento - Análisis de una consulta con EXPLAIN. Existen numerosas configuraciones posibles, pero hay que tener cuidado con no multiplicar demasiado los índices para no degradar el rendimiento de la base de datos.

6. Las estadísticas

PostgreSQL recoge estadísticas sobre los datos de cada columna para seleccionar los mejores planes de ejecución de las consultas. Desde la versión 10, además de las estadísticas ya recogidas, es posible recoger estadísticas correlacionadas relativas a varios atributos de una misma tabla, lo que resuelve los problemas de no correlación de las estadísticas cuando se utilizan varios atributos en una búsqueda.

El objetivo de este objeto de estadísticas es mejorar la pertinencia del plan de ejecución de las consultas. Por lo tanto, la elección de los atributos del objeto depende de las consultas utilizadas y de la falta de pertinencia efectiva de los planes de ejecución elegidos, lo que implica un análisis del funcionamiento de la instancia y un correcto conocimiento de las consultas.

La siguiente sinopsis muestra la creación de un objeto de estadísticas:

```
CREATE STATISTICS [ IF NOT EXISTS ] nombre_estadísticas
   [ ( tipo [, ... ] ) ]
   ON columna, columna [, ...]
   FROM nombretabla
```

El tipo de estadística puede ser la dependencia funcional (dependencies) o el número de valores distintos (ndistinto). Cuando el tipo no existe, se tienen en cuenta todos los tipos de estadísticas.

Una vez que se crean las estadísticas, es necesario recoger de nuevo las estadísticas sobre la tabla afectada con el comando ANALYSE.

El siguiente ejemplo muestra la creación de estadísticas sobre los atributos:

```
create statistics prest_fecha_stats on prest_fecha_inicio,
prest_fecha_fin from prestaciones;
analyze prestaciones;
```

Secuencias y atributos de identidad

Una secuencia es un tipo de objeto un poco particular. Se corresponde con un contador que se puede manipular con algunas funciones; principalmente la función nextval(), que permite incrementar el valor del contador y recuperarlo. Esto permite por ejemplo obtener una especie de incremento automático de una columna. Además, es posible compartir una misma secuencia entre varias tablas, creando de esta manera un identificador único intertablas.

El atributo de identidad aparece con la versión 10 de PostgreSQL. Se trata de una funcionalidad muy cercana a las secuencias, pero implementando en PostgreSQL lo que se define en la norma SQL ISO.

Los tipos de datos SERIAL y BIGSERIAL crean automáticamente una secuencia y utilizan la función nextval() en la expresión del valor por defecto de la columna.

En una tabla como la tabla prestaciones, la clave primaria se define como un entero. Es posible utilizar una secuencia para alimentarla automáticamente. El siguiente ejemplo muestra el uso más sencillo:

```
CREATE TABLE prestaciones (   prest_id serial primary key ,
  [ ... ]
);
```

que es equivalente a:

```
CREATE SEQUENCE prest_id_seq;
CREATE TABLE prestaciones (
   prest_id integer default nextval( prest_id_seq )
primary key ,
   [ ... ]
);
```

De esta manera, ya no es necesario proporcionar datos durante la inserción de un registro; la función `nexival()` se encarga de incrementar la secuencia y proporcionar el valor por defecto. Teniendo esto en cuenta, el tipo serial solo es un pseudotipo al que le falta una implementación real que consiste en utilizar un entero y crear una secuencia.

La versión 10 de PostgreSQL aporta el atributo de identidad, similar a una secuencia, pero respetando el estándar SQL. El siguiente ejemplo retoma la tabla `prestaciones`, sustituyendo la secuencia por un atributo de identidad:

```
CREATE TABLE prestaciones (   prest_id int GENERATED BY DEFAULT AS
IDENTITY PRIMARY KEY ,
  [ ... ]
);
```

A diferencia de lo que sucede con una secuencia, el atributo de identidad se muestra claramente como tal una vez que se crea la tabla. La opción `BY DEFAULT` se comporta exactamente como la de una secuencia, pero es posible utilizar la opción `ALWAYS`, que fuerza la utilización del valor generado en lugar del usado durante la inserción de los datos.

Se crea implícitamente una secuencia con el uso del atributo de identidad y, de hecho, las dos nociones son idénticas.

1. Creación de una secuencia

La siguiente sinopsis muestra la sentencia de creación de una secuencia:

```
CREATE SEQUENCE nombre
   [ INCREMENT [ BY ] incremento ]
       [ START [ WITH ] inicio ]
       [ MINVALUE valormin ]
       [ MAXVALUE valormax ]
       [ CYCLE ]
```

El incremento permite indicar el paso de la secuencia, que por defecto es 1.

La opción `START` permite indicar el valor de inicio, que por defecto es cero. Los valores máximos (`MAXVALUE`) y mínimos (`MINVALUE`) pueden ser idénticos y la palabra clave `CYCLE` permite que la secuencia vuelva al valor mínimo una vez que se alcance el valor máximo.

2. Modificación de una secuencia

La modificación de una secuencia permite por ejemplo reinicializar la secuencia.

```
ALTER SEQUENCE nombre [ INCREMENT [ BY ] incremento ]
       [ MINVALUE valormin | NO MINVALUE ]
       [ MAXVALUE valormax | NO MAXVALUE ]
       [ RESTART [ WITH ] inicio ]
```

Las diferentes opciones permiten modificar diferentes argumentos de una secuencia.

3. Eliminación de una secuencia

El comando `DROP SEQUENCE` permite eliminar una secuencia, como en la siguiente sinopsis:

```
DROP [ IF EXISTS ] SEQUENCE nombre
```

La opción `IF EXISTS` evita que se provoque un error durante la eliminación si la secuencia no existe.

Tipos de datos

Existe muchos tipos de datos en PostgreSQL, que se corresponden con la mayor parte de los tipos de datos de la norma SQL.

1. Tipo de datos numéricos

- `smallint`, `int2`: entero con signo sobre 2 bytes.
- `integer`, `int`, `int4`: entero con signo sobre 4 bytes.
- `bigint`, `int8`: entero con signo sobre 8 bytes.
- `serial`, `serial4`: entero sobre 4 bytes con incremento automático. Es un entero asociado a una secuencia.
- `bigserial`, `serial8`: entero sobre 8 bytes con incremento automático. Es un entero asociado a una secuencia.
- `real`, `float4`: número en coma flotante de precisión simple sobre 4 bytes con 6 decimales.
- `double precision`, `float8`: número en coma flotante de precisión doble sobre 8 bytes con 15 decimales.
- `numeric [ (p, s) ]`, `decimal [ (p, s) ]`: número exacto de precisión indicada. Este tipo es particularmente recomendable para los valores monetarios o todos los tipos numéricos donde la parte flotante no deba variar. Las indicaciones se corresponden con el número total de dígitos (`p`) después de la parte decimal (`s`).

No existen tipos u opciones que definan un tipo no firmado. Por lo tanto, los rangos de valores se definen centrados en el cero.

2. Tipo de datos «caracteres»

- `char [ (n) ]`, `character [ (n) ]`: sucesión de caracteres de longitud fija.
- `character varying [ (n) ]`, `varchar [ (n) ]`: sucesión de caracteres de longitud variable limitada.
- `text`: cadena de caracteres de longitud variable ilimitada.

3. Tipo de datos de fechas y horas

- `interval [ (p) ]`: intervalo de tiempo, desde -178 000 000 años hasta +178 000 000, para una precisión hasta el microsegundo.
- `date`: fecha del calendario (año, mes, día), yendo desde 4 713 AC hasta 5 874 897 DC, con una precisión de un día.
- `timestamp [ (p) ] [ without time zone ]`: fecha y hora con el mismo rango de valores que una fecha, pero con una precisión que va hasta el microsegundo.
- `timestamp [ (p)] with time zone`, `timestamptz`: fecha y hora con huso horario.
- `time [ (p) ] [ without time zone ]`: hora del día, desde 00:00:00.00 hasta 24:00:00, con una precisión que va hasta el microsegundo.
- `time [ (p) ] with time zone`, `timetz`: hora del día con huso horario.

4. Tipo de datos «rango de valores»

- `int4range`, `int8range`: rango de enteros delimitado por límites superiores e inferiores para enteros sobre 4 u 8 bytes.
- `numrange`: rango de números delimitado por límites superiores e inferiores.
- `tsrange`, `tstzrange`, `datesranges`: rango de fechas y fechas y hora, con o sin huso horario, delimitado por límites superiores e inferiores.

Los límites de los rangos pueden ser inclusivos o excluyentes, según la manera en que se crean los rangos, gracias a las funciones que se explican más adelante en este capítulo.

5. Tipos de datos varios

PostgreSQL también ofrece algunos tipos de datos con un uso menos actual:

- `inet`, `cidr`: dirección o red IPv4 o IPv6.
- `macaddr`: dirección MAC.
- `bit [ (n) ]`: sucesión de bits de longitud fija.
- `varbit`, `bit varying [ (n) ]`: sucesión de bits de longitud variable.
- `bytea`: dato binario («tabla de bytes»).
- `boolean`, `bool`: booleano (Verdadero/Falso).

- `uuid`: identificador único universal sobre 128 bits representado en forma de 32 dígitos hexadecimales. La extensión `uuid-ossp` permite generar estos identificadores.
- `xml`: un documento o contenido XML se puede almacenar en un tipo `text`, pero este tipo `xml` permite validar el formato del contenido.
- `json`, `jsonb`: objeto JavaScript, documento estructurado que sigue las reglas de la RFC 7159. El tipo `jsonb` almacena datos en binario, mientras que el tipo `json` conserva el texto insertado. El tipo `jsonb` se debe analizar durante la inserción, pero permite un análisis posterior más eficaz y se puede indexar.

6. Tabla de datos

PostgreSQL permite el uso de tablas multidimensionales de datos, independientemente del tipo de datos utilizado. Es posible hacer tablas de enteros, texto, de `uuid` o de `json` de varias dimensiones. Las tablas no tienen longitud fijada durante la declaración. Por lo tanto, son de longitud variable, independientemente de la dimensión.

La declaración de un atributo como tabla se hace utilizando los corchetes [] con el sufijo del tipo de datos deseado. Por ejemplo, la siguiente tabla utiliza una tabla de texto:

```
ALTER TABLE prestaciones ADD COLUMN tags text[];
```

Después, la sintaxis utilizada para insertar los datos se basa en un literal que contiene los corchetes { y } y las comillas para delimitar los valores:

```
UPDATE prestaciones SET tags = '{tech, IT, SQL}' );
```

También es posible utilizar el constructor `ARRAY[]`:

```
UPDATE prestaciones tags = ARRAY[tech, IT, SQL];
```

La lectura de los datos utiliza los corchetes para indicar la clave del dato:

```
SELECT tags[2] FROM prestaciones;
IT
```

Es posible recuperar la tabla como un entero. Existen algunas funciones para manipular estas tablas, explicadas más adelante en este capítulo.

Dominios

Un dominio de datos es una extensión de un tipo de datos asociado a las restricciones y verificaciones que solo permiten definir una vez este conjunto y reutilizarlo en varias tablas. Por ejemplo, un campo `direccion_correo electronico` se puede definir como dominio y reutilizar en toda la aplicación.

1. Creación de un dominio

La sinopsis del comando es la siguiente:

```
CREATE DOMAIN nombre [AS] tipodato
   [ DEFAULT expresión ]
   [ [ CONSTRAINT restricción ]
{ NOT NULL | NULL | CHECK (expresión) } ]
```

La creación de un dominio de datos es muy parecida a la definición de un atributo. Las diferentes opciones son las mismas.

2. Modificación de un dominio

La modificación de un dominio es similar a la modificación de un atributo, utilizando el comando `ALTER DOMAIN`. La diferencia es que la mayor parte de las modificaciones no se aplican a los datos de las tablas que utilizan el dominio. La sinopsis del comando es la siguiente:

```
ALTER DOMAIN nombre { SET DEFAULT expresión | DROP DEFAULT }
ALTER DOMAIN nombre { SET | DROP } NOT NULL
ALTER DOMAIN nombre ADD restricción [ NOT VALID ]
ALTER DOMAIN nombre DROP CONSTRAINT [ IF EXISTS ] restricción
[ RESTRICT | CASCADE ]
ALTER DOMAIN nombre RENAME CONSTRAINT restricción TO restricción2
ALTER DOMAIN nombre VALIDATE CONSTRAINT restricción
ALTER DOMAIN nombre OWNER TO rol
ALTER DOMAIN nombre RENAME TO nvnombre
ALTER DOMAIN nombre SET SCHEMA esquema
```

3. Eliminación de un dominio

Por defecto, la eliminación de un dominio solo es posible cuando este dominio no se utiliza. La sentencia `CASCADE` permite eliminar todos los objetos dependientes de este dominio. La sinopsis del comando es la siguiente:

```
DROP DOMAIN [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

La opción `IF EXISTS` evita que se provoque un error durante la eliminación, si el dominio no existe.

Búsqueda textual

La búsqueda textual en PostgreSQL es un conjunto de funciones que permiten el análisis y la indexación de textos a partir de las raíces de las palabras de este texto, proporcionando los elementos de ordenación en función de la proximidad del resultado con la búsqueda.

Al contrario de lo que sucedía con las expresiones normales o con el operador `LIKE`, este tipo de búsqueda permite la utilización de índices que mejoran el rendimiento y facilita la búsqueda de las raíces de palabras en función de un idioma, lo que posibilita efectuar búsquedas sobre palabras similares.

Las palabras se convierten en raíces o lexemas, que permiten la cercanía de palabras similares con la misma raíz.

Una búsqueda en un texto es la correspondencia entre un documento de tipo `tsvector` y una consulta de tipo `tsquery`. Esta correspondencia es verdadera o falsa y se determina por el operador `@@`.

Búsqueda en una tabla

La búsqueda en una o varias columnas de tipo `text` en una tabla pasa por la conversión a un documento `tsvector`, lo que permite asociarlo a una búsqueda `tsquery`. Por ejemplo, la siguiente consulta permite buscar la palabra clave *'Linux'* en el campo `plandeestudios` de la tabla `formaciones`:

```
SELECT * FROM formaciones WHERE to_tsvector(plandeestudios) @@
to_tsquery( 'Linux');
```

Para mejorar el rendimiento de esta búsqueda, es posible indexar este documento `tsvector`, utilizando `GIN` como tipo de índice:

```
CREATE INDEX formaciones_plandeestudios_es_idx ON formaciones USING
gin(to_tsvector('spanish', plandeestudios));
```

Extensiones

Las extensiones permiten añadir funcionalidades a PostgreSQL.

Una extensión es un conjunto coherente de funciones, tipos de datos, operadores o cualquier otro objeto útil, que ofrece una nueva funcionalidad a PostgreSQL. Una extensión se puede proporcionar por PostgreSQL o cualquier otro proveedor.

Estas extensiones pueden facilitar nuevos tipos de datos, como `ltree` o `prefix`, así como herramientas de administración, como `pg_stat_statement` o `pg_buffercache`.

Cuando se instala PostgreSQL mediante un sistema de paquetes, como en las distribuciones GNU/Linux Debian, Ubuntu, Red Hat o CentOS, un paquete específico contiene las extensiones proporcionadas por PostgreSQL: `postgresql-contrib-10` para Debian y Ubuntu o `postgresql10-contrib` para Red Hat y CentOS.

Las extensiones no proporcionadas por PostgreSQL se instalan por el sistema de paquetes. Entonces, cada extensión es el objeto de un paquete específico; por ejemplo, `postgresql-10-ip4r` para Debian y Ubuntu o `ip4r10` para Red Hat o CentOS.

Estos paquetes hacen referencia a una versión principal específica de PostgreSQL. Por una parte, porque la mayor parte de ellas se escriben en lenguaje C y, por lo tanto, se deben compilar en función de la versión elegida y, por otra parte, porque los archivos se deben instalar en la arborescencia de la versión principal de PostgreSQL.

Cuando se instalan los archivos que contienen el código de la extensión, esta extensión no está disponible inmediatamente. Falta crear los objetos en la base de datos deseada para permitir su utilización, por ejemplo en la creación de una tabla para un tipo de datos o en una consulta `SELECT` para una función.

1. Creación de una extensión

La sentencia `CREATE EXTENSION` permite esta disposición:

```
CREATE EXTENSION [ IF NOT EXISTS ] extensión
    [ WITH ] [ SCHEMA nombresquema ]
    [ VERSION versión ] [ FROM antiguaversión ]
    [ CASCADE ]
```

Por defecto, la última versión de la extensión se instala en el esquema actual. Cuando se indica un esquema, se debe crear con anterioridad.

Las opciones `VERSION` y `FROM` permiten seleccionar una versión específica de la extensión.

Es posible que una extensión necesite otra extensión. Para facilitar esta administración, desde la versión 9.6, es posible utilizar la opción `CASCADE` para instalar automáticamente las extensiones de las que se depende.

El siguiente ejemplo permite instalar la extensión `ip4r`, que permite utilizar rangos de direcciones IP.

Se deben instalar los paquetes específicos de las distribuciones.

```
postgres=# create extensión ip4r;
CREATE EXTENSION
```

El comando `\dx` de `psql` permite listar las extensiones instaladas en la base de datos actual:

```
postgres=# \dx
                List of installed extensiones
 Name   | Version |   Schema   |         Description
--------+---------+------------+------------------------------
ip4r    | 2.0     | public     |
plpgsql | 1.0     | pg_catalog | PL/pgSQL procedural language
(2 rows)
```

El comando `\dx+` permite listar el conjunto de objetos instalados por la extensión:

```
postgres=# \dx+ ip4r
                Objects in extension "ip4r"
                    Object Description
-----------------------------------------------------------
cast from bigint to ip4
cast from cidr to ip4r
cast from cidr to ip6r
cast from cidr to iprange
[...]
```

2. Modificación de una extensión

La modificación de una extensión permite modificar el esquema en el que está instalada o incluso, actualizar la extensión cambiando de versión. La sinopsis del comando es la siguiente:

```
ALTER EXTENSION non UPDATE [ TO version ];
ALTER EXTENSION nombre SET SCHEMA schema;
ALTER EXTENSION nombre ADD object;
ALTER EXTENSION nombre DROP member_object;
```

Las dos últimas formas permiten añadir y retirar un objeto de la extensión, por ejemplo una función o una vista. Estos comandos se utilizan generalmente durante la realización de la extensión para poner en marcha su actualización.

3. Eliminación de una extensión

La eliminación de una extensión es posible por defecto cuando no se utiliza ninguno de los objetos contenidos en la extensión. La opción `CASCADE` permite eliminar esta precaución. La sinopsis del comando es la siguiente:

```
DROP EXTENSION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ];
```

La opción `IF EXISTS` evita que se provoque un error durante la eliminación si la extensión no existe.

4. Administración del código

Una extensión, como conjunto de objetos creados por una sentencia SQL, es conveniente para la encapsulación de objetos específica de las aplicaciones y, por lo tanto, es una herramienta para el desarrollador de aplicaciones o el administrador.

Una extensión se define por un script de sentencias SQL y un archivo `control`, que definen las propiedades de la extensión.

Los objetos SQL contenidos en una extensión pueden ser de cualquier tipo, excepto objetos globales, es decir, objetos cuyo alcance es más amplio que una base de datos, como los espacios de tablas o los roles. Por lo tanto, es posible crear tablas, funciones y vistas, así como tipos de datos o dominios.

Por defecto, los archivos se instalan en el directorio extensión de la instalación de PostgreSQL. Un archivo `Makefile` y el comando `pg_config` permiten la instalación de la extensión en la arborescencia de PostgreSQL.

a. El archivo control

El archivo `control` es un archivo clave-valor, que contiene los argumentos de control que permiten a la sentencia `CREATE EXTENSION` crear la extensión en la base de datos.

Los argumentos son los siguientes:

- `directory = str`: directorio que contiene los scripts SQL; por defecto, el nombre de la extensión.
- `default_version = str`: versión por defecto de la extensión. Cuando la versión no se indica, se deberá hacer durante la llamada a `CREATE EXTENSION`.
- `comment = str`: comentario sobre la extensión. También se puede indicar por medio del comando `COMMENT`, en el script SQL. Por una razón práctica, el comentario en el archivo de control solo debe contener caracteres `ASCII`.
- `encoding = str`: juego de caracteres utilizado en el script SQL. Cuando no se define esta codificación, se supone que es la misma que la de la base de datos en la que se crea la extensión.
- `module_pathname = str`: nombre de la librería que contiene la función cuando se trata de una función escrita en C.
- `requires = str`: lista de extensiones necesarias. Cuando se utiliza la opción `CASCADE`, estas extensiones se instalan automáticamente; sin esta opción, se deben instalar antes.
- `superuser = true|false`: cuando este argumento vale `true`, valor por defecto, solo un rol con el permiso `superuser` puede instalar la extensión. Cuando este argumento vale `false`, solo se requieren los permisos de los objetos de la extensión.

- `relocatable = true|false`: se ha asociado una extensión a un esquema durante la creación; cuando este argumento vale `true`, es posible modificar este esquema con `ALTER EXTENSION`. Por defecto, este argumento vale `false`.
- `schema = str`: cuando una extensión no es reubicable (ver argumento anterior), este argumento permite indicar el esquema de instalación de la extensión.

Ejemplo

El siguiente archivo es un archivo de control muy sencillo, que permite instalar la extensión `clientes`:

clientes.control

```
DEFAULT_VERSION = '1.0'
COMMENT = 'Funciones de usuario de la BD Clientes'
RELOCATABLE = true
```

b. El script SQL

El script SQL contiene las instrucciones que permiten crear los objetos en la base de datos.

Este script se ejecuta en una transacción, de tal manera que las instrucciones `BEGIN` y `COMMIT` y el resto de las instrucciones que no se pueden ejecutar en una transacción, como `VACUUM` o `REINDEX CONCURRENTLY`, no son utilizables en este script.

El nombre del archivo está compuesto por el nombre de la extensión y la versión de la extensión, separado por un doble guión, lo que da para la versión 1.0 de la extensión `clientes`: `clientes--1.0.sql`.

Cuando el script contiene sentencias SQL que permiten una actualización de versión, las dos versiones son idénticas. Por ejemplo: `clientes--0.9--1.0.sql`.

Ejemplo

En el caso de la extensión clientes, el archivo contiene algunas funciones SQL:

clientes--1.0.sql

```
CREATE OR REPLACE FUNCTION importe_total ( importe_brt numeric,
iva numeric default
21 )
RETURNS numeric
LANGUAGE 'plpgsql'
STABLE
AS $$
BEGIN
   RETURN importe_brt * (1 + (iva / 100  ));
END;
$$;

CREATE OR REPLACE FUNCTION importe_factura (id_factura text)
RETURNS numeric
LANGUAGE 'plpgsql'
AS $$
DECLARE
  suma_total numeric;
BEGIN
  RAISE NOTICE 'ID Factura: % ' , id_factura;
  SELECT cantidad_total(sum (lf_importe)) into suma_total
    FROM registros_facturas
    WHERE fact_num = id_factura;
RAISE NOTICE 'Suma: % ' , suma_total;
RETURN suma_total;
END;
$$;
```

c. Instalación de la extensión

La instalación se simplifica por la utilización de un archivo `Makefile`, que utiliza las funcionalidades ofrecidas por PostgreSQL a través del comando `pg_config` y la infraestructura `PGXS`.

El archivo `Makefile` de la extensión `clientes` es el siguiente:

```
EXTENSION = clientes
DATA = clientes--1.0.sql

PG_CONFIG = pg_config
PGXS:= $(shell $(PG_CONFIG) --pgxs)
INCLUDE $(PGXS)
```

A continuación, la llamada del comando `MAKE` permite esta instalación:

```
$ sudo make install
/bin/mkdir -p '/usr/share/postgresql/10/extension'
/bin/mkdir -p '/usr/share/postgresql/10/extension'
/usr/bin/install -c -m 644 .//clientes.control
'/usr/share/postgresql/10/extension/'
/usr/bin/install -c -m 644 .//clientes--1.0.sql
'/usr/share/postgresql/10/extension/'
```

Para terminar, la extensión se puede crear en una base de datos:

```
$ psql -Upostgres -d clientes
psql (10.0)
clientes=# create extension clientes WITH SCHEMA clientes;
CREATE EXTENSION
```

El comando `\dx` de `psql` permite listar las extensiones instaladas y su contenido:

```
clientes=# \dx
                     List of installed extensions
 Name     | Version |   Schema   |              Descripcion
---------+---------+------------+---------------------------------------
clientes | 1.0     | clientes   | Funciones de usuario de la DB Clientes
plpgsql  | 1.0     | pg_catalog | PL/pgSQL procedural language
(2 rows)


postgres=# \dx+ clientes
   Objects in extension "clientes"
         Object description
------------------------------------
function clientes.cantidad_factura(text)
function clientes.cantidad_total(numeric,numeric)
(2 rows)
Operadores y funciones
```

Operadores y funciones

1. Operadores

Las listas que se presentan a continuación resumen los operadores principales disponibles en PostgreSQL.

a. Operadores de comparación

Cuando uno de los operandos comparados es `NULL`, entonces la comparación es `NULL`, salvo cuando el operador es `IS DISTINCT FROM`, siendo `NULL` la ausencia de valor.

- <: devuelve `true` si el operando de la izquierda es más pequeño que el operando de la derecha.
- >: devuelve `true` si el operando de la izquierda es más grande que el operando de la derecha.
- <=: devuelve `true` si el operando de la izquierda es más pequeño o igual que el operando de la derecha.
- >=: devuelve `true` si el operando de la izquierda es más grande o igual que el operando de la derecha.
- =: devuelve `true` si los dos operandos son equivalentes.
- <> o !=: devuelve `true` si los dos operandos no son equivalentes.
- `IS [ NOT ] DISTINCT FROM`: devuelve `true` (o `false`) si los operandos son distintos el uno del otro, incluso si uno de los operandos es NULL.
- `IS [ NOT ] NULL`: devuelve `true` (o `false`) si el operando es `NULL`.

Los siguientes operadores funcionan con tipos de datos compuestos, como las tablas, los rangos de valores o los tipos de datos JSON:

- `@>`: devuelve `true` si el operando de la izquierda contiene al operando de la derecha.
- `<@`: devuelve `true` si el operando de la izquierda está contenido por el operando de la derecha.
- `&&`: devuelve `true` si los dos operandos se solapan.

Los siguientes operadores funcionan con rangos de valores:

- `>>`: devuelve `true` si el operando de la izquierda está a la izquierda del operando de la derecha.
- `<<`: devuelve `true` si el operando de la izquierda está a la derecha del operando de la derecha.
- `&>`: devuelve `true` si el operando de la izquierda no se extiende a la derecha del operando de la derecha.
- `&<`: devuelve `true` si el operando de la izquierda no se extiende a la izquierda del operando de la derecha.
- `-|-`: devuelve `true` si el operando de la izquierda es adyacente al operando de la derecha.

Los siguientes operadores funcionan con los tipos de datos JSON y JSONB:

- `>>`: devuelve `true` si el operando de la izquierda está a la izquierda del operando de la derecha.
- `<<`: devuelve `true` si el operando de la izquierda está a la derecha del operando de la derecha.

Búsqueda de motivos

- `~`, `~*`: devuelve `true` si el operando de la izquierda se corresponde con la expresión racional del operando de la derecha. El asterisco hace que la comparación no haga diferencias entre mayúsculas y minúsculas.
- `LIKE`, `ILIKE`: comprueba la correspondencia entre el operando de la izquierda y el motivo del operando de la derecha. El operador `ILIKE` no hace diferencias entre mayúsculas y minúsculas. Un guión bajo en el motivo se corresponde con un carácter y el carácter `%` se corresponde con varios caracteres, de 0 a N.
- `!~`,`!~*`: devuelve `true` si el operando de la izquierda no se corresponde con la expresión racional del operando de la derecha. El asterisco hace que la comparación no haga diferencias entre mayúsculas y minúsculas.

b. Operadores matemáticos

- `+`: suma el operando de la izquierda con el operando de la derecha.
- `-`: resta el operando de la derecha al operando de la izquierda.
- `*`: multiplica los dos operandos.
- `/`: divide el operando de la izquierda por el operando de la derecha.
- `%`: resto de la división del operando de la izquierda por el operando de la derecha.
- `@`: valor absoluto del operando.

Operadores bit a bit

Estos operadores solo aceptan números enteros como operandos:

- `&`: combina los operandos aplicando un Y lógico.
- `|`: combina los operandos aplicando un O lógico.
- `#`: combina los operandos aplicando un O exclusivo.
- `~`: inversa el operando aplicando un NO lógico.
- `<<`: mueve hacia la izquierda los bits del operando de la izquierda del valor del operando de la derecha.
- `>>`: mueve a la derecha los bits del operando de la izquierda del valor del operando de la derecha.

c. Operadores de subconsultas

Estos operadores permiten comparar conjuntos de valores o el resultado de una subconsulta:

- `EXISTS (subconsulta)`: devuelve `true` si la subconsulta devuelve una tupla o más.
- `expr|tupla IN (subconsulta)`: comprueba la presencia del operando de la izquierda en el conjunto del operando de la derecha. La subconsulta debe devolver tantas columnas como el operando de la izquierda, es decir, una única para una expresión.
- `expr|tupla NOT IN (subconsulta)`: comprueba la ausencia del operando de la izquierda en el conjunto del operando de la derecha. La subconsulta debe devolver tantas columnas como el operando de la izquierda, es decir, una única para una expresión.
- `expr|tupla operador ANY/SOME (subconsulta)`: devuelve `true` si la comparación del operando de la izquierda por el operador indicado se corresponde con al menos una tupla de la subconsulta. La subconsulta debe devolver tantas columnas como el operando de la izquierda, es decir, una única para una expresión. `IN` es equivalente a `= ANY`.
- `expr|tupla operador ALL (subconsulta)`: devuelve `true` si la comparación del operando de la izquierda por el operador indicado se corresponde con todas las tuplas de la subconsulta. La subconsulta debe devolver tantas columnas como el operando de la izquierda, es decir, una única para una expresión. `NOT IN` es equivalente a `<> ANY`.

d. Otros operadores

- `BETWEEN`: comprueba la presencia del operando de la izquierda en el intervalo del operando de la derecha.
- `NOT`: inversión lógica.
- `AND`: y lógica.
- `OR`: o lógica.
- `::`: operador de transtipado explícito. El nombre del tipo se debe indicar a continuación, por ejemplo: `double::int`, que convierte el número flotante en entero.
- `||`: operador de concatenación de cadenas de caracteres.

e. Expresiones

- `COALESCE(valor [, ...])`: expresión que devuelve el primer valor no nulo.
- `NULLIF(valor1, valor2)`: expresión que devuelve `null` si los valores son equivalentes.
- `GREATEST(valor [, ...])`, `LEAST(valor [, ...])`: expresión que devuelve el valor más grande o más pequeño.
- `CASE`: expresión condicional que permite, en función de condiciones, devolver diferentes resultados, como la expresión if/else de otros lenguajes. La sinopsis de la expresión es la siguiente:

```
CASE WHEN condición THEN resultado [WHEN ...]
    [ELSE resultado]
END
```

Es posible encadenar varias expresiones `WHEN` y terminar con un resultado por defecto con `ELSE`. Cuando no es verdadera ninguna de las condiciones y no hay casos por defecto, el resultado es `NULL`.

Ejemplo

```
SELECT
CASE WHEN col1 < 1 THEN 'cero'
WHEN col1 > = 1 THEN 'un'
ELSE 'otro'
END
FROM tabla;
```

Cuando el valor del atributo `col1` es inferior a 1, entonces se utiliza el valor '`cero`', y así sucesivamente para el resto de las cláusulas `WHEN`. Después, si no se verifica ninguna condición, se utiliza el valor '`otro`'.

2. Funciones

PostgreSQL ofrece un determinado número de funciones que permiten manipular los datos. Las listas que se presentan más adelante muestran algunas de funciones más utilizadas.

a. Funciones sobre números

- `abs(x)`: valor absoluto.
- `sign(dp | numeric)`: asigna un signo al argumento como sigue: `-1`, `0` o `+1`.
- `ceil(dp | numeric)`: redondea al entero superior.
- `floor(dp | numeric)`: redondea al entero inferior.
- `mod(y, x)`: resto de la división de los argumentos.
- `div(y numeric, x numeric)`: cociente de la división de los argumentos.
- `power(a dp | numeric, b dp | numeric)`: `a` elevado a la potencia `b`.
- `sqrt(dp | numeric)`: raíz cuadrada.
- `cbrt(dp)`: raíz cúbica.

- `round(dp | numeric)`: redondea al entero el más próximo.
- `round(v numeric, s int)`: redondea a `s` decimales.
- `ceil(dp | numeric), floor(dp | numeric)`: redondea al entero respectivamente superior o inferior del argumento.
- `trunc(dp | numeric)`: trunca eliminando la parte decimal.
- `trunc(v numeric, s int)`: trunca dejando solo `s` decimales.
- `random()`: devuelve una doble precisión comprendida entre 0 y 1, 0 incluido. La cualidad aleatoria depende de la implementación del sistema.
- `log(dp | numeric), log(b numeric, x numeric)`: logaritmo de base 10 o de base `b`.
- `ln(dp | numeric)`: logaritmo neperiano.
- `exp(dp | numeric)`: exponencial.
- `radians(dp), degrees(dp)`: calcula el radián o el grado del argumento grado o radián proporcionado.
- `acos(x), asin(x), atan(x), atan2(y, x), cos(x), cot(x), sin(x), tan(x)`: funciones trigonométricas. El tipo de datos, tanto de entrada como de salida, es `double precision`.
- `Pi()`: devuelve la constante PI.

Cuando una función acepta diferentes tipos de datos como entrada, el tipo de datos devuelto es el mismo que el de la entrada.

b. Funciones sobre cadenas de caracteres

- `upper (string)`: convierte una cadena en mayúsculas.
- `lower (string)`: convierte una cadena en minúsculas.
- `initcap (text)`: convierte la primera letra de cada palabra en mayúscula y el resto en minúsculas.
- `substring(string [from int] [for int])`: extrae una subcadena.
- `substr(string, from [, número])`: extrae la subcadena.
- `replace(text, from, to)`: sustituye en `text` todas las ocurrencias de la sub-cadena `from` por la subcadena `to`.
- `overlay(string placing string from int [for int])`: sustituye los caracteres en función de las posiciones de los datos.
- `format(forme [, arg [, ...]] )`: como la función C `sprintf`, formatea los argumentos según la cadena proporcionada.
- `concat(arg [, ...] ), contat_ws(sep, arg [, ...] )`: concatena los argumentos, como el operador ||. La función `concat_ws()` permite añadir un separador.
- `trim( [leading | trailing | both] chars from string), ltrim(string, chars), rtrim(string, chars), btrim(string, chars)`: recorta la cadena `string` en los extremos indicados (izquierda para `leading` o `ltrim`, derecha para `trailing` o `rtrim`, o los dos). Se pueden indicar varios caracteres en el argumento `chars`, que es un espacio por defecto.

- `lpad(string, length [, fill] ), rpad()`: rellena la cadena `string`, respectivamente a izquierda y derecha, con la longitud `length` indicada, con la cadena `fill` deseada y los espacios por defecto. Atención, si la cadena es más larga que la longitud indicada, entonces se trunca.
- `left(str, n), right(str, n)`: devuelve los `n` caracteres respectivamente a izquierda o derecha de la cadena `str`. Cuando `n` es negativo, estas funciones eliminan los `n` primeros caracteres de la cadena `str`.
- `chr(int)`: carácter correspondiente al código ASCII dado.
- `ascii(text)`: código ASCII del primer carácter del argumento.
- `position(substring in string), strpos(string, substring)`: ubicación de la subcadena especificada.
- `length(string)`: número de caracteres en la cadena.
- `byte_length(string)`: número de bytes en una cadena.
- `bit_length(string)`: número de bits en una cadena.
- `char_length(string)`: número de caracteres en una cadena.
- `md5(text)`: calcula la clave MD5 de prueba, devolviendo el resultado en hexadecimal.
- `pg_client_encoding()`: nombre de la codificación cliente actual.
- `to_hex(int | bigint)`: convierte el entero en su representación hexadecimal equivalente.
- `translate(text, from , to )`: todo carácter en `text` que se corresponda con un carácter en el conjunto `from` se sustituye por el carácter correspondiente del conjunto `to`.
- `encode(bytea, tipo )`: codifica los datos binarios en una representación en ASCII únicamente. Los tipos soportados son: `base64`, `hex`, `escape`.
- `decode(text, tipo )`: descodifica los datos binarios a partir de la cadena codificada previamente con `encode`. El tipo de argumento es el mismo que `encode`.
- `quote_literal(any)`: devuelve el argumento `any` como literal, utilizable en una consulta SQL. Se puede usar cualquier tipo de datos y se transtipa en una cadena de caracteres.
- `quote_ident(str)`: devuelve la cadena `str` como identificador, por ejemplo un nombre de tabla.
- `quote_nullable(any)`: idéntica a `quote_literal()`, pero devuelve la cadena `NULL` si el argumento es null.
- `regexp_matches(str, pattern [, flags] )`: devuelve las subcadenas correspondientes a los motivos de la expresión normal `pattern` en forma de tabla de cadenas.
- `regexp_replace(str, pattern, repl [, flags])`: sustituye las subcadenas correspondientes al motivo de la expresión normal `pattern` con la cadena `repl`.
- `regexp_split_to_array(str, pattern [, flags] )`: escinde la cadena `str`, según el motivo `pattern`, en una tabla de cadenas de caracteres.
- `regexp_split_to_table(str, pattern [, flags] )`: escinde la cadena `str`, según el motivo `pattern`, en tuplas.
- `split_part(str, d, n)`: devuelve el miembro `n` de la cadena `str` escindida según el delimitador `d`.

c. Funciones de fecha

Las siguientes funciones devuelven fechas, horas o fechas con hora. Estas funciones siguen el estándar SQL y devuelven valores que se corresponden con el inicio de la transacción en la que se llaman. Una precisión puede indicar el número de dígitos de una fracción de segundo.

Por lo tanto, las fechas y horas actuales devueltas tienen el mismo valor durante una transacción dada, lo que permite obtener una consistencia del dato para una transacción.

- `CURRENT_DATE`: devuelve la fecha actual.
- `CURRENT_TIME [ (precision) ]`: devuelve la hora actual con el huso horario.
- `CURRENT_TIMESTAMP [ (precision) ]`: devuelve la fecha y la hora actuales con el huso horario.
- `LOCALTIME [ (precision) ]`: devuelve la hora actual sin el huso horario.
- `LOCALTIMESTAMP [ (precision) ]`: devuelve la fecha y la hora actual sin el huso horario.

Las siguientes funciones devuelven las fechas y horas que se corresponden con diferentes momentos: inicio de la transacción, de la consulta o instantánea.

- `transaction_timestamp()`: sinónimo de `CURRENT_TIMESTAMP`, explicitando en su nombre el momento de la fecha y hora devueltas.
- `statement_timestamp()`: devuelve la fecha y la hora del inicio de la consulta actual. De esta manera, varias llamadas a la función en una misma consulta devuelven el mismo valor.
- `clock_timestamp()`: devuelve la fecha y la hora del momento de la llamada de la función. De esta manera, varias llamadas a la función dentro de una misma consulta devuelven valores diferentes.
- `timeofday()`: como `clock_timestamp()`, devuelve la fecha y hora actuales, pero en forma de texto.
- `now()`: sinónimo de `transaction_timestamp()`.

Las siguientes funciones permiten obtener una parte de una fecha y de una hora, ya sea truncándola o extrayendo una parte.

- `date_trunc(text, timestamp | interval )`: trunca hasta la precisión específica. El dato devuelto es del mismo tipo que el de la entrada.
- `extract (text from timestamp | interval)`, `date_part(text, timestamp | interval )`: obtener un subcampo. El dato devuelto es de tipo `double precision`.

La siguiente tabla resume los campos disponibles para estas funciones:

Campo	date_trunc	date_part	descripción
microseconds	x	x	Microsegundo
milliseconds	x	x	Milisegundo
second	x	x	Segundo
epoch		x	Número de segundos desde el 1 de enero del 1970 hasta medianoche. Para un intervalo, el número de segundos
minute	x	x	Minuto
hour	x	x	Hora
day	x	x	Día
dow, isodow		x	Día de la semana, desde 0 (domingo) hasta 6 (sábado) o desde 1 (lunes) hasta 7 (domingo) según ISO 8601
doy		x	Día del año
week	x	x	Semana según ISO 8601
month	x	x	Mes
quarter	x	x	Trimestre
year	x	x	Año
isoyear		x	Año según ISO 8601
decade	x	x	Década
century	x	x	Siglo
millennium	x	x	Milenio
timezone		x	Huso horario
timezone_hour		x	Hora del huso horario
timezone_minute		x	Minuto del huso horario

- `age(timestamp [, timestamp ])`: resta los argumentos y devuelve un intervalo. Cuando el segundo argumento no se proporciona, la función utiliza la fecha actual en su lugar.
- `isfinite(timestamp | interval )`: verifica si un tiempo es finito.
- `justify_hours(interval)`: ajusta el intervalo para que los periodos de 24 horas se representen como los días.
- `justify_days(interval)`: ajusta el intervalo para que los periodos de 30 días se representen como los meses.
- `justify_interval(interval)`: ajusta el intervalo para que los periodos de 30 días se representen como los meses y los de 24 horas como los días.
- `make_date(year int, month int, day int)`: crea una fecha a partir de los campos.

- `make_interval(years int DEFAULT 0, months int DEFAULT 0, weeks int DEFAULT 0, days int DEFAULT 0, hours int DEFAULT 0, mins int DEFAULT 0, secs double precision DEFAULT 0.0)`: crea un intervalo a partir de los campos. Todos los campos son opcionales y valen cero cuando se omiten.
- `make_time(hour int, min int, sec double precision)`: crea una hora a partir de los campos.
- `make_timestamp(year int, month int, day int, hour int, min int, sec double precision)`: crea una fecha con hora a partir de los campos.
- `make_timestamptz(year int, month int, day int, hour int, min int, sec double precision, [ timezone text ])`: crea una fecha con hora a partir de los campos.
- `pg_sleep(seconds)`: espera durante el número de segundos indicado.
- `pg_sleep_for(interval)`: espera durante el intervalo indicado.
- `pg_sleep_until(timestamp with time zone)`: espera hasta la fecha y hora indicadas.

d. Funciones de manipulación de las secuencias

- `nextval(sequence)`: incrementa la secuencia y devuelve el nuevo valor.
- `currval(sequence)`: valor de retorno obtenido más recientemente `con nextval` para la secuencia especificada.
- `lastval()`: valor de retorno obtenido más recientemente con `nextval`.
- `setval (sequence , bigint [ , boolean ])`: inicializa el valor actual de la secuencia. El booleano indica si la siguiente llamada a `nextval()` avanza o no el contador.

e. Funciones de agregación

- `sum(expresión)`: suma la expresión para todos los valores de la entrada.
- `count(* | expresión)`: número de valores de la entrada, sin contar los valores nulos para la expresión.
- `max(expresión)`: valor máximo de la expresión para todos los valores de la entrada.
- `min(expresión)`: valor mínimo de la expresión para todos los valores de la entrada.
- `stddev(expresión)`: desviación estándar de los valores de la entrada.
- `avg(expresión)`: media (en sentido aritmético) de todos los valores de la entrada.
- `variance(expresión)`: varianza simple de los valores de la entrada (cuadrado de la desviación).
- `array_agg(expresión)`: agrega los valores en una tabla.
- `string_agg(expresión, del)`: agrega en una cadena los valores separados por un delimitador.
- `xmlagg(xml)`: concatena los documentos XML.
- `json_agg(tupla), jsonb_agg(tupla)`: agrega las tuplas a una tabla JSON o JSONB.

- `json_object_agg(clave,valor)`, `jsonb_object_agg(clave,valor)`: añade a una tabla JSON o JSONB las parejas clave-valor.
- `bit_and( int | bit )`,`bit_or( int | bit )`: máscara lógica `and` y `or` de los valores enteros o de las palabras de bits.
- `bool_and(bool)`,`every(bool)`,`bool_or(bool)`: aplica el operador lógico `and` u `or` a los booleanos agregados. `every()` es un sinónimo de `bool_and()`.

Agregados estadísticos

- `corr(Y, X)`: coeficiente de correlación.
- `covar_pop(Y, X)`,`covar_samp(Y, X)`: calcula la covarianza.
- `stddev(expresión)`,`stddev_samp(expresión)`,`stddev_pop (expresión)`: calcula la desviación estándar.
- `variance(expresión)`, `var_samp(expresión)`, `var_pop(expresión)`: calcula la varianza.
- `mode() WITHIN GROUP (ORDER BY sort_expr)`: devuelve el valor de la entrada más frecuente.
- `percentile_cont(fraccion) WITHIN GROUP (ORDER BY sort_expr)`: devuelve el centil continuo en función de la fracción de la entrada.
- `percentile_cont(fracciones) WITHIN GROUP (ORDER BY sort_expr)`: devuelve una tabla de centiles continuos en función de la tabla de fracciones de la entrada.
- `percentile_disc(fraccion) WITHIN GROUP (ORDER BY sort_expr)`: devuelve el primer valor correspondiente a la fracción de la entrada.
- `percentile_disc(fracciones) WITHIN GROUP (ORDER BY sort_expr)`: devuelve una tabla de valores en función de la tabla de fracciones de la entrada.

f. Funciones de ventana

- `row_number()`: número de registros en la ventana.
- `rank()`: rango del registro actual en la ventana con intervalo. Los registros indicados tienen el mismo rango.
- `dense_rank()`: rango del registro actual en la ventana sin intervalo.
- `percent_rank()`: rango relativo del registro actual en porcentaje.
- `cume_dist()`: rango relativo del registro actual en fracción del número total de registros.
- `ntile(int)`: partición de la ventana, desde 1 hasta el valor que se pasa como argumento.
- `lag( campo [, offset [, default ]])`: devuelve el campo correspondiente a un registro de la ventana anterior al registro actual, con desplazamiento de 1 por defecto. Se devuelve el valor por defecto si no se encuentra ningún registro.
- `lead(value [, offset [, default ]])`: devuelve el campo correspondiente a un registro de la siguiente ventana al registro actual con desplazamiento de 1 por defecto. Se devuelve el valor por defecto si no se encuentra ningún registro.
- `first_value(valor)`: devuelve el primer valor de la ventana para el campo indicado.

- `last_value(valor)`: devuelve el último valor de la ventana para el campo indicado.
- `nth_value(valor, nth)`: devuelve el valor de rango `nth` del valor en la ventana.

g. Funciones de manipulación de las tablas

- `array_length(tabla, dim)`: número de elementos para la dimensión `dim`.
- `cardinality(tabla)`: número total de elementos de la tabla, todas dimensiones juntas.
- `array_lower(tabla, dim)`: límite inferior de la dimensión `dim` de la tabla.
- `array_upper(tabla, dim)`: límite superior de la dimensión `dim` de la tabla.
- `array_ndims(tabla)`: número de dimensiones de la tabla.
- `array_dims(tabla)`: representación de las dimensiones de la tabla en forma de cadena.
- `array_fill( valor, tamaño [] , [, offset[] ])`: rellena una tabla con el tamaño dado, con el valor proporcionado, eventualmente desplazando el registro inferior de la tabla.
- `array_cat(tabla, tabla)`: concatena dos tablas.
- `array_append(tabla, valor)`: añade un elemento al final de la tabla.
- `array_prepend(valor, tabla)`: añade un elemento al inicio de la tabla.
- `array_remove(tabla, valor)`: elimina de la tabla en una dimensión todos los elementos que se corresponden con el valor indicado.
- `array_replace(tabla, val1, val2)`: sustituye en la tabla las ocurrencias de `val1` encontradas por `val2`.
- `array_to_string(tabla, delim [, nul])`: concatena los elementos de la tabla en una cadena, separados por el delimitador `delim`, sustituyendo eventualmente los valores `NULL` por el valor indicado.
- `string_to_array(str, sep [, nul])`: separa la cadena en una tabla según el separador `sep`, sustituyendo eventualmente el texto `null` por un elemento `NULL`.
- `unnest(anyarray)`: transforma una tabla en tuplas con un solo atributo.
- `unnest(anyarray, anyarray [, ...])`: transforma un conjunto de tablas en tuplas, cada tabla con un atributo. Utilizable únicamente en una sentencia `FROM`.

h. Funciones de manipulación de datos JSON

Creación de documentos JSON

- `to_json(element)`, `to_jsonb(element)`: devuelve un objeto JSON o JSONB. Se convierten recursivamente los tipos compuestos y las tablas, si existe el transtipado y los tipos de datos escalares se convierten en texto, con los caracteres de escape necesarios.
- `array_to_json(tabla [, pretty])`: transforma una tabla PostgreSQL en una tabla JSON. Una tabla de varias dimensiones se convierte en una tabla de tablas JSON. Cuando el booleano `pretty` es `true`, las estructuras JSON se completan para mejorar la legibilidad.
- `row_to_json(tupla [, pretty])`: transforma una tupla en un objeto JSON. Cuando el booleano `pretty` es `true`, las estructuras JSON se completan para mejorar la legibilidad.

- `json_build_array(VARIADIC "any"), jsonum_build_array (VARIADIC "any")`: construye un objeto JSON o JSONB a partir de los datos heterogéneos que se pasan como argumentos.
- `json_build_object(VARIADIC "any"), jsonum_build_object (VARIADIC "any")`: construye un objeto JSON o JSONB de pares clave/valor a partir de la alternancia de los argumentos proporcionados.
- `json_object(text[]), jsonum_object(text[])`: construye un objeto JSON o JSONB a partir de una tabla de texto. La tabla de una dimensión debe tener un número par de elementos para tomar, de manera alternativa, las claves y los valores. Una tabla de dos dimensiones debe tener tablas internas de dos elementos para deducir de ellas las parejas clave/valor.
- `json_object(claves text[], valores text[]),json_object (claves text[], valores text[])`: construye un objeto JSON o JSONB, a partir de las tablas claves y valores.

Tratamiento de los objetos JSON

Las siguientes funciones existen en dos versiones para los dos tipos de datos JSON: `json`, formato texto, y `jsonb`, formato binario.

- `json_array_length(json), jsonb_array_length(jsonb)`: devuelve el número de elementos de primer nivel del objeto JSON.
- `json_each(json), jsonb_each(jsonb)`: desarrolla el primer nivel del objeto JSON en un conjunto de pares clave/valor, siendo los valores de tipo `json` o `jsonb`.
- `json_each_text(json), jsonb_each_text(jsonb)`: desarrolla el primer nivel del objeto JSON en un conjunto de pares clave/valor, siendo los valores de tipo `text`.
- `json_extract_path(json, VARIADIC ruta text[]), jsonb_ extract_path(jsonb, VARIADIC ruta text[])`: devuelve el valor JSON identificado por la ruta de la tabla de `text`. Es equivalente al operador #>.
- `json_extract_path_text(json, VARIADIC ruta text[]), jsonum_extract_path_text(jsonb, VARIADIC ruta text[])`: devuelve el valor JSON identificado por la ruta de la tabla de `text`. Es equivalente al operador #>>.
- `json_object_keys(json), jsonb_object_keys(jsonb)`: devuelve el conjunto de claves del primer nivel del objeto JSON.
- `json_populate_record(base, json), jsonb_populate_record (base, jsonb)`: desarrolla el objeto JSON en una tupla, cuyos tipos de atributos se corresponden con el elemento `base`, que puede ser un objeto `NULL` de tipo compuesto, como una tabla.
- `json_populate_recordset(base, json), jsonb_populate_ recordset(base, jsonb)`: desarrolla los elementos de primer nivel del objeto JSON y un conjunto de tuplas cuyos tipos de atributos se corresponden con elemento base, que puede ser un objeto `NULL` de tipo compuesto, como una tabla.
- `json_array_elements(json), jsonb_array_elements(jsonb)`: desarrolla un objeto JSON en un conjunto de valores JSON.
- `json_array_elements_text(json), jsonb_array_elements_ text(jsonb)`: desarrolla un objeto JSON en un conjunto de cadenas de caracteres.
- `json_typeof(json), jsonb_typeof(jsonb)`: devuelve los tipos de datos de los objetos de primer nivel en forma de texto. Los tipos posibles son: `object`, `array`, `string`, `number`, `boolean` y `null`.

- `json_to_record(json)`, `jsonb_to_record(jsonb)`: construye una tupla a partir de un objeto JSON, con ayuda del tipo que describe la llamada de la función, como para una función llamada SRF.
- `json_to_recordset(json)`, `jsonb_to_recordset(jsonb)`: construye un conjunto de tuplas a partir de un objeto JSON, con ayuda del tipo que describe la llamada de la función, como para una función llamada SRF.

Por lo tanto, el tipo de datos de los elementos contenidos en el documento JSON se utiliza durante la llamada a la función para indicar la composición de la tupla o tuplas devueltas.

Ejemplo:

```
select * from json_to_record('{"a":1, "c":"bar"}')
as x(a int, c text)
 a | c
---+-----
1 | bar
(1 row)
```

Los tipos de datos utilizados se deben corresponder con los datos presentes en el documento JSON.

- `json_strip_nulls(json)`, `jsonum_strip_nulls(jsonb)`: devuelve un objeto JSON o JSONB omitiendo todos los campos NULL en el objeto que se pasa en la entrada.
- `jsonum_set(target jsonb, path text[], new_value jsonb [, create_missing boolean])`: devuelve un objeto JSONB modificado, en función de las rutas indicadas en el argumento `path` y de los valores expresados en JSON. Cuando los elementos no existen en el documento JSON, el booleano `create_missing` las permite crear.
- `jsonum_insert(target jsonb,path text[],new_value jsonb [,insert_after boolean])`: devuelve un objeto JSONB en el que se añaden los valores del objeto `new_value`, siguiendo las rutas de la tabla `path`. Por defecto, los elementos se añaden desde la ruta devuelta, salvo si el booleano `after_insert` se pasa a `true`.
- `jsonum_pretty(jsonb)`: devuelve una cadena de caracteres correspondiente al objeto JSONB, correctamente indentado para ser legible.

i. Funciones de manipulación de datos XML

- `xmlroot(xml, versión | no value [, standalone yes|no|no value])`: modifica las propiedades de la raíz XML del documento que se pasa como argumento.
- `xmlforest(contenido [AS nombre] [, ...])`: produce una secuencia de elementos XML.
- `xmlelement(nombre [, xmlattributes (valor [AS atributo] [, ... ])] [, contenido, ...])`: produce un elemento XML con los nombres, atributos y contenidos indicados.
- `xmlconcat(xml[, ...])`: concatena los elementos XML para formar un conjunto de elementos XML.
- `xmlcomment(texto)`: devuelve un comentario XML.
- `xmlpi(cible [, contenido])`: produce una instrucción de tratamiento XML.
- `xmlagg(xml)`: agregado que concatena los documentos XML.

Predicados

- `xml IS DOCUMENT`: devuelve `true` si el elemento proporcionado es un documento XML válido.
- `XMLEXISTS(xpath PASSING xml)`: devuelve `true` si la expresión `xpath` devuelve al menos un elemento.
- `xml_is_well_formed(text)`, `xml_is_well_formed_document (text)`, `xml_is_well_formed_content(text)`: devuelve `true` si el argumento proporcionado es respectivamente XML, un documento o un contenido.

Tratamientos

- `xpath(xpath, xml [, nsarray])`: devuelve el resultado de la expresión `xpath` sobre el documento XML.
- `xpath_exists(xpath, xml [, nsarray])`: devuelve `true` si la expresión `xpath` devuelve, al menos, un elemento.
- `table_to_xml(regclass, nulls, forest, ns)`,`query_to_xml (query, nulls, forest, ns)`, `cursor_to_xml(cursor, nulls, forest, ns)`: convierte respectivamente una tabla, consulta o cursor en una arborescencia XML. Cuando el booleano `nulls` es `false`, la columna no se incluye. Cuando el booleano `forest` es `true`, cada tupla es un elemento cuyo nombre es el nombre de la tabla. Si este booleano es `false`, cada tupla es un elemento llamado `row`, hijo del elemento de la tabla. La cadena `ns` define el espacio de nombres del documento XML producido.
- `table_to_xmlschema(regclass, nulls, forest, ns)`, `query_to_xmlschema(query, nulls, forest, ns)`, `cursor_to_xmlschema(cursor, nulls, forest, ns)`: convierte respectivamente una tabla, consulta o cursor en un esquema XML. Cuando el booleano `nulls` es `false`, la columna no se incluye. Cuando el booleano `forest` es `true`, cada tupla es un elemento cuyo nombre es el nombre de la tabla. Si este booleano es `false`, cada tupla es un elemento llamado `row`, hijo del elemento de la tabla. La cadena `ns` define el espacio de nombres del esquema XML generado.
- `table_to_xml_and_xmlschema(regclass, nulls, forest, ns)`, `query_to_xml_and_xmlschema(query, nulls, forest, ns)`: convierte respectivamente una tabla o una consulta en un documento y un esquema XML. Cuando el booleano `nulls` es `false`, la columna no se incluye. Cuando el booleano `forest` es `true`, cada tupla es un elemento cuyo nombre es el nombre de la tabla. Si este booleano es `false`, cada tupla es un elemento llamado `row`, hijo del elemento de la tabla. La cadena `ns` define el espacio de nombres del esquema XML generado.
- `schema_to_xml(schema, nulls, forest, ns)`, `schema_to_xmlschema(schema, nulls, forest, ns)`,`schema_to_xml_and_xmlschema(schema, nulls, forest, ns)`: convierte un esquema en un documento o un esquema XML o los dos. Cuando el booleano `nulls` es `false`, la columna no se incluye. Cuando el booleano `forest` es `true`, cada tupla es un elemento cuyo nombre es el nombre de la tabla. Si este booleano es `false`, cada tupla es un elemento llamado `row`, hijo del elemento de la tabla. La cadena `ns` define el espacio de nombres del esquema XML generado.
- `database_to_xml(nulls, forest, ns)`, `database_to_xmlschema(nulls, forest, ns)`, `database_to_xml_and_xmlschema(nulls, forest, ns)`: convierte una base de datos en un documento o un esquema XML o los dos. Cuando el booleano `nulls` es `false`, la columna no se incluye. Cuando el booleano `forest` es `true`, cada tupla es un elemento cuyo nombre es el nombre de la tabla. Si este booleano es `false`, cada tupla es un elemento llamado `row`, hijo del elemento de la tabla. La cadena `ns` define el espacio de nombres del esquema XML generado.

- `xmltable( '//path/to/element' PASSING attr COLUMNS {attr_name tipo PATH 'xmlpath', ...})`: devuelve un documento XML en forma de tabla. Esta funcionalidad es una novedad de la versión 10 de PostgreSQL. La utilización de esta función es similar a la de una función `SRF`, aunque un poco particular, y está asociada al nombre de la tabla en la que se encuentran los datos XML. El principio es proporcionar una ruta en el documento XML después de indicar los elementos o atributos XML que forman los atributos de la tabla. Por ejemplo, considerando una tabla `formaciones` que contiene un atributo `plan`, la siguiente consulta permite visualizar una lista con los ítems del plan de estudios:

```
select * from formaciones, xmltable('//plan/items'
PASSING plan COLUMNS item tipo text PATH 'item',
duración tipo numeric PATH 'item/@duration'); ,
```

Funciones de manipulación de los rangos de valores

- `numrange( numeric, numeric [, text])`: crea un rango de valores de tipo numérico.
- `int4range(int4, int4 [, text]) , int8range(int8, int8 [, text])`: crea un rango de valores de números enteros.
- `datesranges(date, date [, text])`: crea un rango de valores de fechas.
- `tsrange(timestamp, timestamp [, text]) , tstzrange(timestamptz , timestamptz [, text])`: crea un rango de valores de fechas y hora con o sin huso horario.

Los límites de los rangos pueden ser inclusivos o excluyentes, según la notación utilizada en el tercer argumento. Los caracteres [o (y] o) se utilizan para limitar los rangos. Los corchetes abiertos y cerrados designan los límites inclusivos, y los paréntesis abiertos y cerrados, los límites excluyentes. Por defecto, en ausencia del tercer argumento, el límite inferior es inclusivo y el superior, excluyente, de tal manera que el valor por defecto vale '[)'.

Los límites superior e inferior pueden no tener valor utilizando la palabra clave `NULL` como argumento o el valor 'unbounded'. Los siguientes ejemplos crean respectivamente un rango de enteros desde 4 hasta 8 con el límite superior inclusivo, y después un rango de fechas que va desde la fecha actual hasta el infinito:

```
select int4range( 4, 8, '[]');
select date_range( current_date, null );
```

Es posible crear rangos de valores utilizando el operador de `transtipado::`.

```
select '[4,8]'::int4range;
```

Manipulación de los datos

1. Inserción de datos

Existen dos métodos para alimentar las tablas con datos. El primero utiliza la sentencia `INSERT`, como en la norma SQL, y el segundo la sentencia `COPY`, principalmente para los casos de volúmenes de datos importantes durante la inserción.

a. La sentencia INSERT ... INTO

La sentencia INSERT respeta la notación de la norma SQL aportando algunas modificaciones.

La sinopsis de la sentencia INSERT es la siguiente:

```
[ WITH [ RECURSIVE ] consulta_cte [, ...] ]
INSERT INTO tabla
   [ ( columna [, ...] ) ]
   {
   DEFAULT VALUES |
        VALUES ( { expresión | DEFAULT } [, ...] ) |
        consulta
         }
   [ ON CONFLICT [ ON CONSTRAINT restricción ]
     DO NOTHING | DO UPDATE SET { columna = expresión } [, ...]
[ WHERE condición ] ]
   [ RETURNING * | expr ]
```

Según el nombre de la tabla, la lista de las columnas permite indicar aquellas que se utilizarán. Cuando todas las columnas de la tabla se usan, esta lista no es obligatoria.

Para cada columna, el valor se expresa con su valor literal o con una expresión, respetando la sentencia de las columnas indicadas en la primera parte del comando.

El siguiente ejemplo muestra una inserción simple:

```
INSERT INTO clientes (cl_nombre, cl_direccion) VALUES ('S.T.E.R.E.G',
'Islas Pitiusas 2, 25290 Las Rozas');
```

En este caso, las columnas que no se utilizan tomarán el valor por defecto identificado en la definición de la tabla.

También es posible no utilizar los nombres de las columnas, como en el siguiente ejemplo:

```
INSERT INTO prestaciones VALUES ( 1 , 'nombre' ,
,  '' , '13/06/2006', '12/06/2006', true ,'');
```

En este caso, se expresan los valores de todas las columnas. Entonces es posible indicar la palabra clave DEFAULT para utilizar el valor por defecto identificado en la definición de la tabla.

Es posible la utilización de subconsultas CTE, como en el caso de una consulta SELECT, y estas consultas se utilizan en la sentencia INSERT.

También es posible utilizar el resultado de una consulta SELECT para alimentar una sentencia INSERT. Es suficiente con hacer corresponder la lista de las columnas de la consulta SELECT con las utilizadas en la sentencia INSERT.

También es posible insertar varios registros con una única sentencia INSERT, como se muestra en el siguiente ejemplo:

```
INSERT INTO prestaciones VALUES
( 2 , 'nombre 2', '','27/06/2007','26/06/2007', true, "),
( 3 , 'nombre 3', '','27/07/2007','26/07/2007', true, ");
```

La sentencia RETURNING permite devolver uno o varios valores en expresiones; por ejemplo, un identificador generado por una secuencia o un valor por defecto generado por una función.

b. Administración de conflictos

A partir de la versión 9.5 de PostgreSQL, es posible interceptar un conflicto de inserción y hacer una actualización de datos en su lugar. Se trata de una funcionalidad similar a la sentencia MERGE, que existe en otros sistemas de bases de datos.

Un conflicto es la violación de una restricción existente sobre la tabla en la que se inserta. Aquí se trata de actualizar una regla que permite hacer un UPDATE o por defecto no hacer nada, con el fin de que no se genere un error.

La siguiente consulta muestra la sintaxis que hay que utilizar. Por ejemplo, durante la inserción de una factura, cuando esta ya existe, para que no haya error si no que se actualice la fecha y el medio de pago, se intercepta el conflicto sobre la columna de clave primaria y se hace el UPDATE, comprobando a través de la sentencia WHERE que los datos son diferentes:

```
insert into facturas as f (fact_num, fact_fecha, fact_fecha_pago,
fact_medio_pago, cl_nombre)
values ('02212S' ,'1993-01-04 00:00:00+01','1993-02-08
09:01:05.340623+01','C','Empresa Particular')
ON CONFLICT(fact_num)
DO UPDATE SET fact_fecha_pago = EXCLUDED.fact_fecha_pago,
fact_medio_pago = EXCLUDED.fact_medio_pago
WHERE f.fact_fecha_pago != EXCLUDED.fact_fecha_pago or
f.fact_medio_pago != EXCLUDED.fact_medio_pago;
```

c. La sentencia COPY

La sentencia COPY permite transferir los datos desde los archivos a las tablas de la base de datos. Esta sentencia permite una inserción rápida de los datos en la base y es interesante para grandes volúmenes de datos.

```
COPY nombretabla [ (columna [, ...]) ]
   FROM { 'nombrearchivo' |PROGRAM 'comando' | STDIN }
   [ [ WITH ]
       [ FORMAT texto | csv | binary ]
       [ OIDS booleano ]
       [ DELIMITER 'delimitador' ]
       [ NULL 'cadena NULA'  ]
       [ HEADER ]
       [ QUOTE 'comilla' ]
       [ ESCAPE 'escape' ]
       [ FORCE NOT NULL columna [, ...] ]
       [ FORCE_NULL ( columna [, ...] ) ]
       [ ENCODING 'codificación' ]
```

Los datos de la entrada pueden provenir de un archivo, de la entrada estándar o de la salida de un programa. El formato de los datos puede ser un archivo text, un archivo CSV o un archivo binario.

En el caso de archivos text o CSV, cada línea se corresponde con una tupla de la tabla donde los campos se separan por un carácter dedicado (coma, punto y coma, tabulación). El archivo debe ser accesible desde el sistema de archivos del servidor PostgreSQL.

El formato binario se corresponde con una copia bit a bit de los datos de las tablas, y no de su representación textual.

La utilización más sencilla se corresponde con el siguiente ejemplo:

```
COPY contactos FROM '/tmp/contactos.csv';
```

El archivo `contactos.csv` es equivalente a:

```
Dupont Martin   \N   martin@dupont.es  \N     Dupont and Co
Dupond Martin   \N   martin@dupond.es  \N     Dupond SARL
```

En este caso, el archivo debe utilizar el carácter `[tab]` para separar los campos. La opción `DELIMITER` permite modificar este carácter. Cuando se indica la opción `CSV`, el separador de campos es la coma.

También es posible indicar el carácter que enmarca una columna con la opción `QUOTE`, que se corresponde por defecto con las comillas dobles ("). La opción `ESCAPE` indica en este caso el carácter que permite escapar las comillas dobles en el contenido de los campos, para que el delimitador seleccionado no se interprete dentro de la cadena de caracteres.

La opción `NULL` indica el valor de los campos correspondientes al valor `NULL` en la tabla. Por defecto, este carácter `NULL` se corresponde con la cadena `\N` o una cadena vacía en modo `CSV`.

La opción `HEADER` indica que el archivo tiene una primera línea de encabezado, que se debe ignorar.

La opción `FORCE_NOT_NULL` permite indicar las columnas que forzosamente serán no nulas. Si no hay datos, se transformará en una cadena de caracteres vacía.

La opción `ENCODING` permite indicar el juego de caracteres utilizado para los datos.

El siguiente ejemplo muestra la carga de los datos desde un archivo `CSV`, con los encabezados y los campos delimitados por comillas dobles y separadas por comas:

```
COPY contactos FROM '/tmp/contactos-2.csv' WITH DELIMITER AS ',' CSV
HEADER QUOTE AS '"';
```

El archivo `contactos-2.csv` es equivalente a:

```
"Dupont","Martin",,"martin@dupont.es",,"Dupont and Co"
"Dupond","Martin",,"martin@dupond.es",,"Dupond SARL"
```

También es posible cargar los datos desde una fuente diferente a un archivo. En efecto, los datos se pueden cargar desde la entrada estándar, como en el siguiente ejemplo:

```
COPY contactos (ct_nombre, ct_apellidos, ct_telefono,
ct_email, ct_funcion, cl_nombre) FROM stdin;
Dupont Martin   \N   martin@dupont.es   \N   Dupont and Co
Dupond Martin   \N   martin@dupond.es   \N   Dupond SARL
\.
```

Esta técnica la utiliza principalmente la herramienta `pg_dump` en las copias de seguridad para los datos de las tablas.

La opción `OID` solo es útil para conservar los OID de las tuplas. Por defecto, no hay OID para las tuplas, lo que limita los casos de uso de esta opción.

2. Lectura de datos

a. La sentencia SELECT

La lectura de los datos es una de las etapas más importantes de las bases de datos: los datos almacenados generalmente se dedican a ser explotados y es la sentencia `SELECT` la que va a permitir su lectura.

Una sentencia `SELECT` sencilla distingue varias partes:

- La primera parte lista los campos que serán visibles en la respuesta. Pueden utilizar una expresión y tener otro nombre diferente a la columna de la tabla, modificándolo con la palabra clave `AS`. Es posible visualizar todos los campos utilizando el carácter * en lugar de la lista de los campos.
- La segunda parte lista las tablas utilizadas.
- Para terminar, un conjunto de palabras clave permite filtrar los datos según numerosos criterios:
  - La sentencia `WHERE` permite filtrar sobre las condiciones, comparando los valores entre ellos, por ejemplo.
  - La sentencia `GROUP BY` permite reagrupar los registros sobre sus valores y la sentencia `HAVING` permite filtrar estas agrupaciones.
  - Las palabras clave `UNION`, `INTERSECT` y `EXCEPT` permiten combinar varias expresiones `SELECT`.
  - La sentencia `ORDER BY` permite ordenar los registros de la salida según la expresión indicada.
  - La sentencia `LIMIT` indica el número de registros de la salida y la sentencia `OFFSET` permite retrasar esta restricción. Por ejemplo, la sentencia `LIMIT` con un valor de 10 y un retraso (`OFFSET`) de 0 mostrará los 10 primeros registros. La misma consulta con un retraso de 10 mostrará los 10 registros siguientes.
  - Las cláusulas `FOR UPDATE` y `FOR SHARE` sirven para bloquear los registros de las tablas, que permiten una actualización posterior.

La siguiente sinopsis muestra la sentencia `SELECT`:

```
[ WITH [ RECURSIVE ] consulta_cte [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expresión [, ...] ) ] ]
    * | expresión [ AS visualización ] [, ...]
    [ FROM from [, ...] [TABLESAMPLE método (ratio)] ]
    [ WHERE condición ]
    [ GROUP BY expresión [, ...] ]
    [ HAVING condición [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expresión [ ASC | DESC | USING operador ]
[, ...] ]
    [ LIMIT { número | ALL } ]
    [ OFFSET inicio ]
    [ FOR { UPDATE | SHARE } [ OF tabla [, ...] ] [ NOWAIT ] ]
```

La sentencia FROM permite asociar varias tablas a una misma consulta para cruzar la información entre estas tablas y, por lo tanto, extraer información interesante. A partir de la versión 9.5 de PostgreSQL, es posible utilizar solo una muestra de una tabla con la sentencia TABLESAMPLE usando un método de muestreo SYSTEM, más rápido, o BERNOUILLI, más justo, para un ratio dado. Esta funcionalidad permite las consultas mucho más rápidas devolviendo una aproximación del resultado.

La técnica más sencilla es listar las tablas simplemente separadas por una coma y utilizar la sentencia WHERE para asociar los campos sobre sus valores.

Por ejemplo, la siguiente consulta asocia la tabla clientes con la tabla contactos, probando la igualdad del campo cl_nombre correspondiente al nombre del cliente:

```
SELECT * FROM clientes, contactos WHERE contactos.cl_nombre =
clientes.cl_nombre;
```

Es posible ampliar este principio utilizando más tablas y más filtros combinados con los operadores lógicos AND u OR, como en el siguiente ejemplo:

```
SELECT prestaciones.prest_id, facturas.fct_num,
prestaciones.prest_fecha_inicio
   FROM prestaciones, registros_facturas, facturas
   WHERE
       prestaciones.prest_id = registros_facturas.prest_id
   AND
       registros_facturas.fct_num = facturas.fct_num
   ORDER BY  prestaciones.prest_id;
```

Además, las columnas se seleccionan utilizando el nombre de la tabla y el resultado ordenado según la sentencia de un identificador.

Las subconsultas

El principio de la consulta expuesta de esta manera es extensible porque una consulta SELECT puede llamar a otras consultas SELECT.

Principalmente existen dos métodos de encapsulación de una consulta en otra: la primera devuelve un atributo a la lista de los atributos devueltos por la consulta de nivel superior. Cuando se devuelven varios atributos por la subconsulta, la consulta de primer nivel ve este conjunto de atributos como una tupla de un tipo compuesto.

El siguiente ejemplo pregunta el catálogo de PostgreSQL para listar las relaciones. La subconsulta devuelve el esquema al que pertenece cada tabla:

```
SELECT
 (SELECT nspname FROM pg_namespace n
    WHERE nspname not in ('pg_catalog', 'información_esquema')
      and n.oid=c.relnamespace) AS esquema,
 relname AS tabla
 FROM pg_class c
 WHERE c.relkind='r';
```

La segunda permite utilizar la subconsulta como una relación, utilizándose la subconsulta en la sentencia FROM y, por lo tanto, haciéndose joins como en el siguiente ejemplo:

```
SELECT
 n.esquema,
 c.relname AS tabla
 FROM pg_class c
   JOIN
     (SELECT oid, nspname AS esquema
        FROM pg_namespace
        WHERE nspname NOT IN
              ('pg_catalog', 'información_esquema')
      ) AS n
   on c.relnamespace = n.oid
 WHERE c.relkind='r';
```

Los dos ejemplos anteriores son equivalentes en términos de resultado, pero PostgreSQL los puede interpretar de manera diferente, en particular cuando las subconsultas son más complejas.

Los joins

Disponemos de otra técnica para leer las tablas dos a dos: los joins. Esta segunda técnica permite obtener el mismo resultado, pero la escritura puede ser más legible.

Existen dos tipos de joins, los joins internos (`INNER JOIN`) y los joins externos (`OUTER JOIN`). Entre los joins internos, el join más sencillo es el join natural, que permite leer dos tablas utilizando simplemente como enlace entre ellas los atributos que tienen el mismo nombre en cada una de las tablas:

```
SELECT * FROM clientes NATURAL JOIN contactos;
```

Las dos consultas siguientes producen exactamente el mismo resultado con las palabras clave `USING` u `ON`, pero utilizando técnicas diferentes:

```
SELECT * FROM clientes JOIN contactos USING (cl_nombre);
SELECT * FROM clientes JOIN contactos ON clientes.cl_nombre =
contactos.cl_nombre;
```

Estas consultas de joins son idénticas a las consultas que utilizan la sentencia `WHERE` para realizar los joins.

El join cruzado es otro join interno que realiza un producto cartesiano, pero sin eliminar ningún registro:

```
SELECT * FROM clientes CROSS JOIN contactos;
```

Es equivalente a:

```
SELECT * FROM clientes, contactos;
```

El otro tipo de join, el join externo, añade al join interno los datos de la tabla que no validan la clave join. El join externo puede afectar a la tabla de la izquierda (`LEFT`), de la derecha (`RIGHT`) o a las dos (`FULL`). Los atributos que vienen de la otra tabla son nulos.

Por ejemplo, si un cliente se registra en la tabla `clientes`, pero todavía no ha solicitado prestaciones:

- el join interno de las dos tablas no lo visualizará,
- el join externo lo añadirá al resultado, como en el siguiente ejemplo:

```
SELECT * FROM clientes LEFT OUTER JOIN prestaciones ON
clientes.cl_nombre = prestaciones.cl_nombre;
```

Aquí, el join extrae los datos no asociados a la tabla de la izquierda. Es posible buscar estos datos en la tabla de la derecha con la palabra clave `RIGHT` y en las dos tablas con la palabra clave `FULL`.

Antes de la versión 9.3 de PostgreSQL, durante la utilización de subconsultas como miembro de un join, no era posible hacer referencia al otro miembro del join. La palabra clave `LATERAL` permite hacer esto, mejorando el rendimiento. En efecto, este tipo de problemas se resolvía con complejas consultas que no permitían la optimización.

El siguiente ejemplo muestra la utilización de un atributo del otro miembro del join, gracias a la palabra clave `LATERAL`. En esta consulta, la lista de los OID de los esquemas elegidos se utiliza directamente en la subconsulta, seleccionando las relaciones. De esta manera, PostgreSQL puede realizar una optimización en el join:

```
SELECT esquema,
 table_name
 FROM
   (SELECT oid, nspname AS esquema
        FROM pg_namespace
        WHERE nspname NOT IN
              ('pg_catalog', 'información_esquema')
      ) AS n,
   lateral (SELECT
      relnamespace, relname AS table_name
      FROM pg_class
      WHERE relnamespace = n.oid AND relkind='r') AS c;
```

Las agrupaciones

El método de agrupación `GROUP BY` se implementó en PostgreSQL siguiendo el estándar SQL. Desde la versión 9.5 de PostgreSQL, se puede hacer la agrupación utilizando los `GROUPING SET`, lo que permite recoger en una única consulta el equivalente de varios `GROUP BY` diferentes, como durante la concatenación de varias consultas con `GROUP BY` con un UNION. Existen tres clausulas que permiten este tipo de agrupación:

- `GROUPING SETS ( (attr1, attr2) [ , ( attr1 ), () ...] )`: agrupa sobre todos los conjuntos de atributos específicos.
- `ROLLUP ( attr1, attr2, ... )`: agrupa sobre todas las combinaciones de atributos de la lista.
- `CUBE ( attr1, attr2, ... )`: agrupa sobre todas las combinaciones que utilizan los atributos en la sentencia de la lista.

El siguiente ejemplo agrupa las prestaciones por año, después por cliente y sobre el hecho de que la prestación esté conforme o no, utilizando todas las combinaciones de estos tres criterios:

```
select count(*), cl_nombre, prest_confirm, extract( year from
prest_fecha_inicio)
   from prestaciones
  group by  rollup ( (cl_nombre), (prest_confirm), (extract( year
from prest_fecha_inicio)) );
```

b. La sentencia COPY

La sentencia COPY permite generar un archivo CSV desde una tabla de la base de datos. Las opciones son las mismas que durante la lectura de los datos:

```
COPY { nombretabla [ (columna [, ...]) ] | (consulta) }
   TO { 'nombrearchivo' |PROGRAM 'comando' | STDIN }
   [ [ WITH ]
       [ FORMAT text | csv | binary ]
       [ OIDS booleano ]
       [ DELIMITER 'delimitador' ]
       [ NULL 'cadena NULA'  ]
       [ HEADER ]
       [ QUOTE 'comilla' ]
       [ ESCAPE 'escape' ]
       [ FORCE_QUOTE { ( columna [, ...] ) | * } ]
       [ ENCODING 'codifica' ]
```

Las opciones son similares al comando COPY FROM. La opción FORCE_QUOTE permite forzar el enmarcado de los valores no nulos por el carácter QUOTE, para las columnas designadas en la lista, cuando se utiliza el formato CSV.

Por ejemplo, el siguiente comando COPY permite generar un archivo CSV:

```
COPY contactos TO '/tmp/contactos.csv' WITH DELIMITER AS ','
CSV HEADER QUOTE AS '"';
```

El siguiente comando es equivalente, pero utiliza una consulta select:

```
COPY ( SELECT * FROM contactos ) TO '/tmp/contactos.csv' WITH
DELIMITER AS ',' CSV HEADER QUOTE AS '"';
```

Desde la versión 9.6 de PostgreSQL, es posible utilizar las sentencias INSERT, UPDATE o DELETE con la sentencia RETURNING como consulta de la sentencia COPY.

3. Actualización de los datos

La sentencia UPDATE

La sentencia de actualización UPDATE permite modificar el contenido de una tabla en función de diferentes datos.

La sinopsis del comando es la siguiente:

```
[ WITH [ RECURSIVE ] consulta_cte [, ...] ]
UPDATE [ ONLY ] table
   SET
       columna = { expresión | DEFAULT }
       [, ...]
   [ FROM lista ]
   [ WHERE condición | WHERE CURRENT OF cursor ]]
   [ RETURNING * | expr ]
```

La actualización se realiza generalmente sobre una tabla, utilizando un filtro en la sentencia WHERE. En efecto: si no se utiliza ningún filtro, se modifican todos los registros de las columnas actualizadas.

Esta sentencia WHERE también puede utilizar un cursor declarado inicialmente para identificar los registros que se deben actualizar.

El siguiente ejemplo muestra la actualización del campo cl_direccion de la tabla clientes en función del nombre de este cliente:

```
UPDATE clientes SET cl_direccion = '' WHERE cl_nombre = '';
```

La sentencia ONLY permite limitar la actualización a la tabla actual y no a las tablas heredadas.

La palabra clave FROM permite añadir las tablas utilizadas en la sentencia WHERE para filtrar los datos.

Otra notación puede simplificar la escritura de las actualizaciones, como en el siguiente ejemplo:

```
UPDATE clientes SET ( cl_direccion , cl_ciudad , cl_codigopostal )
= ( ' Islas Pitiusas ' , ' 28290 ', ' Las Rozas ' )
WHERE cl_nombre = 'Cualquier cosa' ;
```

Esta notación es más cercana a la sentencia INSERT TO.

La sentencia RETURNING permite devolver uno o varios valores o expresiones actualizados, lo que es particularmente útil cuando los valores se generan por defecto o por una expresión.

4. Eliminación de datos

a. La sentencia DELETE

La eliminación de los datos permite suprimir registros de una tabla en función de los filtros utilizados en la sentencia WHERE.

La sinopsis del comando es la siguiente:

```
[ WITH [ RECURSIVE ] consulta [, ...] ]
DELETE FROM [ ONLY ] tabla
   [ USING list_using ]
   [ WHERE condición ] | WHERE CURRENT OF cursor ]
   [ RETURNING * | expr [ [ AS ] nombre ] [, ...] ]
```

Por ejemplo, el siguiente comando permite eliminar un contacto en función de su nombre y su apellido:

```
DELETE FROM contactos WHERE ct_nombre = '' AND ct_apellidos = '';
```

De la misma manera que en el comando UPDATE, es posible utilizar otras tablas para el filtro, pero con la palabra clave USING.

Los registros que hay que eliminar se seleccionan por la sentencia WHERE, que también puede utilizar un cursor.

La sentencia RETURNING permite devolver uno o varios valores o expresiones modificadas, lo que es particularmente útil cuando los valores se generan por defecto o por una expresión.

b. La sentencia TRUNCATE

La sentencia `TRUNCATE` también permite eliminar los datos de una tabla, pero no permite seleccionar los registros: todo el contenido de la tabla se elimina.

De esta manera es posible vaciar varias tablas al mismo tiempo, como en la siguiente sinopsis:

```
TRUNCATE [ TABLE ] nombre [, ...]
```

5. Los CTE y la palabra clave WITH

Los CTE (*Common Table Expression*) permiten encapsular las consultas unas con otras. Estas consultas previas a la consulta principal se pueden ver como tablas temporales a las que las siguientes consultas pueden hacer referencia.

Las consultas pueden ser `SELECT`, `INSERT`, `UPDATE` o `DELETE`, tanto en las consultas auxiliares como en la consulta principal.

La estructura de este tipo de comandos es la siguiente:

```
WITH nombre1 AS (
consulta_aux1
) [ ,
nombre2 AS (
consulta_aux2
), ... ]
consulta_principal
```

En esta estructura, la consulta `consulta_aux2` puede hacer referencia a `consulta_aux1`. La consulta principal puede hacer referencia a `consulta_aux2` y `consulta_aux1`.

El siguiente ejemplo permite desplazar los registros de la tabla `prestaciones` a una tabla `prestaciones_2012` para poder archivarlos:

```
WITH del AS (
 DELETE FROM prestaciones
 WHERE prest_fecha_inicio > = '2012-01-01 00:00:00'
   AND prest_fecha_fin < '2013-01-01 00:00:00'
 RETURNING *
),
WITH ins AS (
 INSERT into prestaciones_2012 SELECT * FROM del;
)
SELECT count(*) FROM del;
```

La consulta principal solo cuenta el número de registros eliminados en la primera consulta auxiliar y la segunda consulta auxiliar utiliza el resultado de esta primera consulta gracias a la sentencia `RETURNING`.

Esta consulta se ejecuta en una única transacción de manera atómica, lo que hace que en casos de error, por ejemplo la inserción de un registro en la tabla `prestaciones_2012` como consecuencia de una restricción, se anule la eliminación en la tabla `prestaciones`.

La palabra clave `RECURSIVE` permite a una consulta auxiliar utilizar su propio resultado y, por lo tanto, ser recursiva. Una consulta auxiliar recursiva está compuesta de un primer elemento no recursivo, asociado con la palabra clave `UNION` a un elemento recursivo. El elemento recursivo debe poder no devolver nada, condición que hace posible la continuación del tratamiento.

La siguiente consulta ilustra esta recursividad utilizando la tabla `formaciones`. El atributo `sup_id` se refiere a un identificador de la misma tabla y permite asociar una formación a su padre, indicando qué formación precede a la formación deseada:

```
WITH RECURSIVE ind_f(sup_id, id, titulo) AS (
   SELECT sup_id, id, titulo FROM formaciones WHERE titulo = 'Título'
 UNION ALL
   SELECT f.sup_id, f.id, f.titulo
   FROM formaciones f, ind_f
   WHERE f.id = ind_f.sup_id
 )
SELECT * FROM ind_f;
```

El primer miembro de la consulta recursiva filtra la formación buscada por el título y el segundo miembro realiza un join entre la tabla `formacion` y la consulta recursiva usando como clave de join los atributos `id` y`sup_id`.

6. Las transacciones

PostgreSQL es un servidor de bases de datos transaccional. Para esto, respeta las propiedades ACID:

- Atomicidad: una transacción forma un conjunto atómico. El conjunto de la transacción se valida o no se valida en absoluto.
- Coherencia: una transacción no puede hacer que la base de datos quede incoherente.
- Aislamiento o *Isolation*: una transacción no ve el resto de transacciones actuales.
- Durabilidad: los datos validados lo son de manera efectiva.

Estas diferentes propiedades se implementan por el sistema MVCC: MultiVersion Concurrency Control. Este mecanismo permite aislar la sesión actual de las modificaciones subyacentes, que podrían actualizar los datos a un estado incoherente.

En todo momento, el sistema solo ve los datos coherentes, evitando hacer bloqueos; bloqueos que podrían impedir las lecturas y escrituras.

En PostgreSQL, cada consulta se hace en una transacción. Por defecto, las sesiones utilizan un modo de validación automático, llamado `autocommit`, que hace que cada consulta se valide inmediatamente.

No se puede desactivar este modo automático, pero es posible iniciar explícitamente una transacción con la sentencia `BEGIN`.

Esta sentencia pone fin a la validación automática y el sistema espera una validación con `COMMIT` o una anulación con `ROLLBACK` para terminar la transacción.

a. Niveles de aislamiento

PostgreSQL permite dos niveles de aislamiento: **Read Committed** y **Serializable**.

El nivel **Read Committed** o modo de lectura validada es el nivel de aislamiento por defecto. En este modo, una consulta hecha en la transacción ve los datos tal y como estaban antes del inicio de esta. Además, no ve las modificaciones realizadas en otra transacción mientras esta transacción no se valide. Por el contrario, se ve la modificación realizada en la misma transacción, incluso cuando la transacción no se ha validado.

El nivel **Serializable** es más estricto: en una transacción, una consulta no ve las modificaciones validadas por otras transacciones. Por lo tanto, el contenido de una transacción se aísla del resto de las transacciones mientras se desarrolla la transacción.

El siguiente ejemplo muestra la utilización de una transacción para insertar una nueva línea en la tabla de prestaciones.

Se desarrollan dos transacciones al mismo tiempo y la ejecución de las sentencias `SELECT` e `INSERT` muestran el aislamiento de los datos:

```
> SHOW transaccion_aislamiento
transaccion_aislamiento
_____________
read committed
> BEGIN;
> SELECT prest_id FROM
prestaciones;
prest_id
_____
(0 registros)

> INSERT INTO prestaciones ( prest_id,
prest_nombre, prest_fecha_inicio )
VALUES ( 1, 'Nueva Prestación',
'27/09/2006' );
>
> SELECT prest_id FROM prestaciones;
prest_id
_____
       1
(1 registro)
>
>
>
>
>
>
>
>
> COMMIT;
```

```
> SHOW transaccion_aislamiento;
transaccion_aislamiento
_____________
read committed
>
>
> BEGIN;
>
>
>
>
>
>
> SELECT prest_id FROM prestaciones;
prest_id
_____
(0 registros)

> INSERT INTO prestaciones ( prest_id,
prest_nombre, prest_fecha_inicio )
VALUES ( 2, 'Segunda Prestación',
'27/09/2006');
>
> SELECT prest_id FROM prestaciones;
prest_id
_____
       2
(1 registro)

>
>
> SELECT prest_id FROM prestaciones;
prest_id
_____
       1
       2
(2 registros)

> COMMIT;
```

La segunda transacción, en la columna de la derecha, no permite ver los datos de la primera transacción, salvo cuando está validada. El segundo nivel de aislamiento, **Serializable**, no permite ver los datos de la primera transacción, incluso si se valida. Para comprobar este comportamiento, hay que cambiar el nivel de aislamiento.

El cambio de nivel del aislamiento se realiza con dos comandos:

El primer comando modifica el nivel de aislamiento de la transacción actual. Por lo tanto, hay que utilizarlo después del comando `BEGIN`:

```
SET TRANSACTION ISOLATION LEVEL  { SERIALIZABLE | READ COMMITTED }
```

El segundo comando modifica el nivel de aislamiento para la sesión actual:

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL
{ SERIALIZABLE | READ COMMITTED }
```

Una vez que se realiza el cambio en las dos sesiones, comprobamos que la segunda transacción no ve los datos validados fuera ni durante esta misma transacción.

b. Puntos de copia de seguridad

Existe un mecanismo de subtransacciones en PostgreSQL, que permite anular una parte de una transacción volviendo a un punto anterior de ella con los comandos `SAVEPOINT` y `ROLLBACK to SAVEPOINT`.

El comando `SAVEPOINT` se llama en una transacción abierta con `BEGIN` y recibe como único argumento un nombre, que sirve de punto de registro:

```
> SAVEPOINT punto1;
```

Después, si es necesario, más adelante en la misma transacción, es posible volver a este punto anulando todo lo que se ha hecho durante este tiempo con el siguiente comando:

```
> ROLLBACK TO SAVEPOINT punto1;
```

Es posible arrancar varias `SAVEPOINT` y volver a cualquier `SAVEPOINT` de la misma transacción.

c. Locks de bloqueo

PostgreSQL dispone de un mecanismo de detección de locks de bloqueos. Este tipo de bloqueos aparecen cuando dos transacciones concurrentes están esperando la una a la otra, es decir, que una transacción espera un bloqueo detenido por otra transacción, y esta espera un bloqueo detenido por la primera.

En este caso, pasado un plazo definido por el argumento `deadlocks_timeout`, ajustado a 1 segundo por defecto, PostgreSQL anulará una transacción, la que se estime menos problemática, algunas veces incluso las dos transacciones.

La detección de este tipo de bloqueos y la anulación de las transacciones se reporta en los archivos de trazas de actividad.

Para no llegar a este caso, hay que tener cuidado con la sentencia del tratamiento en las transacciones. Esto se produce cuando varios tratamientos que actúan sobre las mismas tablas se programan en sentencias diferentes.

Programación

Introducción

La programación del lado servidor consiste principalmente en crear procedimientos almacenados en el servidor utilizando un administrador de procedimientos. PostgreSQL no gestiona los lenguajes de procedimientos, sino que delega esta tarea en un administrador dedicado, lo que permite implementar numerosos lenguajes de programación como lenguaje de procedimientos almacenados. Por ejemplo, es el caso de `PL/Perl` o `PL/Python`, que se basa en un intérprete externo a PostgreSQL, mientras que el lenguaje `PL/pgSQL` se entrega con PostgreSQL.

Procedimientos almacenados

Los lenguajes de procedimientos almacenados no están disponibles por defecto en una base de datos. Se deben instalar en cada base de datos con el comando `CREATE LANGUAGE` o con el comando de sistema equivalente `createlang`. Es posible instalar un lenguaje en una base de datos modelo como `template1` para hacer que esté disponible automáticamente en las bases de datos creadas a partir de este modelo.

1. SQL

Una función del tipo SQL puede ejecutar una lista de consultas SQL.

En este tipo de función no es posible utilizar comandos de control de transacciones como `SAVEPOINT` o `COMMIT`, ni comandos de herramientas como `VACUUM`, que no se pueden ejecutar dentro de una transacción.

Se pueden ejecutar sentencias DML como `SELECT`, `INSERT`, `UPDATE`, `DELETE` y las sentencias DDL como `CREATE`, `ALTER` o `DROP`. El cuerpo de la función puede estar enmarcado por comillas simples o por el símbolo del dólar doble, particularmente útiles cuando las consultas contienen comillas simples.

Solo se pueden devolver los datos de la última sentencia SQL. Por defecto, en una función que solo devuelve una única tupla, solo se devolverá la primera línea del último registro, sea cual sea la ordenación aplicada en esta consulta. También es posible para una función devolver un conjunto de registros.

Los argumentos utilizados como entrada de la función se pueden utilizar de dos maneras. Cuando no tienen nombre, su índice permite usarlos directamente, como en el siguiente ejemplo:

```
CREATE FUNCTION addition(integer, integer)
RETURNS integer
LANGUAGE SQL
AS $$
   SELECT $1 + $2;
$$;
```

Esta función simplemente suma los dos números enteros que se pasan como argumento. Esta misma función se puede escribir utilizando argumentos con nombre, como en el siguiente ejemplo:

```
CREATE FUNCTION addition( m integer, n integer)
RETURNS integer
LANGUAGE SQL
AS $$
   SELECT m + n;
$$;
```

La elección de los nombres de los argumentos es importante porque pueden existir conflictos con los nombres de los atributos; los atributos son elegidos prioritariamente. Se puede utilizar como prefijo el nombre del argumento junto al nombre de la función, por ejemplo `addition.m`.

Es posible devolver un conjunto de valores, equivalente a una tupla, de dos maneras:

- Declarando un tipo compuesto utilizado como tipo de retorno, como en el siguiente ejemplo:

```
CREATE TYPE completedate
 as ( now timestamptz, dow double precision,
      dia texto,  mes texto, week double precision);
CREATE OR REPLACE FUNCTION getdate()
 RETURNS completedate
 LANGUAGE SQL
AS $$
SELECT now(), fecha_part('dow', now() ) AS dow,
      to_char(now(), 'TMday') AS dia,
      to_char( now(), 'TMmonth') AS mes ,
      fecha_part('week', now()) AS week;
$$;
```

- Declarando los argumentos de salida con la palabra clave `OUT`:

```
CREATE OR REPLACE FUNCTION getdate2
 ( OUT now timestamptz, OUT dow double precision,
   OUT dia texto, OUT mes texto, OUT week double precision )
LANGUAGE SQL
AS $$
SELECT now(), fecha_part('dow', now() ) AS dow,
      to_char(now(), 'TMday') AS dia,
      to_char( now(), 'TMmonth') AS mes ,fecha_part('week',
now()) AS week;
$$;
```

Dando por hecho que un tipo se crea implícitamente durante la creación de una tabla, este tipo se puede utilizar como tipo de retorno de una función.

Una función SQL también puede devolver un conjunto de tuplas creando de esta manera una relación. Por ejemplo, la siguiente función permite generar una lista de fechas:

```
CREATE OR REPLACE FUNCTION getdates(inicio integer, intervalo integer)
RETURNS setof completedate
LANGUAGE SQL
AS $$
SELECT x.x, fecha_part('dow', x.x ) AS dow,
      to_char(x.x, 'TMday') AS dia,
      to_char( x.x, 'TMmonth') AS mes,
      fecha_part('week', x.x )AS week
FROM generate_series( now() - (inicio * '1d'::interval),
                      now(),
                      ( intervalo * '1d'::interval) )
AS x ORDER BY 1 desc;
$$;
```

En la sentencia `FROM` de una `select`, se puede llamar a la función como si se tratara de una tabla:

```
> SELECT * FROM getdates(2, 1);
             now               | dow |   dia   |   mes     | week
-------------------------------+-----+---------+-----------+----
2014-11-10 12:05:20.874563+01  |   1 | lunes   | noviembre |  46
2014-11-09 12:05:20.874563+01  |   0 | domingo | noviembre |  45
2014-11-08 12:05:20.874563+01  |   6 | sábado  | noviembre |  45
(3 rows)
```

Los argumentos permiten controlar la fecha, el número de registros devueltos y el intervalo deseado entre dos fechas.

La siguiente sintaxis permite dar valores por defecto a estos argumentos:

```
CREATE OR REPLACE FUNCTION getdates
( inicio integer DEFAULT 10,
 intervalo integer DEFAULT 1)
RETURNS setof completedate
LANGUAGE SQL
AS $$
SELECT x.x, fecha_part('dow', x.x ) AS dow,
      to_char(x.x, 'TMday') AS dia,
      to_char( x.x, 'TMmonth') AS mes,
      fecha_part('week', x.x )AS week
FROM generate_series( now() - (inicio * '1d'::interval),
                     now(),
                     ( intervalo * '1d'::interval) )
AS x ORDER BY 1 DESC;
```

De tal manera que la función también se puede llamar sin valores:

```
> SELECT * FROM getdates(2, 1);
             now               | dow |   dia   |   mes     | week
-------------------------------+-----+---------+-----------+------
2014-11-10 12:05:20.874563+01  |   1 | lunes   | noviembre |   46
2014-11-09 12:05:20.874563+01  |   0 | domingo | noviembre |   45
2014-11-08 12:05:20.874563+01  |   6 | sábado  | noviembre |   45
[...]
(3 rows)
```

Para terminar, es posible utilizar una lista dinámica de valores como entrada con la palabra clave `VARIADIC`, que crea una tabla de valores manipulables en la función, como en el siguiente ejemplo:

```
CREATE OR REPLACE FUNCTION getdates
(VARIADIC fechas timestamptz[] )
RETURNS setof completedate
LANGUAGE SQL
AS $$
SELECT dates[x], fecha_part('dow', dates[x] ) AS dow,
      to_char( dates[x], 'TMday') AS dia,
      to_char( dates[x], 'TMmonth') AS mes,
      fecha_part('week', dates[x] )AS week
FROM generate_subscripts( dates, 1 ) x(x);
$$;
```

Por lo tanto, durante la llamada a la función es posible pasar un número variable de valores:

```
SELECT getdates( now(), '1978-10-27 00:00:00');
                      getdates
-----------------------------------------------------
("2014-11-10 14:25:26.259588+01",1,lunes,noviembre,46)
("1978-10-27 00:00:00+01",5,viernes,octubre,43)
(2 rows)
```

a. Volatilidad

Por defecto, las funciones se declaran como `VOLATILE`, es decir, que se espera un resultado diferente para cada llamada de la función. Se puede añadir una optimización en este punto. En efecto, una función declarada `IMMUTABLE` no modifica los datos y siempre devuelve el mismo resultado para los argumentos dados, de la misma manera que `2+2=4`.

Una función declarada como `STABLE` puede leer los datos de una tabla, pero siempre devolverá el mismo resultado para una lectura dada.

Por lo tanto, la siguiente función se puede declarar `IMMUTABLE`:

```
CREATE FUNCTION addition(integer, integer)
RETURNS integer
IMMUTABLE
LANGUAGE SQL
AS $$
   SELECT $1 + $2;
$$;
```

b. Costes de la llamada

Una función que devuelve varios registros puede tener un impacto sobre el rendimiento de la consulta en la que se llama. Debido a la encapsulación, no siempre es posible para el planificador de consultas evaluar este número de registros.

Es posible dar una estimación del número de registros o del coste, informando de esta manera al planificador de consultas con las opciones `ROWS` y `COST`:

```
CREATE OR REPLACE FUNCTION getdates
(VARIADIC dates timestamptz[] )
RETURNS setof completedate
LANGUAGE SQL
COST 100
ROWS 20
AS $$
SELECT dates[x], date_part('dow', dates[x] ) AS dow,
      to_char( dates[x], 'TMday') AS dia,
      to_char( dates[x], 'TMmonth') AS mes,
      date_part('week', dates[x] )AS week
FROM generate_subscripts( dates, 1 ) x(x);
$$;
```

2. PL/pgSQL

El lenguaje PL/pgSQL se entrega con PostgreSQL. Es suficiente con activarlo en una base de datos para poder utilizarlo, como en el siguiente comando:

```
CREATE LANGUAGE plpgsql;
```

o con el comando de sistema `createlang`:

```
[POSTGRES]$ CREATELANG plpgsql BASE
```

La ventaja del lenguaje PL/pgSQL es que es muy parecido al lenguaje SQL y, por lo tanto, es compatible con los tipos de datos, operadores y funciones del SQL. De hecho, un procedimiento en PL/pgSQL es portable a todas las instancias de PostgreSQL.

Es el caso del lenguaje PL/pgSQL, que se instala como una extensión en la base-modelo `template1` y, por lo tanto, queda automáticamente disponible por defecto en una nueva base de datos.

a. Estructura de una función

Una función PL/pgSQL se declara con el comando `CREATE FUNCTION` y la estructura es la siguiente:

```
CREATE or REPLACE FUNCTION nombrefunción (argumentos) RETURNS
type AS $$
   DECLARE
   declaración;
       -- bloque de declaración de las variables internas;
   BEGIN
       instrucción;
       -- bloque de instrucción;
   END
$$ LANGUAGE 'plpgsqel' ;
```

Por lo tanto, existen dos bloques: uno primero para las declaraciones y uno segundo para las instrucciones.

El tipo de retorno, es decir, la naturaleza de la respuesta de la función, puede ser de cualquier tipo de datos SQL. Las funciones PL/pgSQL también pueden devolver un tipo `record`, que es equivalente a un registro de una tabla. Una función también puede devolver un conjunto de tipo de datos, el equivalente de una tabla. El tipo `void` asimismo permite no devolver datos.

Por defecto, los datos de la entrada se declaran con los nombres `$1`, `$2`, `$n`, pero se les puede asignar un nombre en la declaración de la función.

b. Ejemplo de función

La siguiente función realiza un cálculo sencillo sobre un valor numérico:

```
CREATE OR REPLACE FUNCTION importe_total
   ( importe_brt numeric, iva numeric default 21 )
RETURNS numeric
LANGUAGE 'plpgsql'
STABLE
AS $$
BEGIN
   RETURN importe_brt * (1 + (iva / 100  ));
END;
$$;
```

La siguiente función utiliza una consulta `select` para devolver un valor:

```
CREATE OR REPLACE FUNCTION importe_factura
   (id_factura text)
RETURNS numeric
LANGUAGE 'plpgsql'
AS $$
DECLARE
 suma_total numeric;
BEGIN
 RAISE NOTICE 'ID Factura: % ' , id_factura;
 SELECT importe_total(sum (lf_cantidad)) into suma_total
   FROM registros_facturas
   WHERE fact_num = id_factura;
RAISE NOTICE 'Suma: % ' , suma_total;
RETURN suma_total;
END;
$$;
```

Las funciones se pueden utilizar en una consulta `select`, por ejemplo:

```
clientes=# select importe_total( 101.119 );
        importe_total
-----------------------------
121.949
(1 row)
clientes=# select fact_num, cl_nombre, importe_factura( fact_num )
from facturas order by random() limit 1;
NOTICE:  ID Factura: 03227S
NOTICE:  Suma: 9045.000000000000000000000
fact_num |        cl_nombre         |       importe_factura
---------+--------------------------+-----------------------------
03227S   | Empresa Particular       | 9045.000000000000000000000
(1 row)
```

Es posible controlar los errores declarando subtransacciones, gestionar bloques de excepciones utilizando códigos de error de PostgreSQL, así como generar errores con `RAISE ERROR`.

El lenguaje PL/pgSQL es un lenguaje completo, con estructuras de control, cuya descripción completa va más allá del objetivo de este libro. Los desarrolladores de procedimientos almacenados encontrarán más información en la página de documentación siguiente: https://www.postgresql.org/docs/9.4/plpgsql.html

Triggers

Un trigger es una función especial que se llama asociada a un evento `INSERT`, `UPDATE` o `DELETE` de una tabla. Independientemente del lenguaje de procedimientos almacenados que utilice para escribir estas funciones, se pueden implementar con el comando `CREATE TRIGGER`:

```
CREATE TRIGGER nombre
{ BEFORE | AFTER | INSTEAD OF }
{ INSERT | UPDATE [ OF columna [, ...]]
 | DELETE | TRUNCATE [ OR ... ] }
   ON tabla
   [ FOR [ EACH ] { ROW | STATEMENT } ]
   WHEN ( condición )
   EXECUTE PROCEDURE nombrefunc ( argumentos )
```

Un trigger se utiliza antes, después o en lugar de un evento y se puede usar sobre cada registro afectado o una única vez para la operación, con la opción `FOR EACH STATEMENT`.

Una función de trigger generalmente no tiene argumentos de entrada y devuelve un tipo de datos `trigger`.

1. Código PL/pgSQL

Cuando una función `trigger` se escribe en PL/pgSQL, algunas variables se definen implícitamente para conocer el contexto de utilización de la función. Las variables implícitas son las siguientes:

- `TG_WHEN`: momento del desencadenamiento: `AFTER` o `BEFORE`, es decir, antes o después de la sentencia SQL que desencadena la llamada a la función.
- `TG_OP`: operación `INSERT`, `UPDATE` o `DELETE`.
- `TG_TABLE_NAME`: nombre de la tabla sobre la que se desencadena la función.
- `TG_SCHEMA_NAME`: nombre del esquema de la tabla sobre la que se desencadena la función.
- `TG_NARGS`, `TG_ARGV`: número de argumentos y tabla de los argumentos de entrada de la función.

Durante la utilización de un trigger sobre los registros (`FOR EACH ROW`), los datos manipulados por la consulta son accesibles a través de las variables `old` y `new`. La variable `old` es accesible para las consultas `UPDATE` y `DELETE`, y la variable `new` para las consultas `UPDATE` e `INSERT`.

a. Ejemplo

El siguiente ejemplo permite crear una entrada en la tabla `clientes` cuando se inserta un registro en la tabla `prestaciones`. El cliente es obligatorio. Si no existe, se crea. La función trigger se llama antes de la inserción, de tal manera que, cuando se realiza la inserción, el cliente ya existe y por lo tanto la clave extranjera es válida. La función es la siguiente:

```
CREATE OR REPLACE FUNCTION trg_prest_create_clientes()
RETURNS TRIGGER
LANGUAGE PLPGSQL
AS $BODY$
begin
 IF TG_OP = 'INSERT' AND TG_TABLE_NAME = 'prestaciones'
 THEN
   perform 1 from clientes where cl_nombre = new.cl_nombre;
   if not found then -- no, entonces lo creamos
     INSERT INTO clientes ( cl_nombre ) VALUES ( new.cl_nombre );
   end if;
 END IF;
RETURN new;
end;
$BODY$;
```

La función devuelve la tupla `new`, que es invocada en una sentencia `INSERT`. Se utiliza la variable `found` para controlar el resultado de la consulta `perform`. La función es llamada por el trigger, creado con la siguiente sentencia:

```
CREATE TRIGGER prestaciones_create_clientes
BEFORE INSERT ON prestaciones
FOR EACH ROW
EXECUTE PROCEDURE trg_prest_create_clientes();
```

El momento del desencadenamiento se indica por las clausulas `BEFORE INSERT`. La sentencia `FOR EACH ROW` desencadena la función para cada registro de la sentencia `INSERT`. La siguiente sentencia `INSERT` desencadena silenciosamente la función y, por lo tanto, la creación o no del registro en la tabla `clientes`:

```
insert into prestaciones ( prest_nombre, cl_nombre) values
('Instalación de nuevo hardware' ,'SAGEFIR');
```

A continuación, podemos comprobar la presencia del cliente con el nombre utilizado en la tabla `clientes`.

Nota: este tipo de uso es un facilitador para la aplicación que utiliza la base de datos clientes. Pero ¿de verdad queremos insertar un nuevo cliente si simplemente se trata de un error de introducción de datos? La respuesta con toda probabilidad es negativa, y comprobamos que la utilización de un trigger de este tipo, aunque sea muy sencilla, no siempre es una buena idea a medio o largo plazo.

b. Eliminación de triggers

Es posible eliminar los triggers con el comando DROP TRIGGER:

```
DROP TRIGGER prestaciones_create_clientes ON prestaciones;
```

c. Trigger sobre un evento

Además de los triggers habituales, PostgreSQL tiene la posibilidad de desencadenar funciones sobre los eventos de tipo creación, modificación y eliminación de objetos, tales como las tablas, vistas o cualquier otro objeto que una base de datos pueda contener. Por lo tanto, los objetos globales como los roles, los espacios de tablas y las bases de datos quedan excluidos de esta lista.

El desencadenamiento se puede producir en diferentes momentos alrededor de los eventos:

- Antes de la ejecución del evento trigger: `ddl_command_start`.
- Después de la ejecución del evento trigger: `ddl_command_end`. En este contexto, la función `pg_event_trigger_ddl_commands` devuelve la información de los objetos y los comandos que hayan desencadenado el evento.
- Después de la ejecución de un evento que elimina un objeto: `sql_drop`. Este momento se sitúa antes de `ddl_command_end`. En este contexto, la función `pg_event_trigger_dropped_objects` devuelve la información de los objetos eliminados.
- Antes de la ejecución de un evento que modifica una tabla: `table_rewrite`. En este contexto, la función `pg_event_trigger_ table_rewrite_oid()` devuelve el OID de la tabla modificada y la función `pg_event_trigger_table_rewrite_reason()` devuelve un mensaje asociado a esta modificación.

Las funciones llamadas por los triggers son específicas: no pueden devolver nada, pero utilizan la palabra clave `event_trigger` como tipo de retorno.

d. Creación de un trigger sobre evento

El comando `CREATE EVENT TRIGGER` permite establecer un trigger sobre un evento utilizando una función de tipo `event_trigger` existente. La sinopsis del comando es la siguiente:

```
CREATE EVENT TRIGGER nombre
   ON evento
   [ WHEN TAG IN (filter_value [, ... ]) [ AND ... ] ]
   EXECUTE PROCEDURE nombre_función();
```

El tag puede ser una sentencia particular, por ejemplo: `DROP FUNCTION`.

2. Tratamientos asíncronos

Los triggers se ejecutan de manera síncrona, es decir, en la misma transacción que la sentencia desencadenante. Cuando el tratamiento es largo porque manipula numerosos registros o depende de elementos exteriores, normalmente es preferible utilizar un tratamiento asíncrono, es decir, separado de la transacción de la consulta.

No hay mecanismo interno de PostgreSQL que permita los tratamientos asíncronos, pero las herramientas Skytools ofrecen el componente PGQ, que es una cola de tratamientos de mensajes implementada en PostgreSQL y que permite desarrollar un demonio externo a PostgreSQL para consumir los eventos.

La alimentación de la cola de mensajes utiliza un trigger.

El desarrollo de un demonio PGQ va más allá del marco de este libro; sin embargo, se proporcionan numerosos detalles en la siguiente página: https://wiki.postgresql.org/wiki/PGQ_Tutorial

Control de funciones

La extensión plpgsql_check permite controlar el código PL/PgSQL de las funciones. El perfilador permite evaluar el detalle de la ejecución de las funciones.

1. Perfilador de funciones

Desde la versión 9.6 de PostgreSQL, existe una extensión que propone perfilar la ejecución de funciones PL/pgSQL para estudiar el comportamiento y, por lo tanto, optimizar su funcionamiento y rendimiento. Esta extensión no está disponible en forma de paquete binario en las distribuciones RedHat o Debian, por lo que es necesario compilar el módulo. Además, no se recomienda realizar las perfilaciones de funciones en un sistema de producción, sino en un entorno de pruebas.

El perfilador está formado por dos partes: un módulo cargado en la instancia PostgreSQL que recoge la información y un comando cliente que recupera la información de la instancia y produce el informe de perfilado.

2. Instalación

El código fuente del perfilador de funciones está disponible en la siguiente dirección: https://www.openscg.com/bigsql/docs/plprofiler/. La página de descarga es la siguiente: https://pypi.org/project/plprofiler/#files. En el momento de escribir este libro, la versión más reciente es la 3.3 y el archivo de las fuentes correspondiente es el siguiente: plprofiler-3.3.tar.gz.

La instalación del perfilador necesita algunas herramientas y librerías, así como los archivos de encabezado de PostgreSQL. Estas dependencias se pueden instalar utilizando el sistema de paquetes de la distribución. En un sistema RedHat:

```
sudo yum install postgresql10-devel gcc python-setuptools
python-devel perl
```

En un sistema Debian:

```
sudo apt-get install postgresql-server-dev-10 gcc python-setuptools
python-dev perl
```

Una vez que se instalan las dependencias, el módulo se compila utilizando los siguientes comandos en un sistema RedHat:

```
unzip openscg-plprofiler-380a6e19d5a7.zip
mv openscg-plprofiler-380a6e19d5a7 openscg-plprofiler
cd openscg-plprofiler
sudo PATH=/usr/pgsql-10/bin/:$PATH make USE_PGXS=1 install
cd python-plprofiler/
sudo PATH=/usr/pgsql-10/bin/:$PATH python setup.py install
```

En un sistema Debian:

```
unzip openscg-plprofiler-380a6e19d5a7.zip
mv openscg-plprofiler-380a6e19d5a7 openscg-plprofiler
cd openscg-plprofiler
sudo PATH=/usr/lib/postgresql/10/bin/:$PATH make USE_PGXS=1 install
cd python-plprofiler/
sudo PATH=/usr/lib/postgresql/10/bin/:$PATH python setup.py install
```

Entonces, el módulo se instala en la arborescencia de PostgreSQL, y el programa cliente en el sistema. Es necesario modificar la configuración de PostgreSQL para cargar el módulo en el momento del arranque de la instancia, después de crear la extensión en la base de datos deseada, en la misma instancia. En el archivo postgresql.conf de la instancia, el argumento que se debe modificar es `shared_preload_librairies`. Puede haber sido modificado con antelación. En ese caso hay que completar la lista de extensiones:

```
shared_preload_libraries =  'plprofiler'
```

Entonces, la instancia se debe arrancar de nuevo. En la base de datos donde están presentes las funciones que hay que probar, se debe crear la extensión:

```
CREATE EXTENSION plprofiler;
```

3. Perfilado de funciones

La utilización del perfilador se hace de dos maneras:

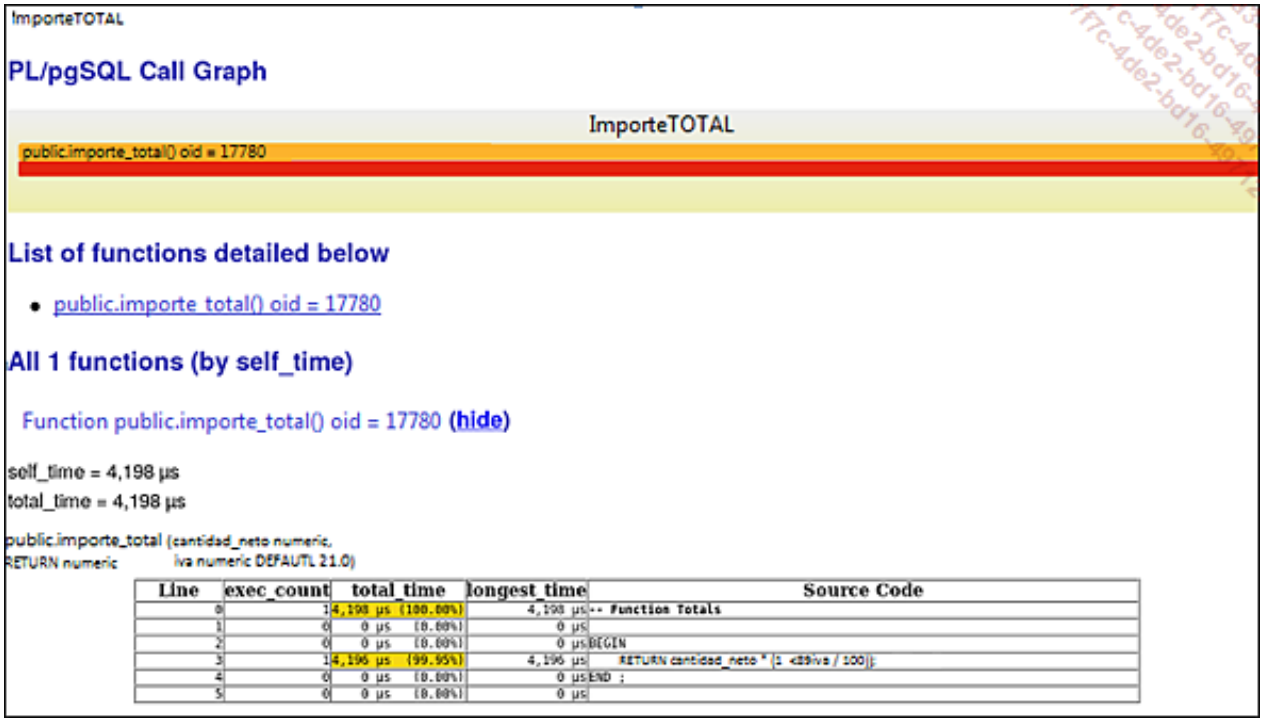
- El programa cliente lanza la función, conectándose a la instancia PostgreSQL.
- La función o las funciones se llaman por otra aplicación, por ejemplo durante unas pruebas de carga, y después el programa cliente recupera los datos de la sesión de prueba.

En todos los casos, el programa cliente genera el informe en formato HTML. Entonces el informe se puede leer usando un navegador web.

El perfilado de una función se hace simplemente utilizando la llamada a la función desde el cliente del perfilador. El comando completo es el siguiente:

```
plprofiler run -Upostgres -dclientes -c "select importe_total( 101.119 )
" --output=cantidadtotal.html --name=ImporteTOTAL
--title=ImporteTOTAL -desc="ImporteTOTAL"
select cantidad_total( 101.119 )
-- row1:
 importe_total: 121.94951400000000000000000
----
(1 rows)
SELECT 1 (0.007 seconds)
```

Entonces, el informe se genera en el archivo importetotal.html, que se visualiza en cualquier navegador web, tal y como se muestra en la siguiente pantalla:



El segundo método permite captar todas las llamadas de funciones después de guardarlas en una sesión. Durante las pruebas de una aplicación, este método permite asegurarse de captar todas las llamadas y, de esta manera, ser exhaustivo.

Con el siguiente script SQL y la utilización del cliente `pgbench`, podemos simular un gran número de llamadas y, de esta forma, producir los datos de perfilado pertinentes:

```
SET plprofiler.enabled TO true;
SET plprofiler.collect_interval TO 10;
select fact_num, cl_nombre, importe_factura( fact_num )
from facturas order by random() limit 1;
```

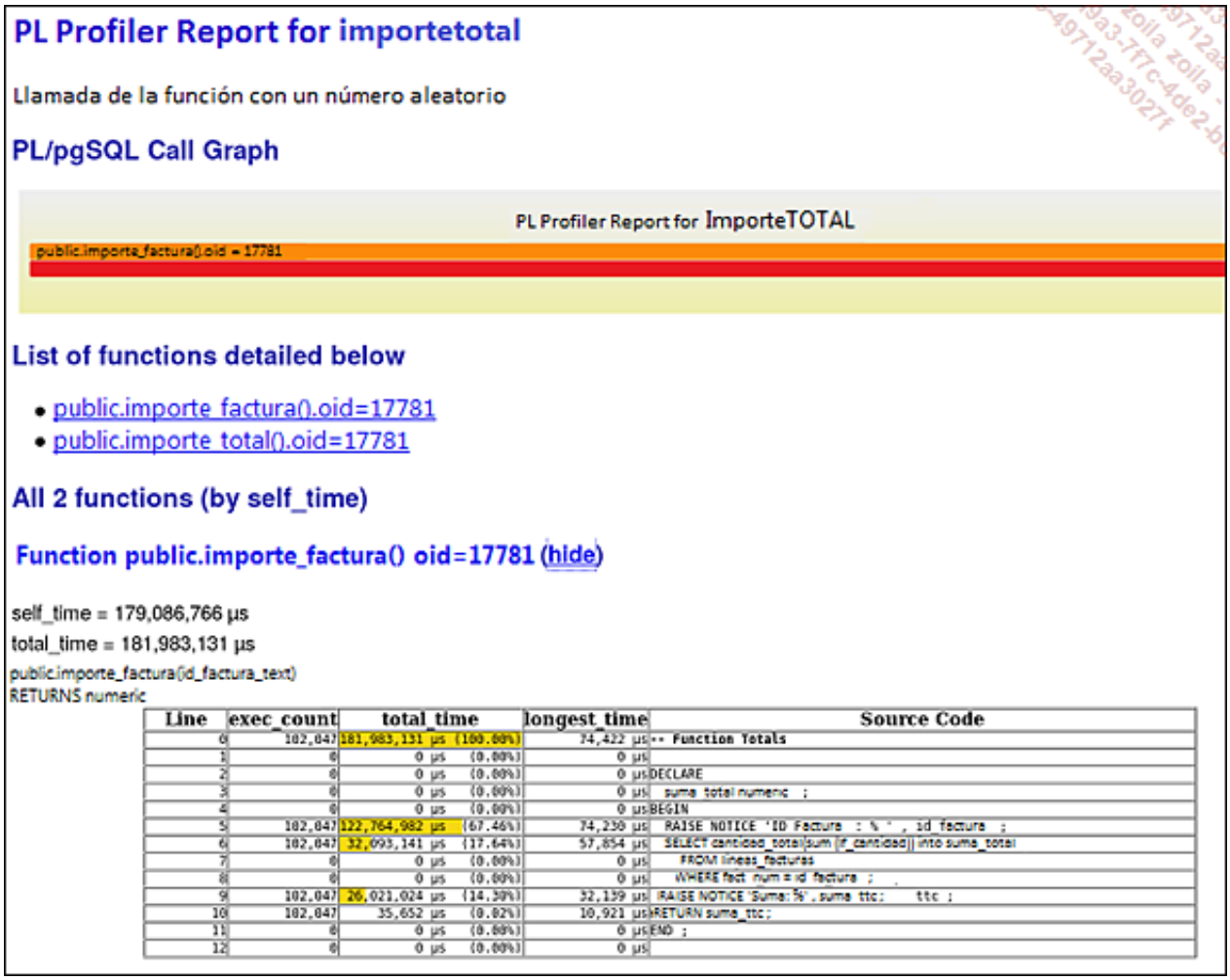
El script se invoca con el siguiente comando:

```
pgbench -f importetotal.sql -c 10 -j 10 -T 120 -Upostgres -d clientes
2>/dev/null
```

A continuación, los siguientes comandos registran la sesión en los datos internos del perfilador y después generan un informe en formato HTML:

```
plprofiler save --name=importetotal -U postgres -d clientes
plprofiler report --name=importetotal -U postgres -d clientes >
importetotal2.html
```

La siguiente pantalla muestra una parte del informe HTML:



PL Profiler Report for importetotal

Llamada de la función con un número aleatorio

PL/pgSQL Call Graph

PL Profiler Report for ImporteTOTAL

public.importe_factura().oid = 17781

List of functions detailed below

- public.importe_factura().oid=17781
- public.importe_total().oid=17781

All 2 functions (by self_time)

Function public.importe_factura() oid=17781 (hide)

self_time = 179,086,766 µs

total_time = 181,983,131 µs

public.importe_factura(id_factura_text)
RETURNS numeric

Line	exec_count	total_time	longest_time	Source Code
0	102,047	181,983,131 µs (100.00%)	74,422 µs	-- Function Totals
1	0	0 µs (0.00%)	0 µs	
2	0	0 µs (0.00%)	0 µs	DECLARE
3	0	0 µs (0.00%)	0 µs	suma_total numeric ;
4	0	0 µs (0.00%)	0 µs	BEGIN
5	102,047	122,764,982 µs (67.46%)	74,236 µs	RAISE NOTICE 'ID Factura : % ', id_factura ;
6	102,047	32,093,141 µs (17.64%)	57,854 µs	SELECT cantidad_total|sum (f_cantidad)| into suma_total
7	0	0 µs (0.00%)	0 µs	FROM lineas_facturas
8	0	0 µs (0.00%)	0 µs	WHERE fact_num = id_factura ;
9	102,047	26,021,024 µs (14.30%)	32,139 µs	RAISE NOTICE 'Suma: %' , suma_ttc; ttc ;
10	102,047	35,652 µs (0.02%)	10,921 µs	RETURN suma_ttc;
11	0	0 µs (0.00%)	0 µs	END ;
12	0	0 µs (0.00%)	0 µs	

En este ejemplo, es fácil comprobar que la mayor parte del tiempo (67 %) se invierte en la instrucción `RAISE NOTICE`, que es totalmente opcional; por lo tanto, en este caso tenemos un punto de optimización de la función, que se puede poner en marcha de manera muy sencilla.

4. Control de función PL/pgSQL

Durante la creación de una función PL/pgSQL, no se hacen comprobaciones sobre las consultas contenidas en el código. Solo se hacen los controles mínimos en el lenguaje PL/pgSQL, pero pueden aparecer errores durante la ejecución. Para mejorar la calidad del código de las funciones, la extensión plpgsql_check permite algunos controles adicionales:

- Comprobación de los objetos utilizados: tablas, vistas, atributos en las consultas con SQL incorporado.
- Uso de los tipos de datos de los argumentos.
- Variables y argumentos de entrada no utilizados, así como los argumentos de salida.
- Detección de porciones de código no realizables.
- Comprobación de la presencia de la sentencia `RETURN`.

La verificación consiste en llamar a la función `plpgsql_check_function` con la función que se debe verificar como argumento. Es posible forzar la verificación en cada llamada a las funciones, pero esto implica un sobrecoste que no siempre es aceptable.

a. Instalación

La instalación de esta extensión se hace utilizando el comando `pgxn`:

```
pgxn install plpgsql_check
```

Una vez que se ha instalado la extensión, hay que activarla en la base de datos deseada:

```
create extensión plpgsql_check;
```

b. Comprobación

Tomando como ejemplo la función `importe_factura`, se modifica el atributo `fact_num` y se provoca voluntariamente un error tipográfico, resaltando de esta manera el atributo que provoca el error. La verificación propone el atributo existente en la tabla, como muestra el siguiente ejemplo:

```
clientes=# select plpgsql_check_function('importe_factura(text)');
                              plpgsql_check_function
-------------------------------------------------------------------------
error:42703:6:SQL statement:column "fact_nu" does not exist
Query: SELECT importe_total(sum (lf_importe))       FROM registros_facturas
           WHERE fact_nu = id_factura
--               ^
Hint: Perhaps you meant to reference the column
"registros_facturas.fact_num".
(5 rows)
```

Es posible verificar todas las funciones presentes en una base de datos utilizando el catálogo interno. En particular, la tabla `pg_proc`, con la siguiente consulta:

```
SELECT p.oid, p.proname, plpgsql_check_function(p.oid)
  FROM pg_catalog.pg_namespace n
  JOIN pg_catalog.pg_proc p ON pronamespace = n.oid
  JOIN pg_catalog.pg_language l ON p.prolang = l.oid
 WHERE l.lanname = 'plpgsql'  AND p.prorettype <> 2279;
```

Explotación

Ejecución de una instancia

Una instancia de PostgreSQL se corresponde con la ejecución de un proceso `postgres`. Este proceso recibe las conexiones de los clientes y abre un nuevo proceso `postgres` para cada conexión exitosa. Otros procesos `postgres` para diferentes tareas se lanzan durante el arranque por el proceso padre: un proceso durante la carga de la escritura de los datos y otro para la recuperación de las estadísticas, como en el resultado del siguiente comando `ps`:

```
$ ps fx
32460 ?        S      0:01 /usr/lib/postgresql/10/bin/postgres -D
/var/lib/postgresql/10/main -c
config_file=/etc/postgresql/10/main/postgresql.conf
32462 ?        Ss     0:00  \_ postgres: 10/main: checkpointer process
32463 ?        Ss     0:00  \_ postgres: 10/main: writer process
32464 ?        Ss     0:00  \_ postgres: 10/main: wal writer process
32465 ?        Ss     0:01  \_ postgres: 10/main: autovacuum launcher process
32466 ?        Ss     0:01  \_ postgres: 10/main: stats collector process
32467 ?        Ss     0:00  \_ postgres: 10/main: bgworker: logical
replication launcher
2891 ?        Ss     0:00  \_ postgres: 10/main: postgres clientes [local]
idle
```

Aquí, se abre una conexión y el proceso en segundo plano `postgres` asociado indica el nombre de usuario (`postgres`), la base de datos (`clientes`), el origen de la conexión (`[local]`) y el estado del proceso (`idle`).

1. Definición de los archivos

PostgreSQL mantiene todos los archivos de un grupo de bases de datos en el directorio seleccionado durante la inicialización del grupo de bases de datos.

La única excepción viene con la utilización de los espacios de tablas, que permiten a una instancia almacenar los archivos en otros lugares.

Arborescencia de los directorios

El directorio de los datos de una instancia, creado con el comando `initdb`, contiene un determinado número de directorios y archivos. La siguiente lista presenta el conjunto de directorios de una instancia:

- `base`: contiene todos los directorios y archivos de las bases de datos.
- `global`: contiene las tablas comunes a toda la instancia.
- `pg_xact`: contiene los datos relativos a la validación de las transacciones. Este directorio se llamaba pg_clog antes de la versión 10 de PostgreSQL.
- `pg_commit_ts`: contiene los datos de fechas y hora de las transacciones.
- `pg_multixact`: contiene los datos relativos al estado de los datos multi-transacciones.
- `pg_subtrans`: contiene los datos relativos a las subtransacciones.
- `pg_tblspc`: contiene los enlaces simbólicos hacia los espacios de tablas.
- `pg_twophase`: contiene los datos relativos a las transacciones preparadas.
- `pg_wal`: contiene los archivos binarios de transacciones. Este directorio se llamaba pg_xlog antes de la versión 10 de PostgreSQL.

- `pg_stat`: contiene los archivos de las estadísticas recogidas.
- `pg_stat_tmp`: contiene los datos temporales de las estadísticas recogidas.
- `pg_replslot`: contiene los slots de replicación.
- `pg_logical`: contiene los datos relativos a la decodificación lógica.

Los archivos no se tienen que manipular. Solo el servidor debe leer y escribir directamente en estos directorios.

2. Nombre de los archivos

En el directorio `base`, PostgreSQL crea un directorio por cada base de datos de la instancia. El nombre del directorio no se corresponde con el nombre de la base de datos, pero es un identificador utilizado por el servidor. Los catálogos de sistema contienen el enlace entre el nombre de una base y su identificador.

La siguiente consulta sobre el catálogo de sistema permite encontrar esta asociación:

```
postgres=# SELECT oid,datname FROM pg_catalog.pg_database;
```

De la misma manera, en cada directorio hay muchos archivos que tienen como único nombre los identificadores. Estos archivos se corresponden principalmente con las tablas y los índices. Es posible que un nombre de archivo se corresponda con un identificador, pero algunos comandos modifican estos archivos. Los catálogos de sistema contienen todas las asociaciones.

La siguiente consulta sobre el catálogo de sistema permite encontrar esta asociación:

```
postgres=# SELECT relfilenode,relname FROM pg_catalog.pg_class;
```

Cuando el tamaño de un archivo sobrepasa 1 gigabyte, PostgreSQL crea un nuevo archivo, utilizando el nombre del archivo existente (`relfilenode`) y añadiéndole un número incremental, como en el siguiente modelo: `relfilenode.1`, `relfilenode.2`,`relfinode.3`, `[...]`.

Por lo tanto, PostgreSQL hace que ningún archivo de tabla o de índice sobrepase 1 gigabyte.

Administración del servidor

1. Configuración

El archivo de configuración propio de cada instancia de PostgreSQL se llama `postgresql.conf` y se sitúa generalmente en el directorio inicializado con la instancia, correspondiente a la opción `-D` del comando `initdb`. En los casos de los sistemas Debian, se sitúa en el directorio `/etc/postgresql/`, seguido de la versión principal de PostgreSQL y del nombre de la instancia elegida.

El archivo `postgresql.conf` puede contener las directivas `include` e `include_dir`, que pueden hacer referencia a otros archivos que contienen diversas directivas. Por defecto, estas dos directivas no se utilizan.

El archivo `postgresql.auto.conf` siempre se ubica en el directorio de los datos y contiene los argumentos modificados por el comando `ALTER SYSTEM`. Este archivo siempre se lee en último lugar, lo que hace que los argumentos presentes sobrecarguen los mismos argumentos presentes en el resto de los archivos de configuración.

Estos archivos se parecen a las directivas de configuración, que permiten adaptar el comportamiento del servidor al hardware sobre el que funciona, así como a las bases de datos utilizadas. Una vez que el servidor arranca, algunas de estas directivas se pueden sobrecargar con el comando `SET` o asociándolas a una base de datos o a un rol, con los comandos `ALTER` respectivos. Las directivas se pueden definir durante el arranque o durante la ejecución del servidor o bien con una sencilla recarga del archivo de configuración.

Las directivas que indican un valor de tamaño de memoria o de duración se pueden expresar utilizando una unidad, lo que simplifica la escritura y relectura del archivo de configuración.

Las unidades de tamaño son: `KB`, `MB`, `GB`. Las unidades de tiempo son: `ms`, `s`, `min`, `h`, `d`.

Los argumentos de configuración están dotados de valores por defecto, que se pueden revisar sobre todo en función del hardware realmente utilizado.

Cuando se añaden modificaciones al archivo de configuración, no se tienen en cuenta sin intervención del administrador. Es necesario que la instancia PostgreSQL vuelva a leer el archivo y algunas directivas necesitan un rearranque completo de la instancia, lo que implica una parada de las sesiones actuales y, por lo tanto, un corte de servicios. Cada argumento se puede modificar en un contexto concreto. La aplicación de las modificaciones solo es posible en este contexto:

- `postmaster`: arranca la instancia PostgreSQL. Las modificaciones se hacen en el archivo `postgresql.conf` o equivalente.
- `sighup`: envía una señal SIGHUP a los procesos postgres. Los scripts de arranque como `/etc/init.d/postgresql-10` o `pg_ctlcluster` envían esta señal con la opción `reload`.
- `backend`: arranca un proceso en segundo plano, es decir, durante la apertura de una nueva sesión. Cuando se modifica el valor en `postgresql.conf`, la señal SIGHUP modifica el argumento para las nuevas sesiones.
- `superuser`: utilización del comando `SET` por un superusuario (`postgres`). Cuando se modifica el valor en `postgresql.conf`, la señal SIGHUP modifica el argumento de las sesiones en las que el comando `SET` no se ha llamado.
- `user`: utilización del comando `SET` por cualquier usuario. Cuando se modifica el valor en `postgresql.conf`, la señal SIGHUP modifica el argumento de las sesiones en las que el comando `SET` no se ha llamado.

El contexto concreto de las directivas, así como los valores por defecto y la posición de cada una de las directivas en los archivos, se detallan en la vista `pg_settings` que, de esta manera, permite analizar la configuración existente.

a. Conexiones

- `listen_addresses = <cadena> (postmaster)`: indica la dirección o las direcciones TCP/IP del servidor que debe utilizar PostgreSQL. Las aplicaciones cliente deberán usar una de las direcciones indicadas para conectarse a la instancia. Cuando hay varias direcciones, se deben separar por una coma. El carácter * permite a la instancia escuchar en todas las interfaces de red del servidor. Si la cadena está vacía, entonces no es posible ninguna conexión a través de la red. Solo son posibles las conexiones a través de un socket UNIX.
- `port = <entero> (postmaster)`: indica el puerto TCP utilizado. El valor por defecto es 5432, pero, cuando varias instancias de PostgreSQL deben funcionar en el mismo servidor, es necesario cambiar este valor.

- `max_connections = <entero> (postmaster)`: indica el número máximo de conexiones que la instancia puede aceptar. El valor por defecto es 100. En función de los recursos utilizados por cada conexión, se puede considerar bajar esta cifra y adaptar las aplicaciones, principalmente los sitios web, utilizando los administradores de conexiones, como PgPool.
- `superuser_reserved_connections = <entero> (postmaster)`: indica el número de conexiones reservadas a los superusuarios de la instancia. Esta configuración permite conservar las conexiones disponibles entre todas las conexiones disponibles por `max_connections`.
- `ssl = <booleano> (postmaster)`: permite las conexiones encapsuladas en el SSL. Este argumento está desactivado por defecto.

b. Memoria

- `shared_buffers = <tamaño> (postmaster)`: indica el número de segmentos de memoria compartida utilizados por la instancia. Un segmento de memoria compartida vale 8192 bytes. El número mínimo es de 16 segmentos, es decir 128 kilobytes, pero este número no debe bajar de 2 veces el número máximo de conexiones entrantes (2 x `max_connections`). Aumentar este valor mejora de manera significativa el rendimiento global de PostgreSQL. En un servidor dedicado, esta directiva de configuración no debería sobrepasar un tercio de la memoria RAM disponible. Si hay otras aplicaciones que funcionan en un mismo servidor, hay que tenerlas en cuenta. De hecho, los límites altos de este ajuste son 8 GB para un sistema GNU/Linux y 2 GB para un sistema Windows.
- `work_mem = <tamaño> (user)`: indica la cantidad de memoria utilizable durante las operaciones de ordenación y de hasheo antes de usar los archivos temporales. El valor por defecto es 4 MB. Para las consultas complejas o durante conexiones simultáneas, se pueden realizar varias operaciones de ordenación y hasheo al mismo tiempo y cada una de ellas utilizar la cantidad de memoria indicada. De esta manera, la cantidad total de memoria puede ser varias veces este valor. Por lo tanto, hay que encontrar un equilibrio entre la cantidad de memoria para cada operación, el número de conexiones simultáneas y la cantidad de memoria disponible.
- `maintenance_work_mem = <tamaño> (user)`: indica la cantidad de memoria disponible para las operaciones de mantenimiento, tales como la creación de índices o la limpieza de una tabla. El valor por defecto es 64 MB. Este valor por defecto puede ser muy bajo y un aumento significativo del mismo puede tener efectos positivos en las operaciones afectadas.
- `autovacuum_work_mem = <tamaño> (user)`: indica la memoria utilizada por las tareas de `autovacuum`. Por defecto, se utilizada el tamaño de `maintenance_work_mem`.
- `temp_file_limit = <tamaño> (superuser)`: indica el tamaño máximo que pueden ocupar los archivos temporales de una sesión. Por defecto, el valor es -1, que indica que no hay límite.
- `dynamic_shared_memory_type = posix|sysv|windows|mmap (postmaster)`: indica la implementación de memoria compartida. En un sistema GNU/Linux, `posix` se utiliza por defecto y, en un sistema Windows, `windows` se utiliza por defecto. La implementación `sysv` se ha utilizado históricamente y `mmap` solo se debería usar para depurar la memoria.
- `bgwriter_delay = <tiempo> (sighup)`: indica el periodo entre dos vueltas de actividad del proceso `bgwriter`. En cada vuelta, el proceso escribe el número de unidades de memoria RAM y después se pausa. La resolución mínima es de 10 ms.
- `bgwriter_lru_maxpages = <entero> (sighup)`: indica el número de unidades de memoria RAM. El valor cero desactiva el proceso `bgwriter`. Solo funcionan las `CHECKPOINT`. El valor por defecto es 100.

- `bgwriter_lru_multiplier = <decimal> (sighup)`: indica un factor de multiplicación para determinar el número de unidades de memoria RAM que se deben tratar durante la próxima vuelta, en función del número de unidades de memoria RAM «dirty» encontradas. El valor por defecto es 2. Independientemente del número de unidades de memoria RAM «dirty» encontradas, la próxima vuelta no tratará más de `bgwriter_lru_maxpages` unidades de memoria RAM. Un valor superior a 1 se considera como agresivo.

c. Recolectores de estadísticas

- `track_activities = <boolean> (superuser)`: activa la recogida de información sobre los comandos ejecutados. Por defecto, este argumento está activado.
- `track_counts = <boolean> (superuser)`: activa la recogida de información sobre la actividad de las bases de datos. Por defecto, este argumento está activado, principalmente para el funcionamiento del proceso `autovacuum`.
- `track_io_timing = <boolean> (superuser)`: activa la recogida de los tiempos de las escrituras y lecturas sobre el sistema de almacenamiento. Por defecto, este argumento está desactivado. El comando `pg_test_timing` permite medir la sobrecarga que genera su activación.
- `track_functions = all|pl (superuser)`: activa el seguimiento de la ejecución de las funciones. El valor `pl` traza las funciones de los lenguajes tales como `plpgsql`, mientras que `all trace` también traza las funciones SQL y `c`. Por defecto, este argumento está desactivado.

d. Opciones de las herramientas de limpieza

Los comandos `vacuum` y el demonio de `autovacuum` utilizan algunas directivas de configuración, así como los ajustes de las tablas para desencadenar las operaciones de mantenimiento.

- `vacuum_cost_delay` = <tiempo> (user): indica un tiempo durante el cual se detiene el tratamiento, cuando se alcanzan los costes. El valor por defecto, 0, desactiva estas pausas. La resolución real es de 10 milisegundos, lo que hace que la pausa efectivamente sea un múltiplo de 10 milisegundos. Este argumento es útil para evitar bloquear las lecturas y escrituras, incluso si los tratamientos de limpieza pueden ser más largos.
- `vacuum_cost_page_hit = <entero> (user)`: indica el coste estimado de la limpieza de una página en memoria.
- `vacuum_cost_page_miss = <entero> (user)`: indica el coste estimado de la limpieza de una página en disco.
- `vacuum_cost_page_dirty = <entero> (user)`: indica el coste de la limpieza de una página sucia.
- `vacuum_cost_limit = <entero> (user)`: indica el coste total de limpieza desencadenando la pausa del tratamiento. El valor por defecto es 200. Este valor se corresponde con la suma de los argumentos anteriores.

Proceso Autovacuum

- `autovacuum = <booleano> (sighup)`: activa el demonio `autovacuum`. Se activa por defecto.
- `autovacuum_naptime = <tiempo> (sighup)`: indica el periodo de activación de los tratamientos de limpieza automática en segundos. El plazo por defecto es 60 segundos.

- autovacuum_max_workers = <entero> (postmaster): número de tratamientos en paralelo que el proceso autovacuum puede lanzar. El valor por defecto es 3.
- autovacuum_vacuum_threshold = <entero> (sighup): indica el número de registros modificados necesarios para desencadenar una limpieza sobre una tabla. El valor por defecto es 50 y se puede indicar en los ajustes de cada tabla (ver el capítulo Definición de los datos, sección Modificación de una tabla).
- autovacuum_analyze_threshold = <entero> (sighup): indica el número de registros modificados necesario para desencadenar un análisis sobre una tabla. El valor por defecto es 50 y se puede indicar en los ajustes de cada tabla (ver el capítulo Definición de los datos, sección Modificación de una tabla).
- autovacuum_vacuum_scale_factor = <porcentaje> (sighup): indica un porcentaje de la tabla que hay que añadir al número de registros que se debe modificar para desencadenar una limpieza sobre una tabla. El valor por defecto es 0.2 (20 %) y se puede indicar en los ajustes de cada tabla (ver el capítulo Definición de los datos, sección Modificación de una tabla).
- autovacuum_analyze_scale_factor = <porcentaje> (sighup): indica un porcentaje de la tabla que hay que añadir al número de registros que se debe modificar para desencadenar un análisis sobre una tabla. El valor por defecto es 0.1 (10 %) y se puede indicar en los ajustes de cada tabla (ver el capítulo Definición de los datos, sección Modificación de una tabla).
- autovacuum_vacuum_cost_delay = <entero> (sighup): sobrecarga el argumento vacuum_cost_delay para el proceso autovacuum. El valor por defecto es 20 ms.
- autovacuum_vacuum_cost_limit = <entero> (sighup): sobrecarga el argumento vacuum_cost_limit para el proceso autovacuum. El valor por defecto es -1.
- autovacuum_freeze_max_age = <entero> (postmaster): indica la edad, en número de transacciones, que una tabla puede esperar (atributo relfrozenxid de la tabla pg_class) antes de que el proceso autovacuum lance una tarea específica para prevenir el problema de bucles de los identificadores de transacciones. El valor por defecto es 200 millones, lo que hace que ninguna tabla pueda tener un identificador de transacción que sobrepase este valor respecto al identificador de transacción actual. Este ajuste puede ser más bajo para cada tabla (ver el capítulo Definición de los datos, sección Modificación de una tabla).

e. Logs de actividad

Los archivos de trazas de actividad permiten guardar los eventos que aparecen durante la ejecución de una instancia. Existen diferentes métodos de registro, diferentes naturalezas de mensajes y la cantidad de información también se puede ajustar.

Los siguientes argumentos permiten indicar dónde se deben escribir los mensajes. Existen dos estrategias: PostgreSQL se encarga de todo el trabajo o envía los mensajes a un software especializado, como por ejemplo syslog en los sistemas UNIX. Esta segunda estrategia normalmente es más eficaz, por lo que se estudia con detenimiento.

- log_destination= stderr|syslog|csvlog|eventlog (sighup): indica dónde se deben enviar los mensajes de las trazas de actividad del servidor. Hay cuatro posibilidades:
  - stderr, valor por defecto,
  - syslog, a un software de tipo syslog,
  - csvlog, a un archivo csv,

    - `eventlog` (únicamente para los sistemas Windows).

      Es posible utilizar varios destinos separándolas por una coma.

- `logging_collector = <boolean> (postmaster)`: activa el proceso en segundo plano que recolecta las trazas de actividad.
- `log_directory = <cadena> (sighup)`: indica el directorio utilizado para escribir el archivo de trazas cuando el argumento `logging_collector` está activado. La ruta puede ser absoluta o relativa al directorio de los datos.
- `log_filename = <cadena> (sighup)`: indica el nombre del archivo de trazas cuando `logging_collector` está activado. Es posible utilizar los mismos motivos que los de la función `strftime()` para personalizar el nombre del archivo. Por defecto, el valor es `postgresql-%Y-%m-%d_%H%M%S.log`.
- `log_rotation_age = <tiempo> (sighup)`: indica el tiempo de vida de un archivo de trazas de actividad cuando el argumento `logging_collector` está activado. El valor por defecto se corresponde con 24 horas y, una vez alcanzado el plazo, se crea un nuevo archivo utilizando el motivo de la directiva `log_filename`.
- `log_rotation_size = <tamaño> (sighup)`: indica el tamaño máximo de un archivo de trazas cuando el argumento `logging_collector` está activado. El valor por defecto es 10 megabytes y cuando se alcanza el tamaño se crea un nuevo archivo, según el motivo de la directiva `log_filename`. El valor 0 desactiva la rotación basada en el tamaño del archivo.
- `log_truncate_on_rotation = <booleano> (sighup)`: cuando el argumento `logging_collector` está activado, esta opción permite a PostgreSQL eliminar el contenido de un archivo de trazas. Según el motivo utilizado, PostgreSQL puede tener que escribir las trazas en un archivo existente. Según el valor del argumento, las antiguas trazas se eliminarán o no.
- `syslog_facility = <enum> (sighup)`: cuando la directiva `log_destination` vale `syslog`, este argumento determina la opción utilizada por Syslog. Los valores pueden ser: `LOCAL0`, `LOCAL1`, `LOCAL2`, `LOCAL3`, `LOCAL4`, `LOCAL5`, `LOCAL6`, `LOCAL7` y el valor por defecto es `LOCAL0`.
- `syslog_ident = <cadena> (sighup)`: cuando está activado el envío de las trazas a Syslog, indica el nombre utilizado para identificar al remitente de los mensajes. El valor por defecto es `postgres`.

La siguiente lista reúne las directivas que permiten ajustar el nivel de detalle de los mensajes:

- `client_min_messages = <cadena> (user)`: durante una sesión, indica la cantidad de mensajes enviados al software cliente. Los diferentes niveles son: `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `LOG`, `NOTICE`, `WARNING`, `ERROR`, `FATAL` y `PANIC`. El nivel `DEBUG5` envía mucha información, y el nivel `PANIC`, muy poco. Cada nivel incluye los niveles inferiores, es decir, los que envían menos mensajes. El valor por defecto es `NOTICE`.
- `log_min_messages = <cadena> (superuser)`: indica la cantidad de mensajes escritos en los archivos de traza. Los diferentes niveles son: `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL` y `PANIC`. El nivel por defecto es `WARNING`.
- `log_min_error_statement = <cadena> (superuser)`: indica el nivel de los mensajes que se deben registrar durante un error en las sentencias SQL. Los diferentes niveles son: `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL` y `PANIC`. El valor por defecto es `ERROR`, entonces el valor `PANIC` desactiva esta funcionalidad.
- `log_error_verbosity = <cadena> (superuser)`: indica el nivel de detalle escrito en las trazas. Los diferentes niveles son: `TERSE`, `DEFAULT` y `VERBOSE`.

- log_min_duration_ statement = <tiempo> (superuser): define un tiempo mínimo de ejecución para una consulta, expresado en milisegundos, más allá del cual esta consulta se anotará en los archivos de traza. El valor 0 permite anotar todas las consultas, mientras que el valor -1 desactiva esta funcionalidad.

Las siguientes líneas listan las directivas, que permiten seleccionar la naturaleza de la información que se escribe en los archivos de traza.

- log_connections = <booleano>, log_disconnections = <booleano> (backend): indica que se debe anotar las conexiones y desconexiones al servidor. Con las desconexiones también se anotan las duraciones de las sesiones.
- log_statement = none|ddl|mod|all (superuser): indica la naturaleza de las consultas SQL anotadas. Los valores pueden ser none, ddl, mod y all. La opción ddl solo anota los comandos de definición, mod anota los comandos de modificación y all, los dos anteriores. El valor por defecto es none.
- log_line_prefix = <cadena> (sighup): indica el formato del prefijo de los registros escritos en los archivos de traza. Este formato puede contener caracteres de escape específicos, que permiten anotar información como el nombre de usuario o el nombre de la base de datos. Por ejemplo:
  - %u: nombre de usuario,
  - %d: nombre de la base de datos,
  - %p: PID del proceso,
  - %l: número de registros de la sesión,
  - %x: identificador de la transacción,
  - %a: nombre de la aplicación definida por el cliente, con la directiva application_name,
  - %t: fecha y hora de la consulta,
  - %m: fecha y hora de la consulta en milisegundos,
  - %n: fecha y hora de la consulta en milisegundos, en EPOCH Unix,
  - %s: fecha y hora del inicio de la sesión,
  - %q: Fin del prefijo para el proceso no correspondiente a una sesión.
- log_lock_waits = <booleano> (superuser): activa un mensaje en las trazas cuando una consulta sobrepasa el umbral definido por deadlocks_timeout, es decir, un segundo por defecto. Desactivado por defecto.
- log_checkpoint = <booleano> (superuser): activa un mensaje en las trazas sobre la ejecución de los puntos de controles.
- log_replication_commands = <booleano> (superuser): permite trazar los comandos de replicación. El valor off, por defecto, desactiva esta traza.
- log_temp_files = <tamaño> (superuser): permite trazar los archivos temporales cuyo tamaño sobrepasa el ajuste. El valor -1, por defecto, desactiva esta traza.
- log_autovacuum_min_duration = <tiempo> (sighup): permite trazar las tareas de autovacuum cuyo tiempo sobrepasa el ajuste. El valor -1, por defecto, desactiva el ajuste.

f. Archivos de escritura de las transacciones

Los archivos de traza binarios son un mecanismo de escritura de las transacciones. Cada transacción se escribe en un archivo binario, llamado WAL, de *Write Ahead Log*, antes de que los datos se escriban realmente en los archivos de las tablas y de los índices.

Este mecanismo permite al servidor obtener un mejor rendimiento, garantizando la escritura de los datos. También se utiliza como componente de base para la replicación integrada y los mecanismos de restauración de datos.

La siguiente lista reúne los argumentos principales:

- `wal_level = minimal|replica|logical (postmaster)`: determina la cantidad de información escrita en los archivos de transacciones. Estos niveles de información determinan las funcionalidades de restauración y replicación de los datos activados en la instancia. `Minimal` no permite ninguna de estas funcionalidades; `replica` sustituye `archivo` y `hot_standby` desde la versión 9.6, y permite la replicación física, y `logical` permite la replicación lógica. Cada nivel incluye los niveles anteriores e induce una cantidad de información escrita cada vez más importante. Desde la versión 10, el valor por defecto es `replica`.

- `fsync = <booleano> (sighup)`: esta opción permite al servidor forzar la escritura de los archivos de traza binarios en el disco utilizando una llamada al sistema de tipo `fsync()`. De esta manera, los datos se escriben realmente en el disco, garantizando dicha escritura en caso de parada accidental del sistema operativo o del hardware. La desactivación de la sincronización permite obtener mejor rendimiento. PostgreSQL delega al sistema operativo la sincronización de la escritura. Sin embargo, existe riesgo de pérdida de datos en los casos de parada accidental, incluidos los de las transacciones validadas.

- `synchronous_commit = on|remote_apply|remote_write| local|off (user)`: indica si la validación de la transacción debe esperar la escritura en discos (`on`) o si la respuesta se envía antes al cliente (`off`). Al contrario de lo que sucedía con la desactivación de `fsync`, no hay riesgo de corrupción de datos, sino únicamente una posible pérdida de transacciones. Cuando se utiliza la replicación síncrona, el argumento `remote_write` permite asegurarse de que los datos se han recibido correctamente por el sistema operativo de la instancia o de las instancias «`standby`». `remote_apply` permite asegurarse de que los datos están disponibles en la instancia o las instancias «`standby`», mientras que `local` solo espera el sistema local. Esta directiva de configuración permite controlar de manera fina el sincronismo de las transacciones y posibilita la modificación del argumento en el marco de una transacción con la sentencia `SET`. Por ejemplo, si la directiva vale `local`, el arranque de la siguiente transacción permite hacer síncrona la transacción:

```
begin;
set local synchronous_commit = remote_apply;
...
commit;
```

- `wal_sync_method = <cadena> (sighup)`: indica el método de sincronización utilizado. Los métodos son: `FSYNC`, `FDATASYNC`, `OPEN_SYNC` u `OPEN_DATASYNC`. El valor por defecto depende del sistema operativo.

- `full_page_writes = <booleano> (sighup)`: activa la escritura de los datos por páginas enteras, garantizando la restauración de la página. Activada por defecto, la desactivación permite un volumen de datos más bajo en los archivos de transacciones con un riesgo de corrupción durante la restauración.

- `wal_buffers = <tamaño> (postmaster)`: indica el número de segmentos de página en memoria compartida para los datos de las trazas binarias. El valor por defecto es -1, lo que hace que el tamaño se calcule como 1/32 de `shared_buffers`.

- `wal_writer_delay = <tiempo> (sighup)`: plazo entre cada vuelta del proceso de escritura de los archivos de transacciones; 200 milisegundos por defecto.
- `commit_delay = <tiempo> (superuser)`: indica el tiempo entre la validación de una transacción y la escritura en el archivo de trazas. Un plazo superior a cero permite la escritura de varias transacciones al mismo tiempo. Sin embargo, si el número mínimo de transacciones definido por `commit_siblings` no se alcanza, la sincronización no se realiza.
- `commit_siblings = <entero> (user)`: indica el número de transacciones mínimo que se pueden tratar durante el periodo definido por `commit_delay`.
- `max_wal_size = <tamaño> (postmaster)`, `min_wal_size = <tamaño> (postmaster)`: estos dos argumentos indican el rango del volumen de archivos de traza de transacciones, en el que es deseable proceder a los CHECKPOINT. Este rango de valores sustituye al valor `checkpoint_segment` a partir de la versión 9.5 de PostgreSQL. Este rango permite absorber los picos de escritura, minimizando el volumen de los archivos de traza de transacciones cuando hay poca actividad. Los valores por defecto son 80 MB para `min_wal_size` y 1 GB para `max_wal_size`.
- `checkpoint_completion_target = <decimal> (sighup)`: indica la fracción entre dos puntos de control para leer el siguiente punto de control. El valor por defecto es 0.5.
- `checkpoint_timeout = <tiempo> (sighup)`: indica el tiempo máximo, en segundos, entre dos puntos de control. El valor por defecto es de 5 minutos. Aumentar este tiempo puede mejorar el rendimiento, pero las sincronizaciones sobre el disco pueden ser más largas.
- `checkpoint_warning = <tiempo> (sighup)`: desencadena el envío de mensajes en las trazas de actividad cuando los puntos de control son más frecuentes que este tiempo, expresado en segundos. Estos mensajes en los archivos de trazas de actividad son un buen índice para aumentar los valores de `min_wal_size` y `max_wal_size`.

Almacenamiento y restauración

- `archive_mode = <booleano> (postmaster)`: activa el almacenamiento de los archivos de traza de transacciones.
- `archive_command = <cadena> (sighup)`: permite almacenar los archivos de transacciones con diferentes comandos de sistema. En esta cadena, el carácter `%p` se corresponde con la ruta absoluta del archivo y `%f` se corresponde con el único nombre del archivo (ver la sección Copia de seguridad sobre la marcha a lo largo de este capítulo).
- `archive_timeout = <tiempo> (sighup)`: fueza el almacenamiento de los archivos de transacción cuando se alcanza el tiempo indicado. Un tiempo demasiado bajo puede penalizar en términos de volumen ocupado en disco. El valor 0 desactiva el desencadenamiento.

g. Replicación

- `max_wal_senders = <entero> (postmaster)`: indica el número máximo de clientes de replicación aceptados. Los clientes de replicación pueden ser servidores PostgreSQL «standby», aplicaciones como `pg_base_backup` o cualquier aplicación que utilice el protocolo cliente/servidor de replicación. Este argumento tiene un valor de 10 por defecto desde la versión 10, lo que permite hacer la replicación, y de 0 para las versiones anteriores. Es importante suministrar un valor más grande de lo esperado para conservar las posibilidades de crecimiento de los servidores «standby».
- `max_replication_slots = <entero> (postmaster)`: determina el número de «slots» de replicación. Por defecto, el argumento vale 10 desde la versión 10, lo que activa los slots de replicación, y 0 para las versiones anteriores.

- `wal_keep_segments = <entero> (sighup)`: indica el número de archivos de transacciones ya archivados que el servidor conserva para paliar las desconexiones de los servidores «standby». Cada segmento ocupa 16 MB. Por lo tanto, hay que vigilar el volumen de estos archivos. Cuando se instala un almacenamiento, no es necesario conservar los archivos de transacciones con este argumento porque los servidores «standby» se pueden aprovechar para el almacenamiento.
- `wal_sender_timeout = <tiempo> (sighup)`: finaliza las conexiones de replicación cuando están inactivas más allá del tiempo indicado. El valor por defecto es de 60 segundos.
- `track_commit_timestamp = <booleano> (postmaster)`: anota la fecha y hora de las validaciones de transacciones.

Servidor primary

- `synchronous_standby_names = <cadena> (sighup)`: lista los nombres de los servidores «standby» cuya replicación es síncrona, es decir, que las transacciones se validan en el «standby» antes de responder al cliente del «primary». Existen dos operadores que permiten optimizar el funcionamiento de esta lista: `FIRST` y `ANY`. El operador `FIRST` permite definir una lista ordenada de N servidores «standby» que deberán ser síncronos, mientras que el operador `ANY` define una lista no ordenada de N servidores «standby» que deben ser síncronos. N es un operando del operador. De esta manera se define una parte de los servidores de la lista que deben ser síncronos. Por ejemplo, dos servidores «standby» en una lista de cuatros servidores «síncronos», de tal manera que si una parte de estos servidores deja de estar disponible, el servidor «primario» utiliza el resto de la lista para asegurar el sincronismo de las transacciones. De hecho, este mecanismo permite asegurar una continuidad de servicio a pesar de la no disponibilidad de una parte de los servidores «standby». El siguiente ejemplo define una lista de cuatros servidores síncronos en la que dos deben responder para permitir la validación de la transacción:

```
synchronous_standby_names = 'ANY S (pgs1, pgs2, pgs3, pgs4) '
```

Los servidores «standby» no se listan, son asíncronos por defecto.

- `vacuum_defer_cleanup_age = <entero> (sighup)`: indica el número de transacciones a partir de las cuales los comandos `VACUUM` y actualizaciones `HOT` enviarán a los servidores «standby» la limpieza de los registros muertos. Por defecto, este número de transacciones es cero, lo que hace que los registros muertos se eliminen tan pronto como sea posible. El aumento de este valor permite la ejecución de consultas en los servidores «standby». Ver también el argumento `hot_standby_feedback` sobre los servidores «standby».

Servidor «standby»

Los siguientes argumentos solo se utilizan en un servidor «standby»:

- `hot_standby = <booleano> (postmaster)`: indica la posibilidad de conectarse a la instancia «standby» para ejecutar en ella las consultas en modo de solo lectura.
- `max_standby_archive_delay = <tiempo>`, `max_standby_ streaming_delay = <tiempo> (sighup)`: plazo máximo antes de la anulación de una consulta ejecutada en una instancia «standby». Cuando esta instancia recibe los datos desde el flujo de replicación o los archivos de traza de transacciones, las consultas actuales se deben anular para aplicar los datos replicados. El valor por defecto es de 30 segundos. El valor -1 permite desactivar esta anulación de consulta y esperar su fin para aplicar la replicación.

- `wal_receiver_status_interval = <tiempo> (sighup)`: indica la frecuencia mínima del proceso en segundo plano «wal receiver» para enviar a la instancia «master» la información de replicación. Los datos están presentes en la instancia «master» por la vista `pg_stat_replicationview` y se corresponden con las posiciones en los archivos de traza de transacciones de los datos recibidos, guardados en disco y reproducidos en la instancia «standby». El valor por defecto es de 10 segundos.
- `hot_standby_feedback = <booleano> (sighup)`: permite a la instancia «hot standby» enviar la información de las consultas en curso a la instancia «master». Esta información permite no sufrir cancelaciones de consultas debido a las tareas de mantenimiento, pero puede implicar un consumo superfluo de espacio de almacenamiento en la instancia «master». Este argumento está desactivado por defecto.
- `wal_receiver_timeout = <tiempo> (sighup)`: termina las conexiones de replicación inactivas durante un tiempo superior al valor definido. El valor por defecto es de 60 segundos.

h. Rendimiento de las consultas

- `seq_page_cost = <decimal> (user)`: constante de coste que permite estimar el coste de lectura de una página de datos durante una lectura secuencial de varias páginas en disco. El valor por defecto es 1. Esta constante se puede sobrecargar para un espacio de tablas dado por el comando `ALTER TABLESPACE`.
- `random_page_cost = <decimal> (user)`: constante de coste que permite estimar el coste de lectura de una página aleatoriamente en disco. El valor por defecto es 4.0, lo que significa que esta lectura se estima cuatro veces más costosa que la lectura secuencial de una página. Esta constante se puede sobrecargar para un espacio de tablas dado por el comando `ALTER TABLESPACE`. Reducir esta constante puede incitar al planificador de consultas a hacer lecturas de índices en lugar de lecturas secuenciales de tablas. La utilización de periféricos de almacenamiento como los SSD permite bajar significativamente este valor.
- `cpu_tuple_cost = <decimal> (user)`: constante de coste que estima el coste de tratamiento de un registro en una consulta, relativa a `seq_page_cost`. El valor por defecto es 0.01.
- `cpu_index_tuple_cost = <decimal> (user)`: constante de coste que estima el coste de tratamiento de una entrada de índice en una consulta relativa a `seq_page_cost`. El valor por defecto es 0.005.
- `cpu_operator_cost = <decimal> (user)`: constante de coste que estima el coste de tratamiento de una llamada de función u operador, relativa a `seq_page_cost`. El valor por defecto es 0.0025.
- `effective_cache_size = <tamaño> (user)`: da una estimación de la cantidad de datos de memoria RAM utilizados como caché del sistema de archivos y, por lo tanto, de los datos en disco. Cuanto más grande es el valor, más se incita al planificador de consultas a utilizar lecturas de índices en lugar de lecturas secuenciales de tablas. En el tamaño identificado se incluye la memoria utilizada por `shared_buffers` y los datos efectivamente en caché, sabiendo que varias consultas concurrentes pueden no tener necesidad de los mismos datos. El valor por defecto es 4 GB.
- `default_statistics_target = <entero> (user)`: define la profundidad de las estadísticas recogidas sobre los datos de las columnas de tablas. Este valor se puede sobrecargar para cada columna por el comando `ALTER TABLE SET STATISTICS`. Un valor grande aumenta el tiempo de tratamiento de la consulta `ANALYZE`, pero puede mejorar la calidad de la estimación del planificador de consultas. El valor por defecto es 100.

- `constraint_exclusion = on|off|partition (user)`: controla la optimización de las consultas utilizando las restricciones de tablas. El valor por defecto, `partition`, permite usar esta optimización durante la utilización de tablas heredadas de la partición. El valor `on` permite esta optimización para todas las tablas y el valor `off` lo prohíbe.
- `from_collapse_limit = <entero> (user)`: indica al planificador de consultas el número de subconsultas en la lista `FROM` que podrá fusionar con la consulta superior. El valor por defecto es 8. Un valor más pequeño reduce el tiempo de planificación, pero puede producir planes con menor rendimiento.
- `join_collapse_limit = <entero> (user)`: indica el número de joins que el planificador de consultas va a volver a escribir como miembro de la sentencia `FROM`, sin contar los `FULL JOIN`. El valor por defecto es 8. Un valor más pequeño reduce el tiempo de planificación, pero puede producir planes con menor rendimiento. El valor 1 prohíbe la reescritura de los joins explícitos. Entonces, los joins se hacen en la sentencia o se escriben en la consulta.

i. Carga de los módulos

- `shared_preload_libraries = <cadena> (postmaster)`: lista de los módulos dinámicos que se deben cargar durante el arranque de la instancia PostgreSQL. Se indica los módulos separados por comas, en una única directiva, vigilando con no sobrecargarla porque solo se utiliza la última directiva.
- `session_preload_libraries = <cadena> (backend)`: lista de los módulos dinámicos que se deben cargar durante el arranque de la instancia PostgreSQL. Esta lista se puede especificar para un rol o una base de datos. Es posible modificarla sin volver a arrancar la instancia PostgreSQL.

j. Otras opciones

- `search_path = <cadena> (user)`: indica la sentencia de lectura de los espacios de nombres. Cuando los nombres de los objetos no se cualifican, el servidor utiliza esta lista para encontrar estos objetos. Por defecto, esta lista vale `public`, `$user`.
- `cluster_name = <cadena> (postmaster)`: cuando esta directiva no está vacía, la cadena se añade al nombre del proceso, visible en las herramientas como `ps`.
- `default_transaction_isolation = <cadena> (user)`: indica el nivel de aislamiento de las transacciones. Los dos niveles utilizables por PostgreSQL son `read committed` y `serializable`. El valor por defecto es `read committed`.
- `statement_timeout = <tiempo> (user)`: indica el tiempo máximo de ejecución de una consulta. El valor por defecto es cero y desactiva este tiempo máximo.
- `idle_in_transaction_session_timeout = <tiempo> (user)`: define un plazo en milisegundos, más allá del cual una transacción no terminada se anula automáticamente. El valor por defecto es cero y desactiva esta funcionalidad.
- `lc_messages = <cadena> (superuser)`, `lc_monetary = <cadena>`, `lc_time = <cadena>`, `lc_numeric = <cadena> (user)`: indica los argumentos locales que hay que tener en cuenta para la visualización de los valores numéricos, monetarios o temporales. Estos valores dependen de los argumentos locales disponibles en el sistema operativo. Estos argumentos también se definen durante la llamada al comando `initdb`.
- `client_encoding = <cadena> (user)`: define el juego de caracteres del lado del software cliente. Generalmente, el valor utilizado es el del juego de caracteres de la base de datos. Una transposición entre los juegos de caracteres es posible con algunas condiciones, pero generalmente no se recomienda.

k. Administración de las modificaciones de la configuración

La mayor parte de las modificaciones descritas con anterioridad son modificadas directamente en el archivo `postgresql.conf`; después el administrador actúa sobre la instancia cuando desee para activar las modificaciones.

Modificación interactiva

El comando `SET` permite modificar un argumento de configuración para la sesión actual, para todos los argumentos `superuser` y `user`. Las modificaciones no son válidas hasta el final de la sesión y solo en ella, nunca en las sesiones concurrentes. Esto permite autorizar comportamientos diferentes, por ejemplo para los argumentos de memoria o de rendimiento:

```
> SET work_mem = 2GB;
> SET random_page_cost = 3;
```

El comando `SHOW` permite controlar en la sesión el valor actual:

```
> SHOW work_mem;
work_mem
----------
2GB
(1 row)
```

Modificación permanente

Algunos argumentos se pueden aplicar a elementos concretos, como un espacio de tabla, una tabla o una columna. Entonces, el valor del argumento en el archivo de configuración se sobrecarga y se presenta en el catálogo de sistema de PostgreSQL como metadatos del objeto afectado.

Por ejemplo, es posible modificar la profundidad de las estadísticas de una columna únicamente con el argumento `default_statistic_target`. El siguiente comando registra el valor deseado:

```
ALTER TABLE prestaciones ALTER COLUMN prest_fecha_inicio
SET STATISTICS 200;
```

Las modificaciones también se pueden escribir en el archivo de configuración con un comando SQL, lo que permite una flexibilidad diferente en la administración. Entonces, las modificaciones se escriben en el archivo `postgresql.auto.conf`, que después se lee `postgresql.conf` y lo sobrecarga. El comando `ALTER SYSTEM` implementa este mecanismo:

```
# ALTER SYSTEM SET work_mem = '128MB';
ALTER SYSTEM
# ALTER SYSTEM SET default_statistic_target = 150;
ALTER SYSTEM
```

Las modificaciones se escriben en el archivo de configuración dedicado:

```
cat postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by ALTER SYSTEM command.
work_mem = '128MB'
default_statistic_target = '150'
```

El administrador actúa entonces sobre la instancia cuando desee para activar las modificaciones.

2. Catálogo de sistema

PostgreSQL almacena la información relativa al funcionamiento de una instancia en un espacio de nombres concreto: `pg_catalog`. Este espacio de nombres en cada base de datos permite encontrar los metadatos de las bases de datos y de los datos de ejecución.

Vistas y tablas de la información del servidor

Existen algunas tablas y vistas interesantes, que permiten explotar algunos datos de las tablas. La siguiente lista reúne algunas tablas:

- `pg_class`: información de las tablas, índices, secuencias y vistas.
  - La columna `relname` se corresponde con el nombre del objeto y `oid` con su identificador.
  - La columna `relnamespace` contiene el identificador del espacio de nombres del objeto y `reltablespace`, el del espacio de tablas.
  - Las columnas `relpages` y `retuples` indican respectivamente el número de bloques de 8 kilobytes ocupados por el objeto y el número de registros en el objeto; estos datos se extraen por el colector de estadísticas y solo son exactas en función de estos datos recogidos.
  - La columna `relfilenode` indica el nombre del archivo del objeto, almacenado en el directorio de la base de datos. Este nombre puede ser idéntico al identificador de la tabla.
  - La columna `relkind` indica la naturaleza del objeto: `r`= tabla ordinaria, `t` = tabla TOAST, `i` = índice, s = secuencia, `v` = vista, `m` = vista materializada, `p` = tabla particionada, `f` = tabla extranjera.
  - La columna `relpersistence` indica la persistencia del objeto: `p` = tabla permanente, `u` = tabla no trazada, `t` = tabla temporal.
  - La columna `relhaspkey` indica la presencia de una clave primaria en la tabla, `relhasindex` indica la presencia de índices.
  - La columna `relacl` contiene las reglas de acceso definidas por la sentencia `GRANT`.
- `pg_database`: información sobre las bases de datos de la instancia. La columna `datname` contiene el nombre de la base de datos y la columna `oid` contiene el identificador de la base de datos, que se corresponde con el nombre del directorio de la base de datos en disco. La columna `datdba` identifica al propietario de la base y la columna `encoding` identifica el juego de caracteres utilizado.
- `pg_namespace`: información sobre los espacios de nombres. La columna `nspname` contiene el nombre del espacio de nombres y la columna `oid` contiene el identificador.
- `pg_tablespace`: lista de los espacios de tablas de la instancia. La columna `spcname` contiene el nombre del espacio de tablas y la columna `oid` contiene el identificador de este espacio. La columna `spclocation` indica la ruta de almacenamiento sobre el sistema de archivo.
- `pg_attribute`: lista las columnas de las tablas, se presenta su nombre (`attname`), sentencia en la tabla (`attnum`), tipo de datos (`atttypid`) y la tabla en la que se halla la columna (`attrelid`).

- pg_index: lista los índices de la base de datos, con la tabla a la que están asociados (indrelid), si son únicos (indisunique) o asociados a una clave primaria (indisprimary), el número de atributos (indnatts) y la lista de los atributos de la tabla a la que están asociados (indkey).
- pg_proc: lista las funciones existentes en la base de datos, con el nombre (proname), el lenguaje (prolang) y el código fuente (prosrc) en los casos de un lenguaje interpretado como plpgsql.
- pg_depend: lista las dependencias entre objetos, por ejemplo entre una vista y una tabla.

La siguiente lista reúne las vistas de sistema:

- pg_roles: lista los roles de la instancia. La columna rolname contiene el nombre del rol, y la columna oid, el identificador. Las otras columnas contienen las propiedades correspondientes a los roles.
- pg_locks: lista los bloqueos adquiridos por las consultas actuales, identificadas por el pid del proceso que las ejecuta. Cada bloqueo se identifica por su modo (compartido, exclusivo, en un registro o una tabla), el hecho de ser adquiridos o no (granted). Vemos la tabla sobre la que se presenta el bloqueo en el atributo relation, así como la página y la tupla tuple afectadas.
- pg_stat_activity: lista las conexiones actuales y, por lo tanto, los procesos de la instancia, con las consultas en curso:
  - pid: identificador del proceso en segundo plano, que ejecuta las consultas.
  - datname: nombre de la base de datos donde se abre la conexión y datid para el identificador de esta base.
  - usename: nombre del usuario y usesysid para el identificador de este usuario.
  - application_name: nombre de la aplicación cliente.
  - state: estado del proceso, entre active (activo), idle (inactivo) o idle in transaction (inactivo en una transacción abierta).
  - query: la consulta en curso cuando el proceso está activo. La última consulta en el resto de los casos.
  - backend_start: fecha y hora del arranque del proceso.
  - query_start: fecha y hora del arranque de la consulta o de la última consulta.
  - state_change: fecha y hora del último cambio de estado.
  - wait_event_type: el tipo de evento que el proceso espera, llegado el caso. NULL si el proceso no está en espera. Los tipo de evento son: LWLock (bloqueo ligero, generalmente de los datos en memoria), Lock (bloqueos correspondientes a los datos visibles desde un punto de vista lógico), BufferPin (bloque de datos), IO (operación de entrada/salida del disco), Client (actividad de la aplicación cliente), IPC (actividad de otro proceso de la instancia PostgreSQL), Timeout (espera la expiración de un plazo), Activity (actividad normal de un proceso de la instancia PostgreSQL) y Extension (actividad de un módulo de extensión).
  - wait_event: nombre del evento en espera para el proceso. NULL si el proceso no está en espera. El nombre depende del tipo de evento (wait_event_type).

- backend_type: desde de la versión 10 de PostgreSQL, indica el tipo de proceso entre los siguientes valores: client backend para los procesos que tratan las consultas SQL de las aplicaciones, autovacuum launcher, autovacuum worker, background worker, background writer, checkpointer, walwriter y startup para los procesos correspondientes a los componentes técnicos de PostgreSQL, después walreceiver y walsender para la administración de la replicación.

El siguiente ejemplo muestra la actividad sobre una instancia PostgreSQL que tiene algunas consultas en curso o ya pasadas, filtrando para solo tener los procesos correspondientes a las consultas SQL de las aplicaciones. El bloqueo en espera sobre la actualización de la tabla clientes aparece claramente para el proceso 8638, aquí generado por la transacción en curso del proceso 2891:

```
clientes=# select datname, application_name, backend_start,
state_change, wait_event_type, wait_event, state, query from
pg_stat_activity where backend_type='client backend'  and pid !=
pg_backend_pid();
-[ RECORD 1 ]----+-----------------------------------------------
pid              | 2891
datname          | clientes
application_name | psql
backend_start    | 2017-08-12 22:57:21.372616+02
state_change     | 2017-08-19 16:30:44.515548+02
wait_event_type  | Client
wait_event       | ClientRead
state            | idle in transaction
query            | update clientes SET cl_nombre = cl_nombre;
-[ RECORD 2 ]----+-----------------------------------------------
pid              | 8638
datname          | clientes
application_name | psql
backend_start    | 2017-08-19 16:27:41.258161+02
state_change     | 2017-08-19 16:30:46.462321+02
wait_event_type  | Lock
wait_event       | transactionid
state            | active
query            | update clientes SET cl_nombre = cl_nombre;
-[ RECORD 3 ]----+-----------------------------------------------
pid              | 8643
datname          | clientes
application_name | DBeaver 4.1.0 - Main
backend_start    | 2017-08-19 16:28:21.166806+02
state_change     | 2017-08-19 16:28:21.173485+02
wait_event_type  | Client
wait_event       | ClientRead
state            | idle
query            | SET application_name = 'DBeaver 4.1.0 - Main'
```

- pg_stat_database: información sobre las bases de datos. La columna datname contiene el nombre de la base de datos. La columna numbackends se corresponde con el número de conexiones abiertas en esta base de datos. Las columnas xact_commit y xact_rollback se corresponden respectivamente con el número de transacciones validadas y anuladas en la base de datos.

- pg_stats: formatea la tabla pg_statistic para hacer legibles los datos, leyendo los datos estadísticos de cada una de las columnas de tablas (schemaname, tablename, attname). Es visible la lista de los valores más frecuentes, con su frecuencia (most_common_vals y most_common_freqs), el histograma de los valores (histogram_bounds) o el tamaño medio (avg_width). El tamaño de las tablas depende del tamaño de las estadísticas recogidas para cada uno de los atributos.
- pg_stat_user_tables: lista los datos de utilización de las tablas de usuario. Esta vista contiene:
  - el identificador,
  - el nombre de la tabla (relid y relname),
  - el nombre del esquema (schemaname),
  - información sobre los recorridos secuenciales y el número de registros leídos por este método (seq_scan y seq_tup_read),
  - el número de recorridos del índice y el número de registros leídos por este método (idx_scan y idx_tup_fetch),
  - el número de inserciones, actualizaciones y eliminaciones realizadas (n_tup_ins, n_tup_upd y n_tup_del).

    Algunas columnas informan de la actividad de los comandos de limpieza y del análisis sobre cada tabla: last_vacuum para la última limpieza, last_autovacuum para la última limpieza automática, last_analyse para el último análisis, last_autoanalyse para el último análisis automático y el contador del número de estas operaciones.

- pg_stat_user_indexes: lista los datos de utilización de los índices. Además de los nombres e identificadores del índice y de la tabla (relid, relname, indexrelid, indexrelname y schemaname), esta vista contiene el número de recorridos que utilizan este índice, el número de entradas de este índice utilizadas y para terminar, el número de registros de la tabla leída (idx_scan, idx_tup_read, idx_tup_fetch).
- pg_statio_user_tables: lista los datos de lectura y escritura sobre las tablas. Esta vista contiene:
  - el nombre y el identificador de la tabla y el nombre del esquema (relid, relname y schemaname),
  - el número de bloques en disco leídos desde esta tabla (heap_blks_read, heap_blks_hit),
  - el número de lecturas de la memoria RAM desde los índices (idx_blks_read, idx_blks_hit),
  - el número de bloques en disco leídos y de lecturas de la memoria RAM con éxito, desde la tabla TOAST (toast_blks_read, toast_blks_hit),
  - el número de bloques en disco leídos y de lecturas de la memoria RAM con éxito desde el índice de la tabla TOAST (tidx_blks_read, tidx_blks_hit).

- `pg_statio_user_index`: lista los datos de lectura y escritura sobre los índices. Además del nombre de la tabla, índice, esquema e identificador de la tabla y del índice (`relid`, `indexrelid`, `relname`, `indexrelname`, `schemaname`), esta vista contiene el número de bloques en disco leídos y el número de lecturas de la memoria RAM con éxito para este índice (`idx_blks_read`, `idx_blks_hit`).

3. Funciones útiles para la explotación

Existen algunas funciones disponibles en el servidor para ayudar al administrador. Estas funciones permiten esencialmente conocer los volúmenes utilizados:

- `pg_relation_size (oid | text)`: esta función acepta como entrada el nombre o el identificador de una relación, por ejemplo una tabla o un índice, y devuelve el tamaño en bytes de esta relación.
- `pg_total_relation_size (oid | text)`: esta función se parece a la anterior, pero también incluye los tamaños de los índices y tablas `TOAST`.
- `pg_database_size (oid | text)`: esta función acepta como entrada el identificador e indica el espacio en disco ocupado por una base de datos, expresada en bytes.
- `pg_tablespace_size (oid | text)`: esta función acepta como entrada un identificador o un nombre de espacio de tablas e indica el espacio en disco utilizado, expresado en bytes.
- `pg_size_pretty(bigint)`: esta función acepta como entrada un entero y lo convierte en una cadena de caracteres, que indica un tamaño legible por un ser humano.

Estas funciones se utilizan en una sentencia `SELECT`, como en los siguientes ejemplos, que muestran el tamaño utilizado por la tabla `accounts` de la base `pgbench`:

```
pgbench=# SELECT pg_relation_size (’accounts’);
pg_relation_size
——————
       134758400
(1 row)
pgbench=# SELECT pg_size_pretty (pg_relation_size
(’accounts’) );
pg_size_pretty
———————
129 MB
(1 row)
pgbench=# SELECT pg_size_pretty (pg_total_relation_size
(’accounts’) );
pg_size_pretty
———————
146 MB
(1 row)
```

Administrador de conexiones

El administrador de conexiones trabaja entre las aplicaciones cliente y servidor de las bases de datos. Su principal tarea es gestionar la apertura de las conexiones al servidor de bases de datos, en lugar de las aplicaciones cliente. Esta administración va a mantener las conexiones abiertas en el servidor de bases de datos y permitir a las aplicaciones cliente reutilizarlas más fácilmente. De hecho, esta administración de un conjunto de conexiones permite ahorrar tiempo de apertura de una conexión al servidor de bases de datos y por lo tanto ahorrar recursos. En efecto, la apertura de una conexión en PostgreSQL implica la creación de un nuevo proceso, que es una operación costosa. Este ahorro de recursos, y en consecuencia de tiempo, permite un mejor escalado de las aplicaciones que utilizan un servidor de bases de datos PostgreSQL.

El administrador de conexiones se puede ejecutar en un sistema independiente de aquel en el que funciona el servidor de bases de datos. No existe regla, sino que la elección se hace en función de la topología existente entre las aplicaciones cliente y los servidores de bases de datos y en función de la resiliencia deseada del conjunto. Por ejemplo, si queremos que el administrador de bases de datos participe en los mecanismos de alta disponibilidad de la aplicación, es oportuno no ejecutarlo en el mismo sistema que el servidor de bases de datos, sino quizás en el sistema donde funcionan las aplicaciones cliente.

De hecho, algunos servidores de aplicaciones integran un administrador de conexiones, como J2E o Node.js, lo que hace inútil la adopción de una herramienta. A la inversa, los lenguajes como PHP no integran esta funcionalidad, por lo que se benefician de estas herramientas.

1. Pgpool

El administrador de conexiones Pgpool es una herramienta externa a PostgreSQL. No se proporciona por el grupo PostgreSQL, pero forma parte de los proyectos del ecosistema PostgreSQL.

El rol de Pgpool es insertarse entre el software cliente y los servidores PostgreSQL para reagrupar las conexiones, optimizando de esta manera el número de conexiones abiertas sobre un servidor PostgreSQL. Un cliente puede conectarse directamente a Pgpool exactamente como si se conectara a un servidor PostgreSQL. El software Pgpool conserva las conexiones abiertas, optimizando de esta manera el tiempo de apertura de las conexiones.

Pgpool permite también gestionar varios servidores PostgreSQL para los casos de fallo. Si el primer servidor PostgreSQL no es accesible, entonces Pgpool cambia las conexiones sobre un segundo servidor. Es posible desencadenar voluntariamente este cambio, permitiendo de esta manera las operaciones de mantenimiento sobre un servidor.

Además, Pgpool se puede utilizar como solución de replicación, enviando exactamente las mismas consultas a dos servidores PostgreSQL. Esta solución está limitada por el hecho de que algunas consultas sean dependientes del estado de un servidor, por ejemplo la hora del servidor, y por lo tanto la replicación puede no ser perfectamente idéntica. Cuando un servidor deja de funcionar, Pgpool continúa funcionando con el servidor restante, pero es necesaria una intervención para reactivar el servidor que ha fallado cuando se restablezca.

Durante la utilización de soluciones de replicación lógicas, por ejemplo Slony o Londiste, Pgpool es capaz de repartir sobre los servidores esclavos las consultas en modo lectura, es decir, todas las que empiecen por `SELECT`. Por supuesto, las consultas en modo lectura que permiten una modificación de los datos por la llamada de procedimientos almacenados impiden esta utilización de Pgpool.

a. Instalación

Pgpool está disponible en forma de paquetes para las distribuciones Debian o Red Hat. Por lo tanto, la instalación por el sistema de paquetes se reduce a los siguientes comandos. En los sistemas Debian:

```
# apt-get install pgpool2
```

Y en los sistemas Red Hat:

```
# yum install pgpool-II-10
```

b. Configuración

La configuración de Pgpool se sitúa en el archivo `/etc/pgpool2/pgpool.conf`. Las diferentes directivas de configuración permiten personalizar el comportamiento de Pgpool. Cada directiva se corresponde con una pareja clave/valor unidos por el carácter =. Las principales directivas de configuración son:

- `listen_addresses`: indica la dirección TCP/IP sobre la que escuchan las conexiones entrantes. El valor * se corresponde con todas las interfaces IP, y una cadena de caracteres vacía, con ninguna conexión TCP/IP. El valor por defecto es `localhost` y se puede sustituir por cualquier dirección IP o el nombre del host válido en el host. Las conexiones a través de un socket Unixe siempre están autorizadas.
- `port`: el número de puerto TCP de escucha de Pgpool.
- `num_init_children`: indica el número de proceso que Pgpool crea durante el arranque. El valor por defecto es 32.
- `max_pool`: indica el número máximo de conexiones que cada proceso de Pgpool puede conservar. Pgpool abre una nueva conexión si la pareja nombre de usuario y base de datos no existe en el conjunto de conexiones abiertas por Pgpool. Por lo tanto, se recomienda que esta cantidad de conexiones sea más grande que el número de combinaciones de pares posibles. En el caso contrario, la conexión más antigua se cierra y se abre una nueva conexión.

  El valor por defecto es 4. El número máximo de conexiones que el servidor PostgreSQL acepta se debe corresponder `con num_init_children * max_pool`.
- `child_life_time`: indica el tiempo después del cual se detiene un proceso hijo inactivo. Por defecto, 300 segundos.
- `child_max_connections`: indica el número máximo que un proceso hijo acepta antes de pararse. Por defecto, el valor 0 desactiva esta posibilidad.
- `connection_life_time`: indica el tiempo después del cual se cierra una conexión a PostgreSQL inactiva. Por defecto, el valor 0 desactiva esta posibilidad.
- `replication_mode`: booleano que indica la utilización del modo de replicación. El valor por defecto es false.
- `replication_strict`: booleano que permite evitar el interbloqueo, esperando el final de una consulta antes de enviarla a otro servidor.
- `master_slave_mode`: booleano que indica la utilización del modo Maestro-Esclavo. El valor por defecto es falso. Este modo es incompatible con el modo de replicación.
- `load_balance_mode`: booleano que permite la repartición de las consultas de lecturas `SELECT`. El valor por defecto es `false`.
- `backend_hostnameN`: nombre del servidor PostgreSQL. La directiva se puede repetir N veces y la cifra N se corresponde con el número del servidor PostgreSQL. La serie empieza en cero (0) y cada vez que se añade un servidor se utiliza el siguiente entero: `backend_hostname0`, `backend_hostname1`, `backend_hostname2`...

- `backend_portN`: número de puerto TCP del servidor PostgreSQL. La cifra N se corresponde con la de la directiva `backend_hostname`.
- `backend_weightN`: peso del servidor PostgreSQL. La cifra N se corresponde con la de la directiva `backend_hostname`.

c. Uso de Pgpool

Una vez arranca la instancia Pgpool, las aplicaciones cliente se deben configurar para conectarse a ella utilizando los datos de conexiones proporcionados.

Por ejemplo, el comando `psql` permite conectarse a la instancia Pgpool. En lugar del nombre del servidor de bases de datos donde funciona la instancia PostgreSQL, el puerto TCP local sobre el que Pgpool escucha es suficiente.

Los modos *Replication* y *Load Balancer* de Pgpool no se recomiendan porque implementan procesos técnicos demasiado básicos y plantean numerosos problemas en su uso.

La replicación entre los servidores PostgreSQL se presenta en el capítulo Replicación. Generalmente se implementa un reparto de cargas eficaz en la aplicación, único lugar en el que se sabe realmente lo que hace la consulta SQL y dónde se debe ejecutar.

d. Configuración y arranque

En un sistema Debian, el archivo de configuración es `/etc/pgpool2/pgpool.conf` y el servicio se llama `pgpool2`. El rearranque, después de las modificaciones de la configuración, se hace con el siguiente comando:

```
systemctl restart pgpool2
```

En un sistema RedHat, el archivo de configuración se crea a partir de un archivo de muestra con el siguiente comando:

```
cp /etc/pgpool-II-10/pgpool.conf.sample /etc/pgpool-II-10/pgpool.conf
```

Después, el servicio se arranca con el siguiente comando:

```
systemctl start pgpool-II-10
```

2. pgBouncer

La herramienta pgBouncer es un administrador de conexiones, más sencillo que Pgpool, porque solo administra las conexiones. Además, utiliza los principios de la programación orientada a eventos, basada en la librería de funciones `libevent`, que es más ligera en términos de ejecución porque un único proceso es suficiente para gestionar el conjunto de consultas y conexiones. Para terminar, y al contrario que Pgpool, puede gestionar varios pools de conexiones con una única instancia, simplificando las arquitecturas complejas.

a. Instalación

La instalación es muy sencilla en sistemas Debian y Red Hat con el paquete ofrecido. En sistemas Windows, el instalador Stack Builder le ofrece en las opciones.

Por lo tanto, la instalación con el sistema de paquetes se reduce a los siguientes comandos. En los sistemas Debian:

```
# apt-get install pgbouncer
```

Y en los sistemas Red Hat:

```
 # yum install pgbouncer
```

El paquete instala el archivo de configuración: `/etc/pgbouncer/pgbouncer.ini`.

La primera sección de este archivo permite configurar las posibles conexiones. La segunda parte permite ajustar las opciones por defecto y el comportamiento de la instancia PgBouncer.

b. Configuración de las conexiones

Cada conexión se identifica por el nombre de la base de datos utilizada para abrir una conexión con pgBouncer, asociada a las coordenadas de conexión a la instancia PostgreSQL. Por ejemplo, para crear una conexión en pgBouncer hacia la base de datos `clientes`, en el servidor `serverpg10`, se puede añadir el siguiente registro en la sección `[databases]` del archivo de configuración:

```
clientes = host=serverpg10 port=5432 dbname=clientes user=clientes
```

El nombre de la base de datos se puede omitir, porque es idéntica al nombre de la conexión. Pero también puede ser diferente, lo que permite de esta manera añadir varias conexiones a bases de datos idénticas, identificadas de diferente manera.

Se pueden añadir algunos argumentos a cada conexión, además de los atributos de conexión habituales:

- `pool_size`: define el número de conexiones hacia PostgreSQL que pgBouncer podrá abrir, sobrecargando el argumento genérico `default_pool_size`.
- `connect_query`: define una consulta SQL llamada en cada apertura de sesión en PostgreSQL.

c. Configuración de la instancia

En la sección `[pgbouncer]` del archivo de configuración, algunas directivas de configuración permiten controlar el comportamiento de la instancia por la autenticación, el número de clientes aceptados y el método de administración de las conexiones. Existen tres métodos de administración de las conexiones controladas por el argumento `pool_mode`. Cada modo permite controlar la manera en la que se utilizan las conexiones a la base de datos PostgreSQL en función de los comportamientos de las aplicaciones cliente:

- `session`: cada sesión abierta desde una aplicación utiliza una sesión abierta al servidor PostgreSQL desde la primera consulta SQL.
- `transaction`: la apertura de una transacción desde una aplicación utiliza una sesión abierta al servidor PostgreSQL justo al final de la transacción.
- `statement`: cada consulta enviada por una aplicación utiliza una sesión a la base de datos PostgreSQL.

Cuando las sesiones a las bases de datos PostgreSQL se liberan, pgBouncer las conserva y se reutilizan tan pronto como otra conexión desde una aplicación lo solicita.

La elección del modo de administración de las conexiones depende mucho de la manera en la que se generan las consultas y las transacciones en las aplicaciones. Por lo tanto, es necesario comprender esta administración para seleccionar el valor adecuado de `pool_mode`.

Hay otros argumentos que permiten controlar pgBouncer:

- `listen_addr`: indica la interfaz de red sobre la que aceptar las conexiones. Por defecto, `127.0.0.1.`; el valor `*` permite escuchar sobre todas las interfaces de red.
- `listen_port`: indica el puerto TCP que se debe escuchar para aceptar las conexiones de las aplicaciones. El valor por defecto es `6432`.
- `auth_type`: indica el tipo de autenticación, por defecto: `trust`. Es posible utilizar una conexión por contraseña.
- `auth_file`: indica el archivo que sirve de base de datos local de usuarios. Las aplicaciones cliente deben utilizar una de las cuentas de usuario, mientras que las conexiones pueden utilizar cuentas específicas a la instancia PostgreSQL. El valor por defecto es `/etc/pgbouncer/userlist.txt`.
- `auth_hba_file`: indica el archivo de reglas de autenticación de los clientes, como el archivo `pg_hba.conf` de PostgreSQL.
- `max_client_conn`: indica el número máximo de conexiones cliente que pgBouncer puede aceptar, todas conexiones a PostgreSQL incluidas. El valor por defecto es `100` y muy habitualmente se debe aumentar.
- `default_pool_size`: indica el número por defecto de conexiones que pgBouncer puede abrir hacia un servidor PostgreSQL para una conexión cliente. El valor por defecto es `20` y se puede sobrecargar por el argumento `pool_size` en la conexión.
- `server_lifetime`: indica el tiempo de vida de una conexión hacia PostgreSQL. El valor por defecto es 1200 segundos.
- `server_idle_timeout`: indica el plazo de inactividad de una conexión hacia PostgreSQL antes de finalizar esta conexión. El valor por defecto es 60 segundos.
- `client_idle_timeout`: indica el plazo de inactividad de una conexión cliente antes de finalizar la conexión. Por defecto, el valor 0 desactiva este plazo.
- `admin_users`, `stats_users`: lista los usuarios que pueden conectarse a la base de datos específica `pgbouncer` para administrar la instancia `pgbouncer` o consultar las estadísticas de funcionamiento.

El archivo de las cuentas de usuario es un sencillo archivo de texto que retoma el formato utilizado por PostgreSQL en las versiones 8. El formato lista un usuario por registro, eventualmente acompañado de su contraseña cifrada; por ejemplo:

```
"clientes" "" ""
"postgres" "" ""
```

d. Arranque

Una vez que se modifican los archivos de configuración, el siguiente comando permite volver a arrancar la instancia PostgreSQL:

```
systemctl restart pgbouncer
```

En los sistemas Red Hat, el arranque se hace con el siguiente comando:

```
systemctl start pgbouncer
```

e. Administración

Después del arranque, son posibles las conexiones a pgBouncer utilizando los argumentos de red indicados en la configuración; por ejemplo:

```
psql -p 6432 -g 127.0.0.1 -U clientes clientes
```

Después, es posible conectarse a la base de datos `pgbouncer` para consultar las estadísticas. A continuación se muestran algunos comandos posibles:

```
psql -p 6432 -h 127.0.0.1 -U postgres pgbouncer
pgbouncer=# show pools;
pgbouncer=# show stats;
```

Estos comandos permiten seguir el comportamiento de pgBouncer, principalmente los recursos que se consumen por las conexiones (`show pools`), el conjunto de los clientes (`show clients`), las conexiones abiertas (`show servers`) y las estadísticas globales (`show stats`).

Un archivo de trazas, `/var/log/postgresql/pgbouncer.log`, permite seguir la actividad de pgBouncer.

Copia de seguridad y restauración

Existen dos métodos de copia de seguridad de los datos con PostgreSQL. La primera es la copia de seguridad lógica. Se centra en el nivel de los objetos contenidos en la base de datos y, por lo tanto, el resultado consiste en un conjunto de instrucciones que permiten reconstruir los objetos y los datos. La segunda es la copia de seguridad física, que se situa en el nivel de los archivos, ignorando completamente el sentido de los datos contenidos, pero que permite una gran finura de restauración, en términos de evolución en el tiempo.

1. Copia de seguridad lógica con pg_dump y pg_dumpall

Las estrategias de copia de seguridad en caliente de las bases de datos obligatoriamente deben tener en cuenta la restauración de estos datos.

Existen dos herramientas para que la copia de seguridad lógica en caliente tenga éxito, cada una con sus particularidades y sus opciones. La elección de la herramienta correcta y las opciones correctas determinan la cualidad y la rapidez de la restauración. Por lo tanto, es importante haber planificado las copias de seguridad correctas, con los formatos correctos y conocer al avance de los métodos de restauración adaptados.

a. pg_dump

`pg_dump` es una herramienta en línea de comandos entregada con PostgreSQL y por lo tanto disponible con todas las versiones del servidor. Es la herramienta más completa y flexible para realizar las copias de seguridad lógicas. La unidad de copia de seguridad normalmente es la base de datos, pero es posible hacer copia de seguridad solo de un esquema, incluso de una única tabla.

La herramienta propone varios formatos de copia de seguridad. Cada una tiene sus ventajas y sus inconvenientes. El formato más clásico es el formato de texto, que, de hecho, es un archivo de comandos SQL que permite recrear los objetos e insertar los datos con consultas SQL clásicas. La ventaja de este formato es una gran flexibilidad para la restauración de los datos durante un cambio de versión de PostgreSQL o, por ejemplo, para utilizar estos datos en otros servidores SGBD. Los formatos binarios, específicos de PostgreSQL, ofrecen más flexibilidad, principalmente durante la inserción de los datos. La elección del formato determina el método de restauración: el formato texto se puede restaurar con la herramienta `psql`, y los formatos binarios, con la herramienta `pg_restore`.

De la misma manera, es posible hacer copia de seguridad de los metadatos o solo de los datos y por defecto el conjunto se guarda.

Puede ser relevante, según el formato de copia de seguridad, insertar los comandos de recreación de los objetos en la copia de seguridad para anticiparse a las restauraciones posteriores. Esta elección solo es interesante con el formato de copia de seguridad texto. En efecto, la herramienta de restauración utilizada para los otros dos formatos permite recrear los objetos.

El comando de copia de seguridad siempre tiene el formato siguiente:

```
pg_dump [opciones] base
```

Las opciones del comando son las siguientes:

- `-F {p|t|c}` o `--format={p|t|c}`:
  - `p`: texto, formato por defecto. La herramienta construye un archivo de comandos SQL que permiten la restauración de los objetos y datos seleccionados.
  - `t`: la herramienta construye un archivo con formato TAR.
  - `d`: la copia de seguridad se escribe en un directorio, idéntico a la extracción de los datos de un archivo TAR.
  - `c`: la herramienta construye un archivo binario comprimido.
- `-Z 0..9` o `--compress=0..9`: indica la tasa de compresión para el formato binario comprimido.
- `-j njobs` o `--jobs=njobs`: número de conexiones abiertas en la instancia por la herramienta para extraer los datos. Esta opción puede reducir el tiempo total de la copia de seguridad, pero tener un impacto sobre el rendimiento de la instancia.
- `-n schema` o `-- schema= schema`: selecciona los objetos contenidos en el esquema. Es posible que la copia de seguridad no sea consistente si existen dependencias hacia otros esquemas.
- `-t table` o `--table=table`: selecciona la tabla indicada.
- `-s` o `--schema-only`: hace copia de seguridad únicamente de las definiciones de los objetos y no de los datos contenidos.
- `-a` o `--data-only`: hace copia de seguridad únicamente de los datos, y no de las definiciones de los objetos.
- `-c` o `--clean`: añade en la copia de seguridad los comandos de eliminación de los objetos antes de su creación.
- `--if-exists`: la utilización de la opción `clean` añade las palabras clave `IF EXISTS` a los comandos de limpieza, minimizando los errores potenciales.

- `-C` o `--create`: añade un comando de creación de la base de datos.
- `--inserts`: en lugar del comando `COPY` por defecto, genera consultas `INSERT` para restaurar los datos.
- `--column-inserts` o `--attribute-inserts`: en lugar del comando `COPY` por defecto, genera consultas `INSERT` con los nombres de las columnas, para restaurar los datos. Esto ralentiza considerablemente la restauración, pero es útil para exportar la base a otro SGBD.
- `--no-tablespaces`: no tiene en cuenta los espacios de tablas en la copia de seguridad, lo que permite una restauración de los datos con una topología física diferente.
- `-f file` o `--file=file`: crea la copia de seguridad en el archivo indicado, y no en la salida estándar.
- `-E codage` o `--encoding=codage`: hace copia de seguridad de los datos en el juego de caracteres indicado, y no en el juego de caracteres de la base de datos.
- `-o` u `--oids`: hace copia de seguridad de los identificadores de los objetos (OID). Esta opción solo se debe utilizar si los identificadores también se usan en las tablas de la aplicación, como en una clave extranjera, por ejemplo. Más allá de este caso, no se debe utilizar.
- `-O` o `--no-owner`: no genera los comandos de cambio de usuario en la copia de seguridad.
- `-x` o `--no-privileges` o `--no-acl`: no hace copia de seguridad de los comandos `GRANT` y `REVOKE`.
- `--use-set-session-authorization`: utiliza el comando `SET SESSION AUTHORIZATION` en la copia de seguridad para la creación de los objetos en lugar de `ALTER OWNER`. Esto es más parecido al estándar SQL, pero hace necesario permisos de superusuario para la restauración.
- `--disable-dollar-quoting`: impide la copia de seguridad de las comillas-símbolos de dólar en el cuerpo de las funciones. Obliga a la utilización de las comillas simples, respetando de esta manera el estándar SQL.
- `--disable-triggers`: esta opción únicamente es útil durante la copia de seguridad de los datos solos. Desactiva los triggers durante la restauración de los datos.
- `-S nombreusuario` o `--superuser=nombreusuario`: especifica el nombre del superusuario para desactivar los triggers.
- `-i` o `--ignore-version`: ignora las diferencias de versión entre la herramienta `pg_dump` y el servidor.
- `-v` o `--verbose`: muestra la información detallada de los objetos guardados, así como las fechas y horas durante la copia de seguridad.

Como para todas las aplicaciones cliente de PostgreSQL, hay disponibles varias opciones para indicar las coordenadas del servidor sobre el que conectarse:

- `-U` o `--user`: nombre de usuario.
- `--role`: nombre del rol utilizado para la creación de la copia de seguridad, útil cuando un superusuario no se puede conectar en remoto a la instancia.
- `-h` o `--host`: nombre del host o dirección IP del servidor2.
- `-p` o `--port`: puerto TCP del servidor.
- `-W` o `--password`: solicita la contraseña.

También es posible utilizar las variables de entorno correspondientes, así como el archivo de contraseñas.

b. pg_dumpall

`pg_dumpall` permite hacer copia de seguridad de todas las bases de datos de una instancia en el formato texto únicamente. Por lo tanto, permite hacer copia de seguridad rápidamente de una instancia sin otra manipulación. También permite hacer copia de seguridad, en forma de consultas, de las definiciones de los roles.

Opciones del comando:

- `-s` o `--schema-only`: hace copia de seguridad únicamente de las definiciones de los objetos, y no de los datos contenidos en ellos.
- `-a` o `--data-only`: hace copia de seguridad únicamente de los datos, y no de las definiciones de los objetos.
- `-c` o `--clean`: añade en la copia de seguridad los comandos de eliminación de los objetos antes de su creación.
- `--inserts`: en lugar del comando `COPY` por defecto, genera las consultas `INSERT` para restaurar los datos.
- `--column-inserts`, `--attribute-inserts`: en lugar del comando `COPY` por defecto, genera las consultas `INSERT`, con los nombres de las columnas, para restaurar los datos. Esto ralentiza considerablemente la restauración, pero es útil para exportar la base a otro SGBD.
- `-g` o `--globals-only`: hace copia de seguridad exclusivamente de los objetos globales, es decir, los roles y las definiciones de los espacios de tablas. Las bases de datos no se guardan. Esta opción es útil para obtener una copia de los usuarios, de los grupos y de los espacios de tablas de una instancia.
- `-r` o `--roles-only`: hace copia de seguridad de las definiciones de los roles. Las bases de datos no se guardan. Esta opción es útil para obtener una copia de los usuarios y de los grupos de una instancia.
- `-t` o `--tablespaces-only`: hace copia de seguridad exclusivamente de las definiciones de los espacios de tablas. Las bases de datos no se guardan. Esta opción es útil para obtener una copia de los espacios de tablas de una instancia.
- `-o` u `--oids`: hace copia de seguridad de los identificadores de los objetos (OID). Esta opción solo se debe utilizar si los identificadores también se usan en las tablas de la aplicación, como en una clave extranjera, por ejemplo. Más allá de este caso, no se debe utilizar.
- `-O` o `--no-owner`: no genera los comandos de cambio de usuario en la copia de seguridad.
- `-x` o `--no-privileges` o `--no-acl`: no hace copia de seguridad de los comandos `GRANT` y `REVOKE`.
- `--use-set-session-authorization`: utiliza el comando `SET SESSION AUTHORIZATION` en la copia de seguridad para la creación de los objetos, en lugar de `ALTER OWNER`. Esto es más parecido al estándar SQL, pero hace necesario permisos de superusuario para la restauración.
- `--disable-triggers`: esta opción únicamente es útil durante la copia de seguridad de los datos solos. Desactiva los triggers durante la restauración de los datos.
- `-S nombreusuario` o `--superuser=nombreusuario`: especifica el nombre del superusuario para desactivar los triggers.

- `--disable-dollar-quoting`: impide la copia de seguridad de las comillas-símbolos de dólar, en el cuerpo de las funciones. También obliga a la utilización de las comillas simples, respetando de esta manera el estándar SQL.
- `-i` o `--ignore-version`: ignora las diferencias de versión entre la herramienta `pg_dumpall` y el servidor.
- `-v` o `--verbose`: muestra información detallada en los objetos guardados, así como las fechas y horas durante la copia de seguridad.

c. Elección de la herramienta

Tabla comparativa entre `pg_dump y pg_dumpall`:

	pg_ dump	**pg_ dumpall**
Formato texto	Sí	Sí
Formato tar	Sí	No
Formato comprimido	Sí	No
Formato directorio	Sí	No
Copia de seguridad en paralelo	Sí	No
Nombre de bases de datos	1	Todas
Selección de un esquema	Sí	No
Selección de una tabla	Sí	No
Metadatos	Sí	Sí
Datos	Sí	Sí
Elección del método de inserción de los datos	Sí	Sí
Limpieza de los objetos	Sí	Sí
Creación de la base de datos	Sí	No
Desactivación de los triggers	Sí	Sí
Copia de seguridad de los roles	No	Sí
Elección del juego de caracteres de la copia de seguridad	Sí	No

Ejemplos

Los siguientes ejemplos hacen copia de seguridad de la base `clientes` en un formato y hacia archivos diferentes.

La copia de seguridad en formato `tar`, genera un archivo compatible con el comando UNIX tar. Por lo tanto, es posible consultar el contenido del archivo:

```
$ pg_dump -F t --file=clientes.tar clientes
$ tar tf wikidb.tar
toc.dat
1843.dat
1846.dat
[...]
1859.dat
1853.dat
restore.sql
```

Los archivos `*.dat` son los datos brutos, en forma de texto, y el archivo `restore.sql` contiene las instrucciones que se deben ejecutar para la restauración. Por lo tanto, es posible modificar este archivo, respetando los nombres de los archivos contenidos. Una copia de seguridad en formato directorio (`d`) genera el mismo contenido.

Ejemplo de copia de seguridad en formato comprimido que no es posible modificar:

```
$ pg_dump -F c --file=clientes.dump clientes
```

La copia de seguridad en formato texto genera un archivo de instrucción SQL que es posible editar:

```
$ pg_dump -F p --file=clientes.sql clientes
```

Los volúmenes ocupados por los archivos de copias de seguridad varían sensiblemente según el formato utilizado.

Como muestra el resultado del siguiente comando, sea cual sea el volumen de datos de la base de datos, el formato comprimido es muy interesante:

```
ls -lh clientes.*
-rw-r--r-- 1 user group 967K nov.  19 19:00 clientes.dump
-rw-r--r-- 1 user group  15M nov.  19 19:00 clientes.sql
-rw-r--r-- 1 user group  15M nov.  19 19:00 clientes.tar
```

Ejemplo de copia de seguridad de los roles de la instancia con la herramienta `pg_dumpall`:

```
$ pg_dumpall -g > pg_global.sql
$ cat pg_global.sql
CREATE ROLE postgres;
ALTER ROLE postgres WITH SUPERUSER INHERIT CREATEROLE CREATEDB
LOGIN PASSWORD 'md561bc27e3c57fe11d91802e53c90df7c6';
$
```

2. Restauración lógica con pg_restore y psql

La restauración de los datos y metadatos se puede realizar de dos maneras diferentes, según las opciones tomadas durante la copia de seguridad. Las copias de seguridad en formato binario se deben restaurar con la herramienta `pg_restore` y las copias de seguridad en formato texto (scripts SQL), con la herramienta `psql`. Por lo tanto, es la elección del formato de copia de seguridad lo que determina la herramienta de restauración.

a. pg_restore

La herramienta `pg_restore` permite restaurar las copias de seguridad binarias con la misma flexibilidad que la herramienta de copia de seguridad `pg_dump`. Si los datos y objetos están disponibles en la copia de seguridad, `pg_restore` va a permitir seleccionar de manera precisa lo que queremos restaurar, utilizando por ejemplo una lista de objetos que hay que restaurar o las opciones para indicar una tabla o un índice.

Opciones del comando:

- `-F formato` o `--format=format`: formato del archivo de copia de seguridad que se va a restaurar. Se puede elegir entre:
  - `-t`: formato tar.
  - `-d`: formato directorio.
  - `-c`: formato binario comprimido.
- `-f nombre_archivo` o `--file=filename`: indica del archivo de salida durante la generación de la lista.
- `-j N` o `--jobs=N`: número de conexiones utilizadas para la restauración.
- `-l` o `--list`: lista el contenido del archivo.
- `-L archivo_lista` o `--use-list=archivo_lista`: utiliza el archivo indicado como índice de los objetos que hay que restaurar.
- `-1` o `--single-transaction`: incluye la restauración de los datos en una transacción.
- `--no-data-for-failed-tables`: no restaura los datos de las tablas cuya creación ha fallado.
- `-a` o `--data-only`: solo permite la restauración de los datos.
- `-s` o `--schema-only`: solo permite la restauración de los metadatos.
- `-c` o `--clean`: elimina los objetos de la base de datos antes de crearlos.
- `-C` o `--create`: crea la base de datos antes de restaurarla.
- `-d nombre_base` o `--dbname=nombre_base`: base de datos en la que restaurar los objetos.
- `-n nombre_esquema` o `--schema=nombre_esquema`: solo permite la restauración de los objetos del esquema indicado.
- `-t tabla` o `--table=tabla`: restaura únicamente la tabla indicada.
- `-I index` o `--index=índice`: restaura únicamente el índice indicado.
- `-T trigger` o `--trigger=trigger`: restaura únicamente el trigger indicado.
- `-P nombre_función(argtipo [, ...])` o `--function=nombre_función(argtipo [, ...])`: restaura únicamente la función indicada.
- `-x` o `--no-privileges o --no-acl`: no restaura los permisos de acceso de los comandos `GRANT` y `REVOKE`.
- `--disable-triggers`: desactiva los triggers durante la restauración de los datos.
- `-S nombre_usuario` o `--superuser=nombre_usuario`: nombre del superusuario para la desactivación de los triggers.

- `-O` o `--no-owner`: no envía los comandos de cambio de usuario para inicializar los propietarios de los objetos restaurados.
- `--use-set-session-authorization`: utiliza los comandos `SET SESSION AUTHORIZATION` para inicializar los propietarios de los objetos en lugar del comando `ALTER OWNER`.
- `-e` o `--exit-on-error`: detiene el proceso de restauración en caso de error.
- `-i` o `--ignore-versión`: ignora las diferencias de versión en los casos de diferencia entre la herramienta `pg_restore` y el servidor.
- `-v` o `--verbose`: muestra más información durante el proceso de restauración.

b. psql

La herramienta `psql` no es únicamente una herramienta de restauración. También permite enviar a un servidor los comandos SQL desde un script. Por lo tanto, con `psql` se hace la restauración de los archivos de copia de seguridad en formato texto. Esta herramienta permite leer los comandos desde la entrada estándar o desde el archivo indicado con la opción `-f` o `--file`.

c. Ejemplos

Este ejemplo restaura los metadatos de la copia de seguridad de la base `wikidb` en la base `pruebas`:

```
pg_restore -d pruebas -s < clientes.dump
```

Este otro ejemplo envía los comandos SQL contenidos en el archivo `wikidb.dump` a la entrada estándar del cliente `psql`, que se conecta a la base `pruebas` para restaurar todo lo que haya en el archivo:

```
cat clientes.sql | psql pruebas
```

También es posible combinar directamente las herramientas de copia de seguridad y restauración para mover los datos de una instancia a otra. En el siguiente ejemplo, se utiliza `pg_dumpall` junto a `psql` para realizar una copia del servidor `servidor1` al servidor `servidor2`:

```
pg_dumpall -U postgres -h servidor1 | psql -U postgres
-h servidor2
```

3. Copia de seguridad física

Además de estos métodos de copia de seguridad lógica de las bases de datos, existen otros métodos, más cercanos a una copia de seguridad de archivo.

a. Copia de seguridad en frío

El primer método consiste en copiar los archivos del directorio `DATA` después de haber detenido correctamente PostgreSQL. Una vez terminada esta copia, es suficiente con arrancar PostgreSQL. Este método impone un corte del servicio y, por lo tanto, la no disponibilidad de los datos durante el tiempo de la copia.

La restauración consiste simplemente en reemplazar el directorio de los datos con el directorio de copia de seguridad cuando el servidor se detiene, después de arrancar en esta antigua versión de los datos.

b. Copia de seguridad sobre la marcha

El segundo método no implica ningún corte del servicio. Las funciones `pg_start_backup()` y `pg_stop_backup()` de PostgreSQL permiten copiar el directorio `DATA` de forma que el servidor continúe funcionando.

Los archivos de traza binarios (`WAL`) se deben archivar con la directiva de configuración `archive_command` en el archivo `postgresql.conf`. Este comando permite copiar los segmentos de traza binarios para las copias de seguridad. Gracias a estos archivos que contienen todas las transacciones de la instancia, va a ser posible reconstruir todas estas transacciones cuando la restauración sea necesaria.

El valor de la directiva `archive_command` es una línea de comandos del sistema operativo que permite la copia o movimiento de los archivos. Existen dos cadenas de caracteres genéricas que permiten identificar el segmento que hay que copiar: `%f` y `%p`, que se corresponden respectivamente con el nombre del segmento único y con la ruta absoluta del segmento, incluyendo su nombre.

Este ejemplo permite copiar el archivo en un directorio de almacenamiento utilizando el comando `cp` de Unix:

```
archive_command = 'cp %p /PG_WAL/%f '
```

Durante el funcionamiento normal de PostgreSQL, se rellenan los archivos de traza de transacciones. Para cada segmento con un tamaño de 16 megabytes, PostgreSQL crea un nuevo archivo cuando el segmento llega a ese tamaño. La directiva `archive_command` permite una copia del archivo antiguo cuando cambia el segmento.

Una vez que se establece esta configuración, es suficiente con llamar a la función `pg_start_backup()` desde cualquier base de datos de la instancia:

```
postgres# SELECT pg_start_backup('label' );
```

Esta función ejecuta un `CHECKPOINT` que hace que los datos en disco sean coherentes para la copia de seguridad. Además, crea un archivo `backup_label` que contiene la posición interna a los archivos de traza de transacciones, correspondiente al `CHECKPOINT`.

Después, para realizar la copia de seguridad, es suficiente con copiar el directorio `PGDATA` sin detener el servidor, por ejemplo con la herramienta `rsync`. El directorio `pg_xlog`, que contiene los archivos de traza de transacciones, generalmente se excluye de la copia de seguridad porque es el argumento `archive_command` el que se encarga de archivar los archivos de traza de transacciones.

La llamada a la función `pg_stop_backup()` indica a PostgreSQL el final de la copia de seguridad:

```
postgres# SELECT pg_stop_backup();
```

Hay numerosas herramientas externas de PostgreSQL que implementan este tipo de copia de seguridad, simplificando el establecimiento de las estrategias de copia de seguridad y restauración. Se pueden listar algunas herramientas:

- `pgbarman`: escrito en Python, esta herramienta implementa una estrategia de copia de seguridad y restauración de las copias de seguridad físicas, basadas en las herramientas `rsync` y `ssh`, así como `pg_basebackup` y `pg_receivewal`, que permiten una copia de seguridad a través de las conexiones de red seguras. La herramienta está disponible en la siguiente dirección: http://www.pgbarman.org/
- `wal-e`: esta herramienta se centra en la posibilidad de archivar y hacer copia de seguridad de los datos de las instancias en S3, solución de almacenamiento en línea de Amazon. La herramienta está disponible en la dirección: https://github.com/wal-e/wal-e
- `pgbackrest`: escrito en Perl, esta herramienta no depende de otras herramientas e implementa un protocolo de copia de seguridad que permite el trabajo en paralelo durante la copia, la compresión de las copias de seguridad y la verificación de las sumas de control de las páginas. Además, durante la restauración de una instancia, es posible restaurar sobrecargando una instancia existente o restaurar solo una de las bases de datos de la instancia que hay que restaurar.

El comando pg_basebackup

PostgreSQL proporciona el comando `pg_basebackup`. Permite hacer una copia de seguridad física sencilla y eficaz, basándose en una herramienta como `rsync`, pero utilizando el protocolo de replicación integrado en PostgreSQL para conectarse a una instancia remota y crear el directorio de datos localmente.

Los argumentos del comando son los siguientes:

- `-D directory` o `--pgdata=directory`: directorio de copia de seguridad de los datos. El directorio puede no existir. En estos casos `pg_basepackup` lo creará, pero, si existe, debe estar vacío. Esta opción es obligatoria.
- `-F formato` o `-format=formato`: indica el formato de salida, entre:
  - `p` o `plain`: este formato crea una arborescencia de directorios y archivos idéntica a la del directorio original. Los espacios de tablas se dejan en las mismas rutas absolutas que en el sistema original. Es el valor por defecto del argumento.
  - `t` o `tar`: este formato escribe el conjunto de los datos en un archivo `TAR`, en el directorio de copia de seguridad. El directorio principal se escribe en un archivo `base.tar` y los espacios de tablas se escriben en un archivo `TAR`, cuyo nombre es el identificador OID del espacio de tablas.
- `-r rate` o `--max-rate=rate`: tasa de transferencia máxima de los datos desde la instancia PostgreSQL. Los valores se indican en kilobytes por segundo. El interés es limitar el impacto de la copia de seguridad en el sistema.
- `-R` o `--write-recovery-conf`: crea un archivo `recovery.conf` en el directorio de copia de seguridad, utilizando la información de conexión necesaria para arrancar un servidor «standby».
- `-T olddir=newdir` o `--tablespace-mapping=olddir=newdir`: mueve un espacio de tablas a un nuevo directorio para permitir no depender de las rutas absolutas del sistema original.
- `--waldir=waldir`: permite indicar una ruta absoluta de copia de seguridad de los archivos de traza de transacciones.
- `-X method` o `--xlog-method=method`: indica el método de copia de seguridad de los archivos de traza de transacciones, entre:
  - `n` o `none`: no incluye los archivos de traza de transacciones en la copia de seguridad. Es útil si está implementado el almacenamiento de los archivos de traza de transacciones.

    - `f` o `fetch`: los archivos de traza de transacciones se recogen al final de la copia de seguridad si todavía no se han archivado.
    - `s` o `stream`: crea un flujo de replicación únicamente para los archivos de traza de transacciones. Este método utiliza una conexión adicional, pero, habiendo sido realizada continuamente durante la copia de seguridad, todos los archivos de traza de transacciones usados durante la copia de seguridad también se guardan. Se trata del comportamiento por defecto.
- `-z` o `--gzip`: activa la compresión del archivo TAR.
- `-Z level` o `--compress=level`: indica el nivel de compresión del archivo TAR, entre 0 y 9.
- `-c fast|spread` o `--checkpoint=fast|spread`: indica el método de `CHECKPOINT` realizado al inicio del backup.
- `-l label` o `--label=label`: define la etiqueta del backup para averiguar el archivo `backup_label`. Por defecto se utiliza «`pg_basebackup base backup`».
- `-P` o `--progress`: muestra una barra de progreso durante la copia de seguridad.
- `-v` o `--verbose`: activa un modo de información durante la copia de seguridad.
- `-d connstr` o `--dbname=connstr`: cadena de conexión al servidor.
- `-h host` o `--host=host`: indica el nombre del servidor.
- `-p puerto` o `--port=port`: indica el puerto TCP del servidor.
- `-U username` o `--username=username`: indica el nombre de usuario.
- `-w` o `--no-password`: fuerza a `pg_basebackup` a no pedir contraseña. Si se requiere una contraseña y no se encuentra ningún archivo de contraseñas, la conexión falla.
- `-W` o`--password`: fuerza a `pg_basebackup` a pedir una contraseña.

El siguiente ejemplo realiza una copia de seguridad en formato TAR, haciendo copia de seguridad de los archivos de traza de transacciones y creando un archivo `recovery.conf`:

```
$ pg_basebackup -h servidorpg10 -U postgres -X s -F t -R -D
backup20170827
```

El directorio `backup20170827` contiene entonces un archivo `base.tar`:

```
$ ls -sh backup20170827
127M base.tar
```

El comando pg_receivewal

PostgreSQL proporciona el comando `pg_receivewal`. Se trata de un programa residente (demonio) que permite recuperar el conjunto de archivos de traza de transacciones de una instancia PostgreSQL y duplicarlo en un directorio dado. Este comando se comporta como si se tratara de una instancia «standby». Utiliza el protocolo cliente/servidor de PostgreSQL para conectarse. En términos de funcionalidad, sustituye al comando de almacenamiento establecido en la directiva de configuración `archive_command`, incluso es posible poner en marcha los dos métodos. El comando se lanza en el mismo sistema en el que queremos conservar los archivos de traza de transacciones. Hace necesario abrir una conexión con la instancia PostgreSQL. Por defecto, este comando utiliza un «slot» de replicación para asegurarse de no perder transacciones.

Los argumentos del comando son los siguientes:

- `-D directory` o `--directory=directory`: directorio de almacenamiento de los archivos de traza de transacciones. Esta opción es obligatoria.
- `-d conn`: cadena de conexión a la instancia PostgreSQL.
- `-Z level`: de 0 a 9, indica la tasa de compresión de los archivos de transacciones.
- `-S nombre` o `--slot=nombre`: indica el nombre del «slot» que se debe utilizar en la instancia PostgreSQL. Este «slot» debe existir y se puede crear con la opción `--create-slot` del comando.
- `--synchronous`: indica que el comando debe sincronizar los datos en el disco durante su recepción, así como informar a la instancia PostgreSQL. Esta opción es útil si la replicación de los datos de este comando es síncrona, lo que no es el caso por defecto.
- `-n` o `--no-loop`: en caso de error de conexión a la instancia PostgreSQL, indica al comando que no lo reintente, sino que salga produciendo un error. Por defecto, el comando reintenta la conexión hasta el infinito.

El siguiente ejemplo permite archivar los archivos de traza de transacciones, comprimiéndolos:

```
$ pg_receivewal -d "host=servidorpg10 user=postgres" -Z 7 -D
/backups/wal_archive/
```

El comando no devuelve el control, sino que actúa permanentemente. Se debe integrar en un mecanismo de administración de los servicios para asegurar la calidad del servicio. La herramienta pgBarman lo utiliza por defecto cuando es posible.

c. Restauración de una copia de seguridad sobre la marcha

La restauración de una copia de seguridad que utiliza este método necesita dos etapas. En primer lugar, se vuelve a copiar el directorio `PGDATA` en la ubicación en la que PostgreSQL lo espera; después es necesaria la creación de un archivo `recovery.conf`.

Este archivo `recovery.conf` permite a PostgreSQL copiar los archivos de traza de transacciones archivados con antelación y reconstruir su contenido, a partir de la última transacción validada del directorio `PGDATA`, hasta la última transacción registrada en el archivo de trazas.

Se presenta una muestra de este archivo, `recovery.conf.sample`, en el directorio share de la instalación de PostgreSQL. Hay que copiar este archivo en el directorio `PGDATA` y renombrarlo como `recovery.conf`.

El archivo contiene una directiva `restore_command` que, a la inversa que `archive_command`, permite a PostgreSQL copiar los archivos de traza archivados, como en el ejemplo siguiente:

```
restore_command = 'cp /PG_WAL/%f %p'
```

PITR: Point in Time Recovery

Esta técnica de restauración permite poner la instancia restaurada en un estado diferente al de por defecto sin recrear todas las transacciones. Esto es particularmente útil cuando se han perdido datos como consecuencia de un comando como `DROP TABLE` o `TRUNCATE`.

Hay dos directivas que permiten indicar el momento del final de la restauración. De esta manera, las transacciones situadas entre este momento y el final del registro no se recrearán. La directiva `recovery_target_time` fija una fecha y una hora límites que no hay que sobrepasar:

```
recovery_target_time = '2017-08-27 14:45:00 EST'
```

La directiva `recovery_target_xid` permite indicar un identificador de transacción como valor límite. La directiva `recovery_target_inclusive` permite incluir o no esta transacción:

```
recovery_target_xid = '1100842'
recovery_target_inclusive = 'true | false'
```

La directiva `recovery_target_name` permite indicar una etiqueta como valor límite. Esta etiqueta se ubica en los archivos de traza de transacciones con la función `pg_create_restore point(label)` en el momento oportuno, por ejemplo durante una modificación en una base de datos:

```
SELECT pg_create_restore_point('LABEL_201708271606');
```

Después, en el archivo recovery.conf:

```
 recovery_target_name = 'LABEL_201708271606'
```

A continuación, es suficiente con arrancar PostgreSQL con el comando `pg_ctl` para restaurar los datos. El comando `pg_ctl` utiliza el directorio `PGDATA`, que contiene el archivo `recovery.conf`.

Una vez que termina la restauración, el archivo se vuelve a llamar `recovery.done`, prohibiendo de esta manera un intento de restauración durante el siguiente arranque de PostgreSQL.

d. La herramienta pgBackRest

La herramienta pgBackRest permite gestionar las copias de seguridad y las restauraciones de instancias PostgreSQL utilizando técnicas internas de PostgreSQL para las copias de seguridad físicas en caliente y la restauración PITR. La herramienta se escribe en Perl y no depende de otras herramientas. Implementa un protocolo de copia de seguridad que permite el trabajo en paralelo durante la copia, la compresión de las copias de seguridad, las copias de seguridad completas, diferenciales o incrementales, la verificación de las sumas de control de las páginas y el establecimiento de una política de retención de las copias de seguridad y de los archivos de traza de transacciones. Además, durante la restauración de una instancia, es posible restaurar sobrecargando una instancia existente o restaurar solo una de las bases de datos de la instancia que hay que restaurar.

Esta herramienta se basa en `ssh` para los comandos y la copia de los datos. Por lo tanto, es necesario establecer claves `ssh` entre las cuentas de usuario de los diferentes sistemas. Existe un paquete para los sistemas Debian y RedHat en los almacenes ya presentados. El paquete se debe instalar en todos los sistemas implicados: el sistema en el que se ejecuta el servidor PostgreSQL, el sistema donde se almacena la copia de seguridad y el sistema donde se restaura la copia de seguridad. Además, la versión de la herramienta debe ser idéntica en todos los sistemas.

Para un sistema Debian, los comandos de instalación son los siguientes:

```
apt-get install pgbackrest
```

para un sistema RedHat:

```
yum install pgbackrest
```

Archivos de configuración

En los archivos de configuración, las instancias PostgreSQL se identifican con el nombre de stanza y se corresponden con una sección de un archivo de configuración. Por defecto, el archivo de configuración es `/etc/pgbackrest/pgbackrest.conf`.

En el sistema donde se ejecuta la instancia PostgreSQL, la herramienta pgbackrest debe conocer la ubicación de las copias de seguridad y, por lo tanto, en el archivo de configuración, esta ubicación se declara como en el siguiente ejemplo:

```
[servidorpg10]
db-path=/var/lib/postgresql/10/main
backup-host=backups.domain.net
backup-user=postgres
```

En la configuración de la instancia PostgreSQL, el almacenamiento de los archivos de traza de transacciones debe utilizar `pgbackrest`:

```
archive_command = '/usr/bin/pgbackrest --stanza=servidorpg10
archive-push %p'
```

En el sistema donde se almacenan las copias de seguridad, el archivo de configuración contiene el conjunto de las instancias PostgreSQL de los que hay que hacer copia de seguridad, así como los argumentos específicos para la administración de las copias de seguridad, como la ubicación de las copias de seguridad y el número de copias de seguridad que se deben conservar:

```
[servidorpg10]
db-host=servidorpg10.domain.net
db-path=/var/lib/postgresql/10/main
db-port=5432
db-user=postgres

[global]
repo-path=/data/pgbackrest
retention-full=2
retention-archive=2
```

Una vez que se escribe este archivo, hay que inicializar la ubicación de las copias de seguridad con el siguiente comando:

```
pgbackrest --stanza=servidorpg10 stanza-create
```

Copia de seguridad y retención

Una vez que se establecen las configuraciones, se debe lanzar de forma regular el comando de copia de seguridad, idealmente desde un planificador de tareas, como CRON. Es posible seleccionar el tipo de copia de seguridad entre:

- `full`: copia de seguridad completa.
- `incr`: copia de seguridad incremental, desde la última copia de seguridad.

- `diff`: copia de seguridad diferencial, desde la última copia de seguridad completa.

El comando es:

```
pgbackrest --stanza=servidorpg10 --type=full backup
```

donde el tipo se modifica según la copia de seguridad deseada.

Una vez hechas las copias de seguridad, normalmente es necesario deshacerse de las copias de seguridad más antiguas. La retención se define en los archivos de configuración y se aplica con el siguiente comando:

```
pgbackrest --stanza=servidorpg10 expire
```

El argumento `retention-full` se corresponde con las copias de seguridad completas, e incluye implícitamente todas las copias de seguridad incrementales y diferenciales correspondientes a esta copia de seguridad completa. El argumento `retention-archive` se corresponde con la retención de los archivos de traza de transacciones, y su valor hace referencia al número de copias de seguridad completas correspondiente a los archivos de traza de transacciones. Distinguiendo estos dos tipos de retención, es posible conservar solo las copias de seguridad, y no los archivos de traza de transacciones, en función de las restricciones de espacio en disco.

La lista de las copias de seguridad realizadas está disponible con el siguiente comando:

```
stanza: servidorpg10
   status: ok
   db (current)
       wal archive min/max (10-1): 000000010000021400000034 /
000000010000023000000080
       full backup: 20170814-010202F
           timestamp start/stop: 2017-08-14 01:02:02 / 2017-08-14
01:17:06
           wal start/stop: 000000010000021400000034 /
00000001000002140000003D
           database size: 1.1GB, backup size: 1.1GB
           repository size: 275.6MB, repository backup size:
275.6MB
```

En este ejemplo, el nombre de la copia de seguridad completa es: `20170814-010202F`

Restauración

La restauración de los datos se hace desde el sistema en el que se restaura la instancia.

Es posible completar un juego de datos existente en los casos en los que la herramienta solo sustituya los datos efectivamente modificados. Se utiliza el nombre de la copia de seguridad en el almacén para identificarla, y las opciones están disponibles para preparar la configuración en vista de PITR.

El siguiente comando permite extraer la copia de seguridad deseada con la opción `--set` después de establecer el punto de restauración, indicando el tipo (`--type`) y el destino (`--target`):

```
pgbackrest --stanza=servidorpg10 --set=20170814-010202F
--db-path=/var/lib/postgresql/10/main --type=name
--target=LABEL_201708271606 restore
```

Entonces los datos se extraen del almacén de copia de seguridad y se crea el archivo de configuración `recovery.conf`, que contiene el destino de la restauración del PITR y el comando que permite extraer los archivos de traza de transacciones del almacén de copia de seguridad:

```
restore_command = '/usr/bin/pgbackrest --stanza=servidorpg10
archive-get %f "%p"'
recovery_target_name = 'LABEL_201708271606'
```

Es posible modificar este archivo antes del arranque de PostgreSQL, por ejemplo en el marco de la puesta en marcha de una instancia «standby». Por otro lado, en los sistemas Debian, se deben restaurar los archivos de configuración situados en `/etc`.

Desde ese momento, la instancia PostgreSQL se arranca y tiene lugar el proceso de restauración de los datos.

La herramienta dispone de un sitio web con una documentación completa sobre su configuración y sus comandos en la dirección: https://www.pgbackrest.org/

e. La herramienta pgBarMan

La herramienta pgBarMan es una herramienta escrita en Python. Utiliza los comandos `rsync`, `ssh`, `pg_basebackup` y `pg_receivewal` para poner en marcha la política de copia de seguridad. Hay un paquete disponible en los almacenes ya presentados; la instalación del paquete solo es útil en el sistema en el que se almacenan los datos. Los comandos de instalación son los siguientes para un sistema Debian:

```
apt-get install barman
```

Para un sistema RedHat:

```
yum install barman
```

El archivo de configuración es `/etc/barman.conf` y define la ubicación de las copias de seguridad con `barman_home` o la política de retención con `retention_policy`.

Los archivos de configuración se corresponden con las instancias PostgreSQL de las que se debe hacer copia de seguridad, se definen en el directorio `/etc/barman.d.` y contienen la información necesaria para conectarse a la instancia PostgreSQL. Por ejemplo, el archivo `/etc/barman.d/servidorpg10.conf`:

```
ssh_command = ssh postgres@servidorpg10.domain.net
conninfo = host=servidorpg10.domain.net user=postgres port=5432
```

Una tarea planificada se lanza normalmente para todas las tareas de mantenimiento normales. Entre otros, esta tarea lanza el demonio `pg_receivewal` para archivar los archivos de traza de transacciones. También es posible configurar el almacenamiento de los archivos de traza de transacciones desde la instancia PostgreSQL, utilizando la siguiente parametrización en la configuración de la instancia PostgreSQL:

```
archive_command = 'rsync -a %p
barman@backup.domain.net:/var/lib/barman/servidorpg10/incoming/%f'
```

Copia de seguridad y retención

Una vez que se han puesto en marcha las configuraciones, se debe lanzar de forma regular el comando de copia de seguridad, idealmente desde un planificador de tareas como CRON. El comando de copia de seguridad es el siguiente:

```
barman backup servidorpg10
```

La herramienta utiliza entonces `pg_basebackup` para hacer la copia de seguridad.

Las copias de seguridad ya realizadas son visibles con el comando:

```
barman list-backup servidorpg10
```

La política de retención se aplica automáticamente cuando se realizan las copias de seguridad.

Restauración

La restauración de una copia de seguridad consiste en extraer los datos desde el almacén de copia de seguridad y exportar los archivos al sistema designado, indicando las opciones para el PITR con el siguiente comando:

```
barman recover --target-time '2017-08-27 17:15' --remote-ssh-command
"ssh postgres@servidor2pg10.domain.net" servidorpg10 20170821T170936
/var/lib/pgsql/10/data
```

Entonces el archivo `recovery.conf` se crea en el directorio y solo falta arrancar la instancia PostgreSQL para proceder a la restauración.

El conjunto de la documentación de esta herramienta está disponible en el siguiente sitio web: http://www.pgbarman.org/

Explotación y tareas de mantenimiento

Una vez instalada la base de datos y conectadas las aplicaciones, es necesario seguir la actividad de la instancia. En efecto, una base de datos evoluciona a lo largo del tiempo, principalmente con las inserciones de datos, las actualizaciones y las eliminaciones de datos.

Es necesario comprender su funcionamiento y actuar en consecuencia.

1. Análisis de una consulta con EXPLAIN

El comando `EXPLAIN` permite estudiar el comportamiento de una consulta y principalmente los diferentes métodos utilizados por PostgreSQL para acceder a los datos.

El plan de ejecución detalla el recorrido de las tablas y, por ejemplo, permite entender dónde añadir un índice en una tabla. También muestra los algoritmos utilizados para los joins y los costes estimados de ejecución, expresados en unidades de recuperación sobre las páginas en disco.

La opción `ANALYZE` ejecuta realmente la consulta y añade el tiempo real de ejecución y el número real de registros devueltos.

La sinopsis del comando es la siguiente:

```
EXPLAIN  <opciones> consulta;
```

donde <opciones> puede ser:

```
( ANALYZE <booleano>, BUFFERS <booleano>, COSTS <booleano>,
TIMING <booleano>,FORMAT { TEXT | XML | JSON | YAML } )
```

- ANALYZE ejecuta realmente la consulta y recoge la información de ejecución.
- BUFFERS añade la información de la memoria RAM de datos utilizada.
- COSTS, visualizada por defecto, muestra la estimación de los costes.
- TIMING muestra los cronómetros de la ejecución.

A continuación se muestra, por ejemplo, una sencilla consulta de lectura con un filtro sobre la clave primaria de la tabla prestaciones. Aquí, la tabla tiene muy pocas tuplas. Por lo tanto, el planificador de consultas no juzga necesario utilizar el índice de la clave primaria y realiza una lectura secuencial de la tabla:

```
clientes=# EXPLAIN SELECT * FROM prestaciones
where prest_id = 32;
                        QUERY PLAN
_____________________________________
Seq Scan on prestaciones  (cost=0.00..1.38 rows=1 width=85)
  Filter: (prest_id = 32)
(2 registros)
```

La misma consulta, con la opción ANALYZE, ejecuta realmente la consulta y devuelve el tiempo de ejecución.

```
clientes=# EXPLAIN (ANALYZE) SELECT * FROM prestaciones
where prest_id = 32;
                                                QUERY PLAN
_____________________________________________________________________
Seq Scan on prestaciones  (cost=0.00..1.38 rows=1 width=85)
(actual time=32.019..32.047 rows=1 loops=1)
  Filter: (prest_id = 32)
Total runtime: 32.121 ms
(3 registros)
```

El comando EXPLAIN sobre una vista permite ver que la consulta SELECT de la vista se ha ejecutado efectivamente:

```
clientes=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM factview;
                                                                    QUERY PLAN
Subquery Scan on factview  (cost=4149.40..4206.60 rows=4576 width=65)
(actual time=111.099..111.911 rows=4568 loops=1)
  Buffers: shared hit=804
  ->  Sort  (cost=4149.40..4160.84 rows=4576 width=97) (actual
time=111.098..111.326 rows=4568 loops=1)
        Sort Key: ("substring"(f.fact_num, '..$'::text))
        Sort Method: quicksort  Memory: 585kB
        Buffers: shared hit=804
        ->  HashAggregate  (cost=3802.54..3871.18 rows=4576 width=97)
(actual time=101.369..108.619 rows=4568 loops=1)
              Group Key: f.fact_num
              Buffers: shared hit=804
              ->  Hash Join  (cost=142.96..3081.60 rows=96125 width=43)
(actual time=1.585..51.630 rows=96125 loops=1)
                    Hash Cond: (l.fact_num = f.fact_num)
                    Buffers: shared hit=804
                    ->  Seq Scan on registros_facturas l  (cost=0.00..1725.25
rows=96125 width=21) (actual time=0.007..13.148 rows=96125 loops=1)
                          Buffers: shared hit=764
                    ->  Hash  (cost=85.76..85.76 rows=4576 width=33) (actual
time=1.567..1.567 rows=4576 loops=1)
                          Buckets: 8192  Batches: 1  Memory Usage: 362kB
                          Buffers: shared hit=40
                          ->  Seq Scan on facturas f  (cost=0.00..85.76
rows=4576 width=33) (actual time=0.004..0.702 rows=4576 loops=1)
                                Buffers: shared hit=40
Planning time: 0.345 ms
Execution time: 112.243 ms
(21 rows)
```

En este comando, la opción `BUFFERS` permite visualizar las estadísticas de las páginas de datos leídos.

En este resultado, la indicación de coste es importante para comprender el funcionamiento de la consulta.

Un coste se corresponde con el número de páginas leídas en el disco, más un coste de lectura de los registros, en función del tratamiento realizado por el procesador, que se corresponde con el número de registros multiplicado por la variable `cpu_tuple_cost`.

Por ejemplo, en el plan anterior, la indicación (`cost=4149.40..4206.60 rows=4576 width=65`) muestra cuatro términos diferentes.

El primer término (`cost=4149:40`) es una estimación heredada de las etapas anteriores, el segundo término (`4206.60`) es la estimación final, después de la etapa implicada. A continuación, el tercer término (`rows=4576`) es una estimación del número de registros implicados y el cuarto término (`width=65`) es una estimación del tamaño medio de los registros.

El segundo conjunto de valores se corresponde con la opción `ANALYSE` y representa las medidas realizadas durante la ejecución de la consulta: (`actual time=111.099..111.911 rows=4568 loops=1`). Aquí, el nodo arranca en `111.099ms` y termina en `111.911ms`, es decir, menos de un milisegundo para esta etapa de la consulta. Se mide el número de registros (`rows=4568`) y se debe comparar con el número de registros estimado: en casos de gran diferencia, generalmente encontramos un problema de estimación de las cantidades de datos que hay que recorrer. Normalmente, las estadísticas se deben recalcular con el comando ANALYSE para solucionarlo.

La lectura de los planes de ejecución es un punto importante de la optimización de las consultas SQL. Hay herramientas que ayudan en esta lectura formateando los planes:

- El sitio web https://explain.depesz.com/ presenta el plan de ejecución en forma de tabla y resalta los puntos críticos gracias a códigos de color. La siguiente copia de pantalla muestra el plan anterior:

#	exclusive	inclusive	rows x	rows	loops	node
1.	0.585	111.911	↑ 1.0	4,568	1	→ Subquery Scan on factview (cost=4,149.40..4,206.60 rows=4,576 width=65) (actual time=111.099..111.911 rows=4,568 loops=1) Buffers: shared hit=804
2.	2.707	111.326	↑ 1.0	4,568	1	→ Sort (cost=4,149.40..4,160.84 rows=4,576 width=97) (actual time=111.098..111.326 rows=4,568 loops=1) Sort Key: ("substring"(f.fact_num, '..$'::text)) Sort Method: quicksort Memory: 585kB Buffers: shared hit=804
3.	56.989	108.619	↑ 1.0	4,568	1	→ HashAggregate (cost=3,802.54..3,871.18 rows=4,576 width=97) (actual time=101.369..108.619 rows=4,568 loops=1) Group Key: f.fact_num Buffers: shared hit=804
4.	36.915	51.630	↑ 1.0	96,125	1	→ Hash Join (cost=142.96..3,081.60 rows=96,125 width=43) (actual time=1.585..51.630 rows=96,125 loops=1) Hash Cond: (l.fact_num = f.fact_num) Buffers: shared hit=804
5.	13.148	13.148	↑ 1.0	96,125	1	→ Seq Scan on lineas_facturas l (cost=0.00..1,725.25 rows=96,125 width=21) (actual time=0.007..13.148 rows=96,125 loops=1) Buffers: shared hit=764
6.	0.865	1.567	↑ 1.0	4,576	1	→ Hash (cost=85.76..85.76 rows=4,576 width=33) (actual time=1.567..1.567 rows=4,576 loops=1) Buckets: 8192 Batches: 1 Memory Usage: 362kB Buffers: shared hit=40
7.	0.702	0.702	↑ 1.0	4,576	1	→ Seq Scan on facturas f (cost=0.00..85.76 rows=4,576 width=33) (actual time=0.004..0.702 rows=4,576 loops=1) Buffers: shared hit=40

- El sitio web Postgres EXPLAIN Visualizer (http://tatiyants.com/pev/#/plans/new) muestra el plan en forma de árbol, pero necesita un plan en formato JSON.

2. Recolección de las estadísticas con ANALYZE

El comando `ANALYZE` permite recoger las estadísticas sobre las tablas y alimentar la tabla `pg_statistic`.

La sinopsis del comando es la siguiente:

```
ANALYZE [ VERBOSE ] [ tabla [ (columna [, ...] ) ] ]
```

La opción `VERBOSE` permite visualizar datos adicionales, principalmente las estadísticas de las tablas.

Por defecto, el análisis se realiza sobre todas las tablas de la base de datos actual, pero también es posible seleccionar una o varias tablas y las columnas de estas tablas.

3. Limpieza de las tablas con VACUUM

El comando `VACUUM` es una herramienta de limpieza de la base de datos. Las dos tareas de este comando son la recuperación del espacio en disco no utilizado y el análisis de las bases de datos. Esta herramienta está disponible en dos formas: un comando SQL, `VACUUM`, y un comando de sistema, `vacuumdb`.

La sinopsis de los comandos SQL es la siguiente:

```
VACUUM [ FULL | FREEZE ] [ VERBOSE ] [ table ]
VACUUM [ FULL | FREEZE ] [ VERBOSE ] ANALYZE [ tabla
[ (columna [, ...] ) ] ]
```

Y para el comando de sistema:

```
vacuumdb [opción-de-conexión...]
   [ --full | -f ]
   [ --verbose | -v ]
   [ --analyze | -z ]
   [[ --table | -t ]  TABLA [( columna [,...] )] ]
[basededatos]

vacuumdb [opciones-de-conexión...]
   [ --all | -a ] [ --full | -f ]
   [ --verbose | -v ] [ --analyze | -z ]
```

Durante las operaciones normales de PostgreSQL, por ejemplo las actualizaciones de registros con `UPDATE` o las eliminaciones con `DELETE`, los registros no se liberan. La limpieza permite esta reutilización.

Además, la opción `FULL` (o `-f`) permite compactar las tablas, liberando el espacio en disco no utilizado. Esta limpieza bloquea la tabla, la hace inaccesible para otras consultas. Esta tarea se puede sustituir por la extensión `pg_repack`, que no bloquea las tablas.

La limpieza normal no plantea un bloqueo exclusivo. Por lo tanto, se puede ejecutar en cualquier momento. Pero la opción `FULL`, que modifica en profundidad las tablas, implica un bloqueo exclusivo sobre la tabla actual, impidiendo la utilización normal de esta tabla. Por lo tanto, esta limpieza completa se debe hacer cuando no interfiera con la utilización normal de las bases de datos.

La opción `FREEZE` permite retirar la referencia a los identificadores de transacciones que hayan manipulado los registros de las tablas, lo que se debe hacer normalmente para no tener problemas de bucles de identificadores de transacciones.

La opción `ANALYZE` (o `-z`) combina la limpieza y el análisis de las tablas, exactamente como hace el comando `ANALYZE`. Como en el comando `ANALYZE`, es posible seleccionar las tablas y las columnas.

El comando de sistema `vacuumdb` permite ejecutar esta limpieza en todas las bases de datos con la opción `-a`, lo que no es posible con el comando SQL.

4. Automatización con AUTOVACUUM

El mecanismo de limpieza automática (`AUTOVACUUM`) permite hacer normalmente las operaciones de limpieza y de análisis de las tablas. Únicamente no hace la limpieza completa (`FULL`) , que se debe realizar de otra manera.

Este proceso lanza las tareas de mantenimiento VACUUM y de recogida de análisis. También lanza las tareas de mantenimiento con el objetivo de protegerse contra el bucle de identificadores de transacciones.

Las tareas de AUTO VACUUM se desencadenan en función de la actividad en las tablas y en la configuración se definen umbrales de actividad. Los umbrales se definen con las directivas `autovacuum_vacuum_scale_factor` y `autovacuum_vacuum_threshold` para las tareas VACUUM, `autovacuum_analyse_scale_factor` y `autovacuum_analyse _threshold` para las tareas de análisis. Generalmente, estas directivas se modifican tabla a tabla porque la cantidad de datos modificados puede ser muy diferente de una tabla a otra.

El siguiente comando permite ajustar los umbrales para la tabla `prestaciones` con objeto de lanzar la tarea de mantenimiento de manera más habitual:

```
alter table prestaciones set ( autovacuum_vacuum_scale_factor = 0.05 );
```

5. Mantenimiento de los índices con REINDEX

El comando `REINDEX`, disponible como comando de sistema bajo el nombre `reindexdb`, permite mantener uno o varios índices cuando están corruptos o simplemente cuando existen datos inútiles en el índice. La sinopsis del comando es la siguiente:

```
REINDEX { INDEX | TABLE | DATABASE | SYSTEM } nombre [ FORCE ]
```

La sinopsis del comando de sistema es la siguiente:

```
reindexdb
   [
       [ [ --table | -t ] tabla ]
       [ [ --index | -i ] índice ]
       [ --system | -s ] nombrebase      ]
   [ --all | -a ]
```

El nombre puede identificar, según la opción utilizada:

- Un índice: en este caso, solo se reconstruye el índice en cuestión.
- Una tabla: en este caso, se reconstruyen todos los índices asociados a esta tabla.
- Una base de datos: en este caso, se reconstruyen todos los índices de la base de datos, salvo los de los catálogos de sistema. Cuando se utiliza la opción `SYSTEM`, son los índices de los catálogos de sistema los que se reconstruyen.

El comando de sistema también puede reconstruir todos los índices de todas las bases de datos con la opción `-a`.

El comportamiento de `REINDEX` es parecido a la eliminación de un índice y a la recreación de este. La diferencia principal es que el comando `REINDEX` solo plantea un bloqueo en modo escritura durante la operación, y no en modo lectura, mientras que un proceso de eliminación y de recreación con los comandos `DROP` y `CREATE` impide la lectura de un índice.

6. Organización de las tablas con CLUSTER

El comando `CLUSTER`, disponible con el comando de sistema `clusterdb`, permite modificar físicamente la manera de almacenar los datos de una tabla basándose en la organización de un índice. El interés aparece cuando la tabla se utiliza en función de un índice.

La sinopsis del comando es la siguiente:

```
CLUSTER [ nombreindice ON ] [ nombretabla ]
```

Y la del comando de sistema:

```
clusterdb
   [
       [ --table | -t ] table ]
       nombrebase
   ]
   [ --all | -a ]
```

La primera llamada de este comando permite agrupar una tabla sobre un índice. En este momento, la tabla se reorganiza y una nueva llamada del comando sobre la tabla reorganiza automáticamente el mismo índice. La opción `-a` del comando de sistema permite agrupar todas las tablas de la instancia.

Probar la instalación con pgbench

La herramienta `pgbench`, que se entrega en las contribuciones de PostgreSQL, permite probar una instalación evaluando un número de transacciones por segundo. Esto permite medir las mejoras aportadas por un cambio en la configuración.

Utiliza una base de datos, en la que crea las tablas y las rellena con datos, cuya cantidad se indica en la inicialización. Después, durante la fase de pruebas, `pgbench` envía las transacciones y cuenta el número de transacciones por segundo en función de diferentes argumentos, como el número de conexiones o el número de transacciones actuales.

Las pruebas se pueden desarrollar en una base de datos dedicada, llamada `pgbench`, creada con el siguiente comando:

```
[postgres] $ createdb pgbench
```

Después, el siguiente comando permite inicializar la base para las pruebas:

```
[postgres] $ pgbench -i -s 10 pgbench
```

La opción `-i` activa la inicialización y la opción `-s` es un factor multiplicador que permite insertar más datos, aquí con un factor 10.

A continuación, se pueden lanzar las pruebas con el siguiente comando:

```
[postgres] $ pgbench -c 10 -t 30 pgbench
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 10
number of clients: 10
number of transaction per client: 30
number of transaction actually processed: 300/300
tps = 112.621490 (including connections establishing)
tps = 113.457239 (excluding connections establishing)
```

La opción `-c` indica el número de conexiones cliente, y la opción `-t`, el número de transacciones por cliente. A continuación, los resultados se muestran, indicando el número de transacciones realizadas por segundo, con o sin el tiempo de apertura de las sesiones. Estos datos pueden servir para calibrar la configuración del servidor.

La opción `-T`, en lugar de `-t`, permite ejecutar la prueba durante un tiempo dado en segundos, y no en número de transacciones.

Explotación de las trazas de actividad con pgBadger

El software pgBadger es un script Perl que permite analizar el contenido de los archivos de trazas de actividad para extraer los datos estadísticos. Esto sirve para evaluar el comportamiento de PostgreSQL, por ejemplo el número de sentencias de inserción o actualización.

1. Instalación

La herramienta se empaqueta en diferentes distribuciones GNU/Linux, pero se puede instalar siguiendo las instrucciones indicadas en su documentación, en la siguiente dirección: http://dalibo.github.io/pgbadger/

El procedimiento de instalación es extremadamente sencillo y solo necesita el comando `perl`, que ya está presente en un sistema GNU/Linux:

```
$ tar xf pgbadger-6.2.tar.gz
$ cd pgbadger-6.2
$ perl Makefile.PL
$ make && sudo make install
```

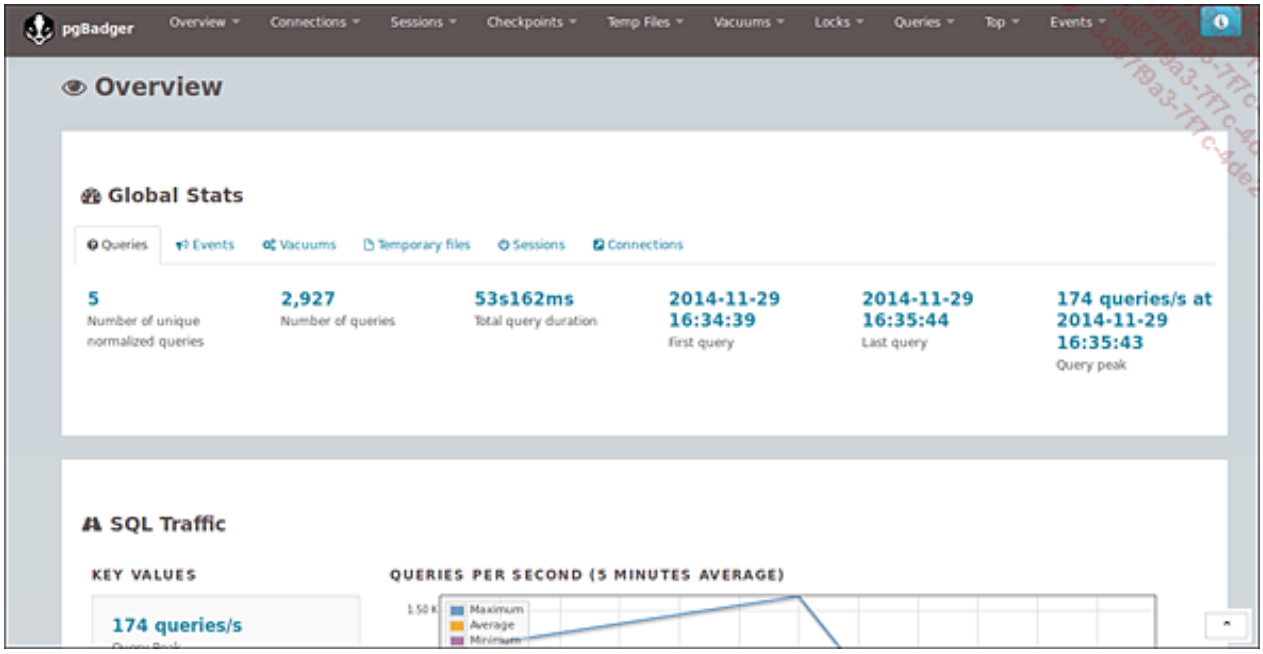
2. Análisis de los archivos de trazas

El archivo de trazas de actividad contiene un determinado número de información, según los ajustes realizados. La herramienta pgBadger puede entender el formato de los archivos de trazas y, por lo tanto, es sencillo de manipular, incluso si el tiempo de análisis de los archivos de trazas puede ser largo y consumir los recursos de la máquina en la que se ejecuta el análisis.

El siguiente comando permite lanzar el análisis y generar un archivo HTML que contiene el informe:

```
pgbadger /var/log/postgresql/postgresql-9.4-main.log -o
informes.html
```

A continuación, el archivo HTML se puede mostrar en un navegador web para leer el informe, como en la siguiente copia de pantalla:



La vista previa del informe da órdenes de magnitud, calculadas a partir del conjunto de consultas encontradas en el archivo de trazas. Numerosos menús permiten acceder a información detallada, como los archivos temporales y errores, así como diferentes clasificaciones de ejecución de las consultas SQL (por ejemplo, los acumulados de tiempo de ejecución más importantes o los que tienen el mayor número de ocurrencias, etc.).

El análisis normal de estos informes permite entender el funcionamiento de la instancia y proceder con las optimizaciones, identificando los puntos que más recursos consumen.

Herramientas

Introducción

PostgreSQL no ofrece herramientas de administración gráfica; únicamente, las herramientas en línea de comandos, como `psql`. Si `psql` es muy eficaz y debe formar parte de las herramientas del DBA no es solo porque siempre está disponible a partir de la instalación de PostgreSQL, sino porque es útil contar con herramientas gráficas, principalmente para los usuarios que desean acceder a los datos sin tener que controlar la línea de comandos

Además, existen herramientas de administración en modo consola, adicionales a la que se entrega con PostgreSQL. Para terminar, las herramientas de monitoring y supervisión permiten integrar la supervisión de PostgreSQL.

La herramienta gráfica: pgAdmin III

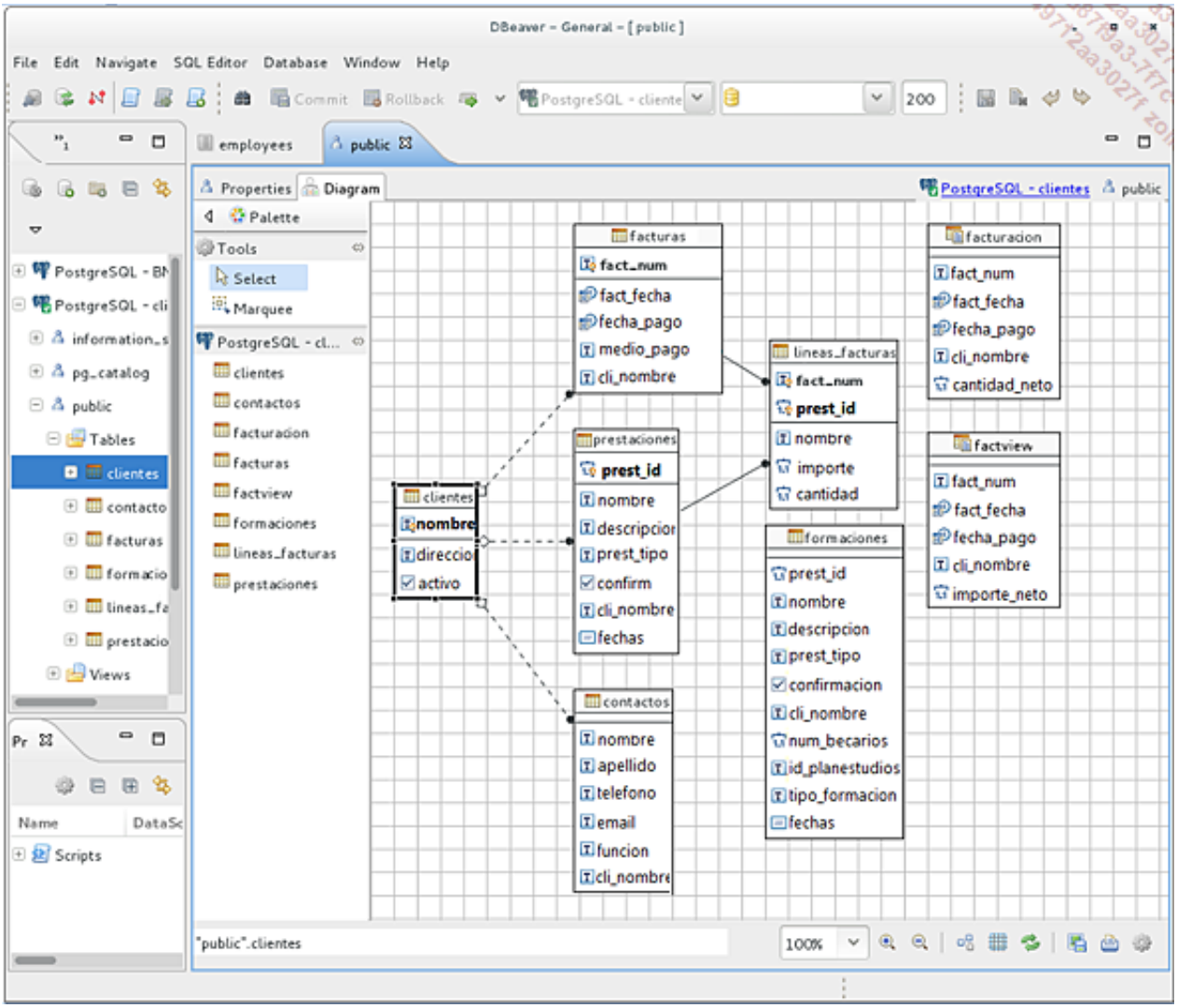
La herramienta pgAdmin III es una herramienta gráfica, históricamente relacionada con PostgreSQL. Desde la versión 10, esta herramienta es abandonada por los desarrolladores en beneficio de la versión PgAdmin IV. Debido a que la reescritura de PgAdmin IV no ha convencido a los usuarios, existe una adaptación de PgAdmin III para PostgreSQL 10 integrada en la distribución pgc únicamente para Windows (ver capítulo Instalación).

La herramienta Dbeaver

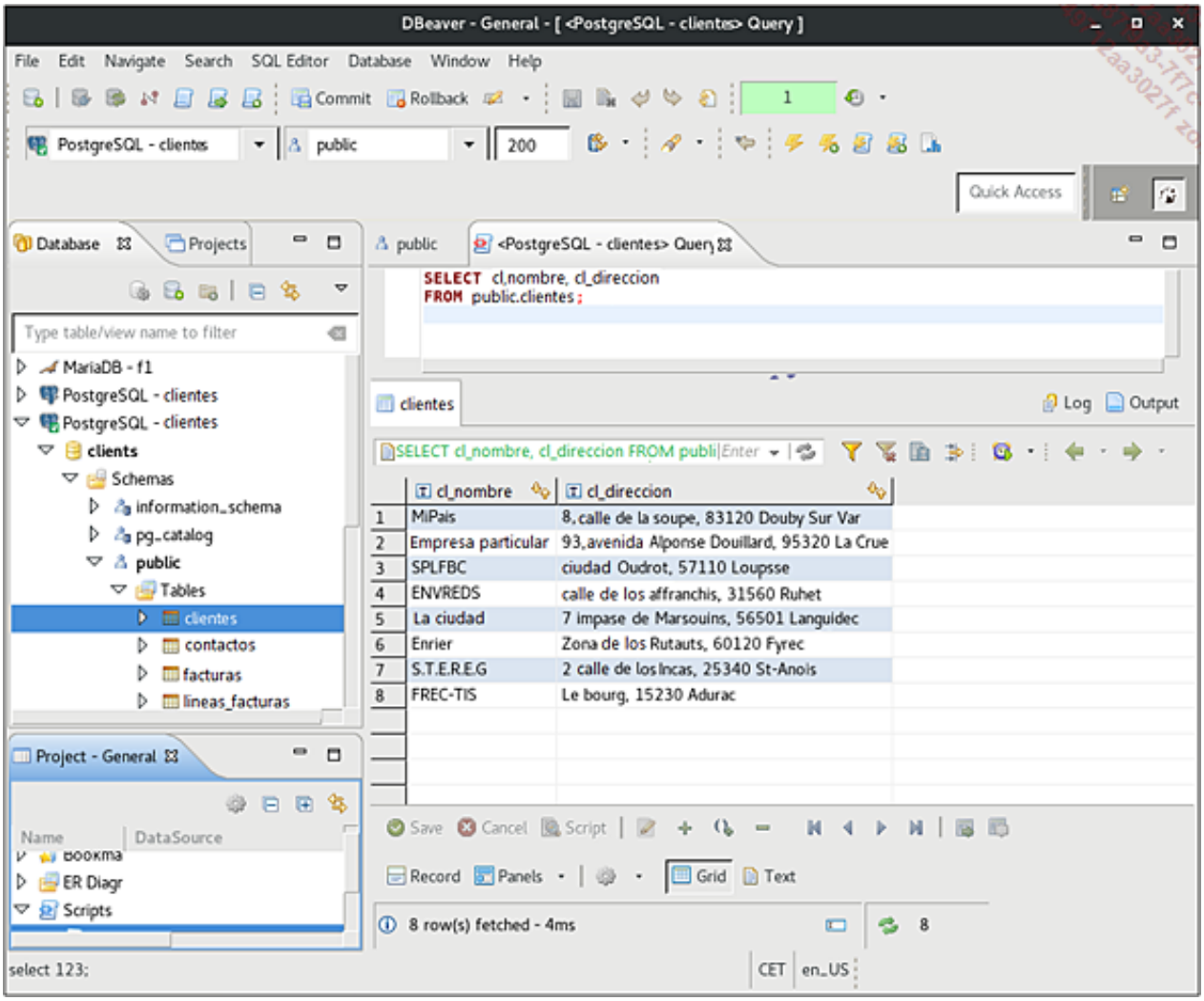
Dbeaver es un cliente SQL universal escrito en Java y basado en Eclipse. Permite conectarse a un gran número de sistemas de bases de datos relacionales, entre ellos PostgreSQL. Utiliza el controlador JDBC para conectarse. Si bien conoce numerosos sistemas de bases de datos, debe descargarse el controlador JDBC durante la configuración inicial de la conexión a PostgreSQL.

La herramienta se descarga en la dirección siguiente: https://dbeaver.jkiss.org/

La herramienta permite representar un modelo gráfico a partir de las tablas y las claves extranjeras existentes:



También permite lanzar consultas desde un editor de consultas SQL. Es posible gestionar las sentencias SQL desde los archivos, cargados desde el administrador de scripts. Los datos mostrados en la cuadrícula son editables directamente y la herramienta gestiona de forma implícita las adiciones y actualizaciones realizadas:

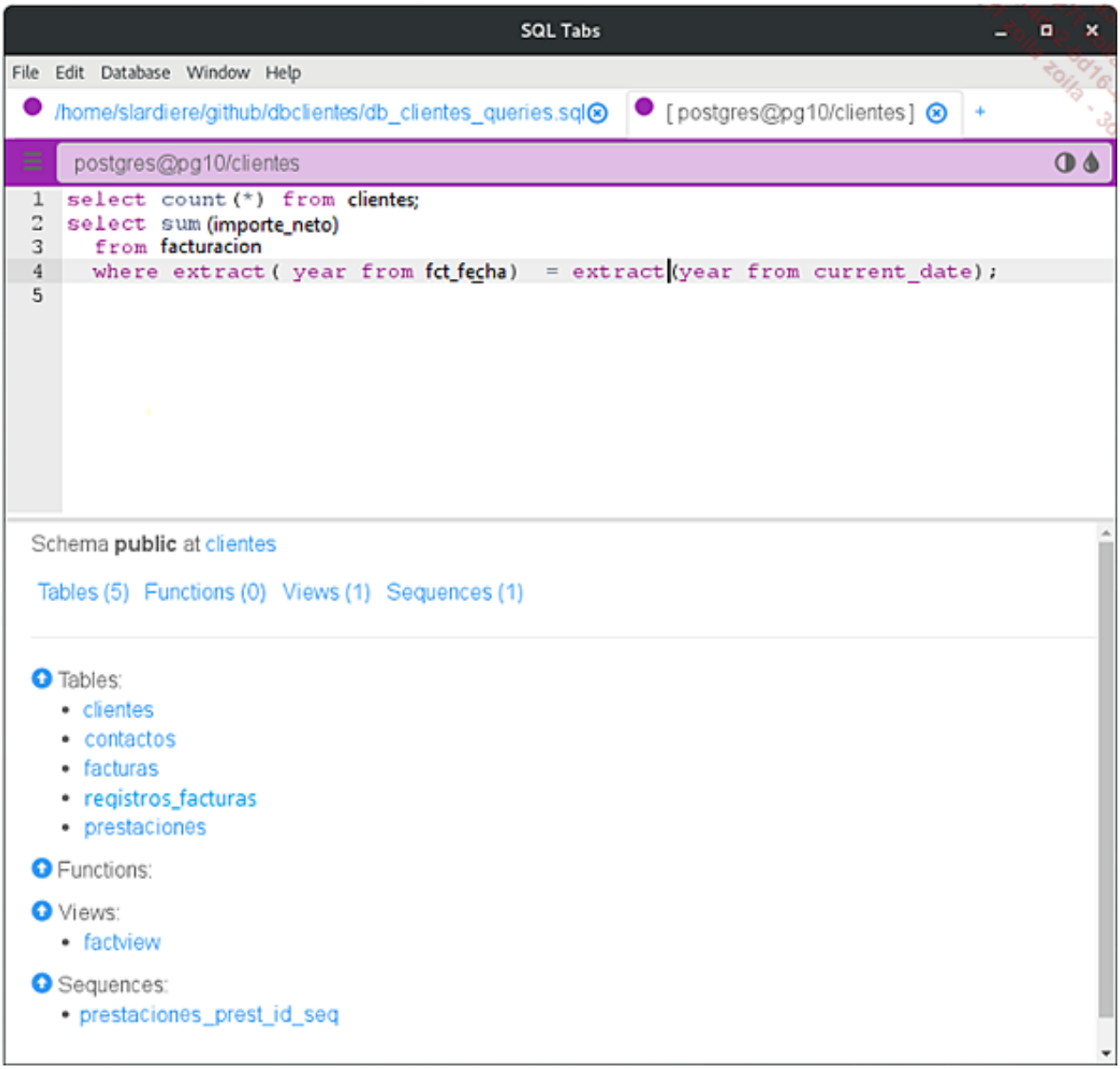


La herramienta SQLTabs

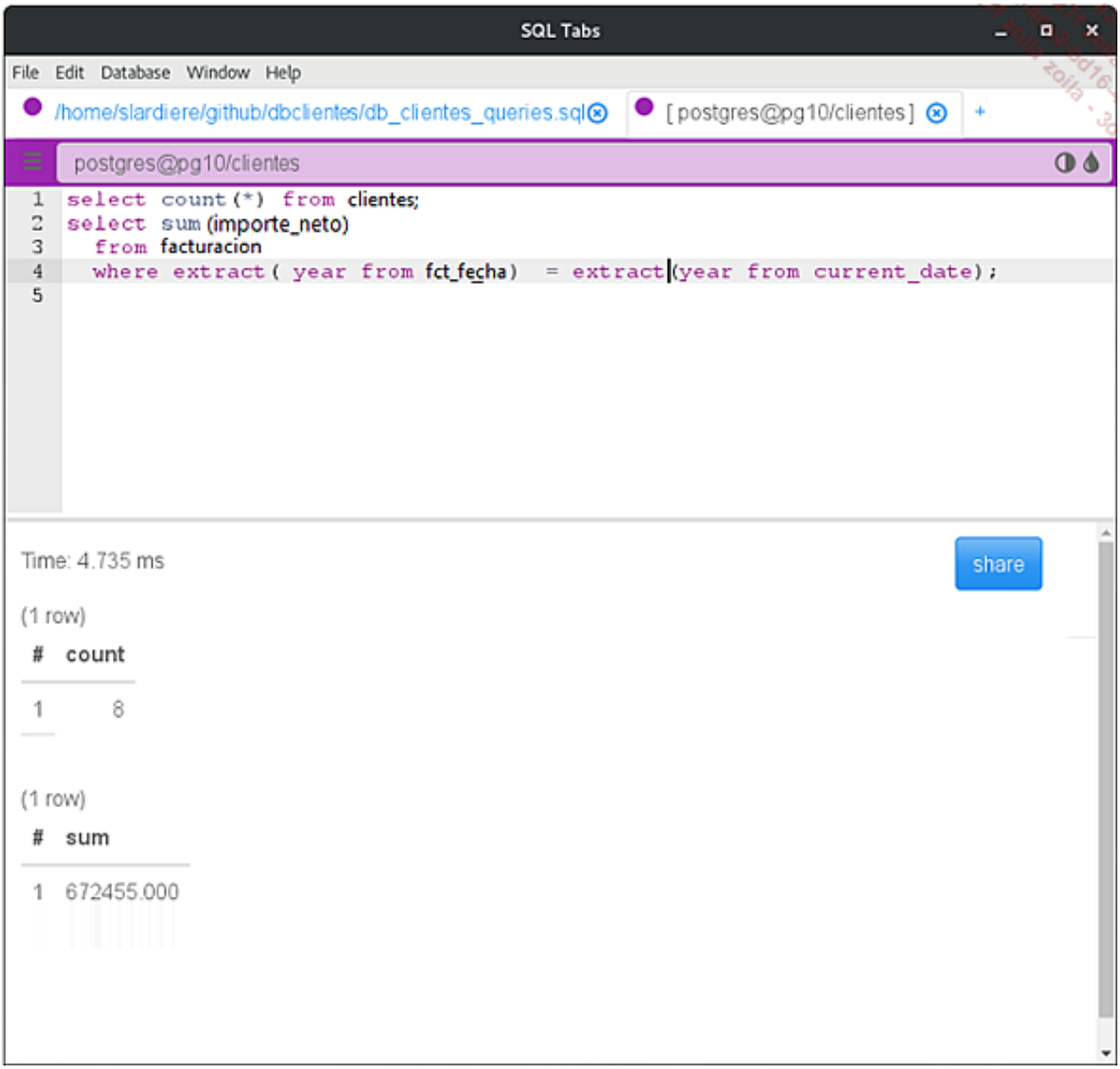
SQLTabs es un cliente SQL que dispone de numerosas funcionalidades interesantes en términos de ergonomía, tanto sobre el plan de la cobertura del modelo de datos como del autocompletado de consultas SQL, hasta la presentación de los resultados y los planes de ejecución.

La herramienta se descarga en la siguiente dirección: http://www.sqltabs.com/

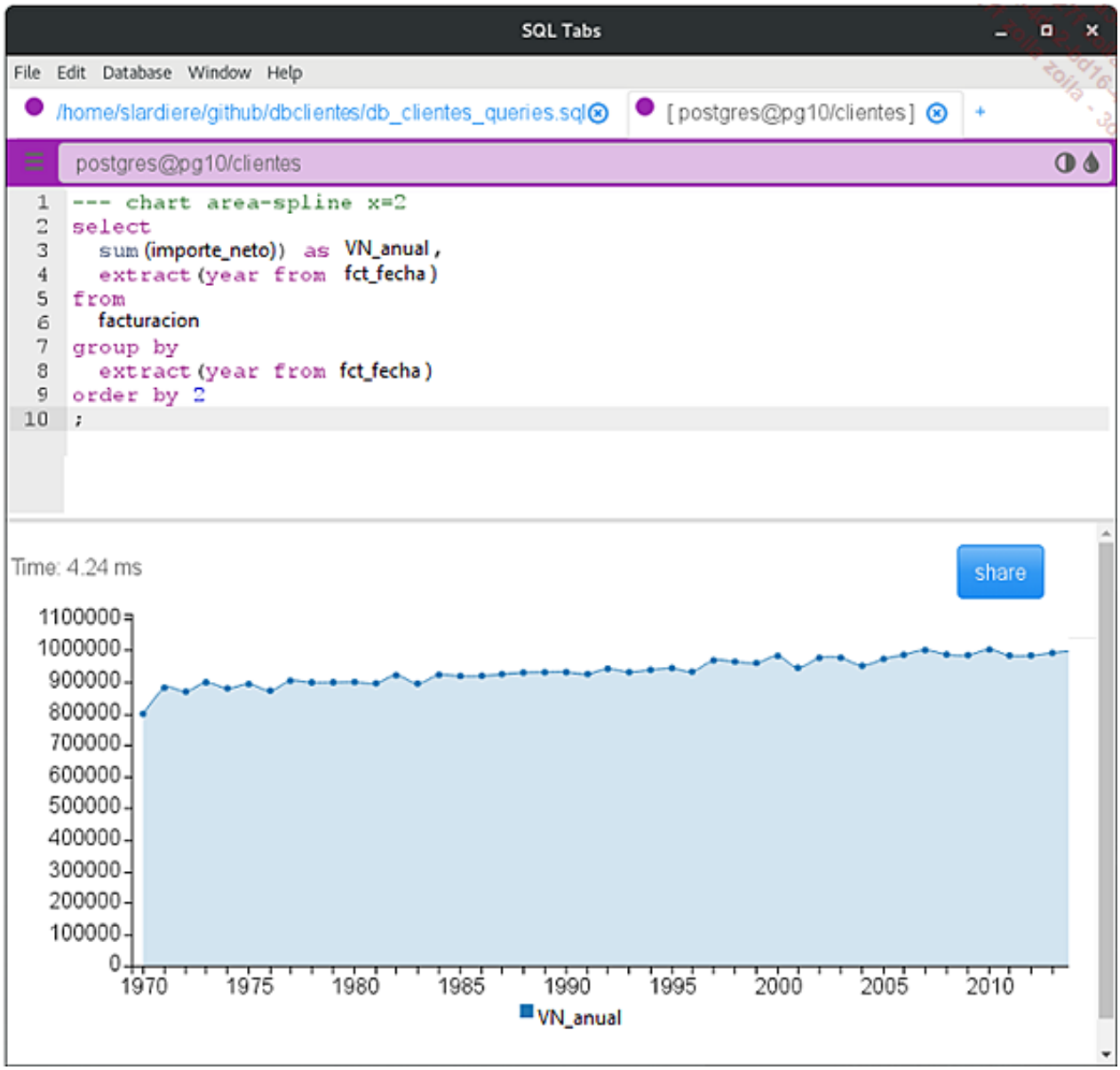
La siguiente pantalla muestra la herramienta, conectada gracias a la URI de la barra de direcciones desde una consulta SQL que se está editando, y los objetos de la base de datos en la zona de presentación:



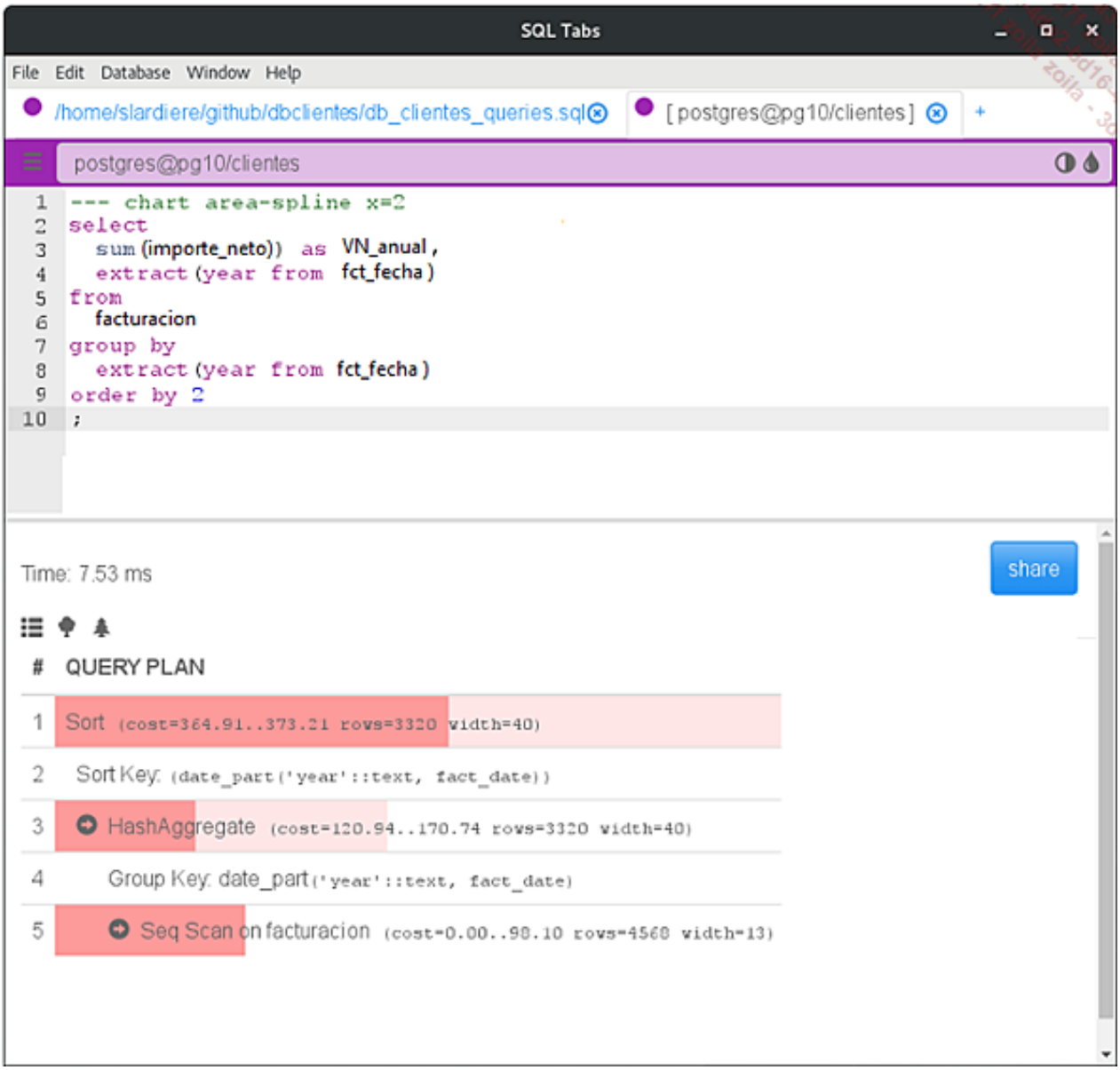
La siguiente pantalla muestra dos consultas cuyos resultados aparecen sucesivamente en la zona de presentación:



Además, la herramienta sabe interpretar los resultados para producir gráficos, como en el siguiente ejemplo:



Para terminar, es posible obtener una representación visual de los planes de ejecución:



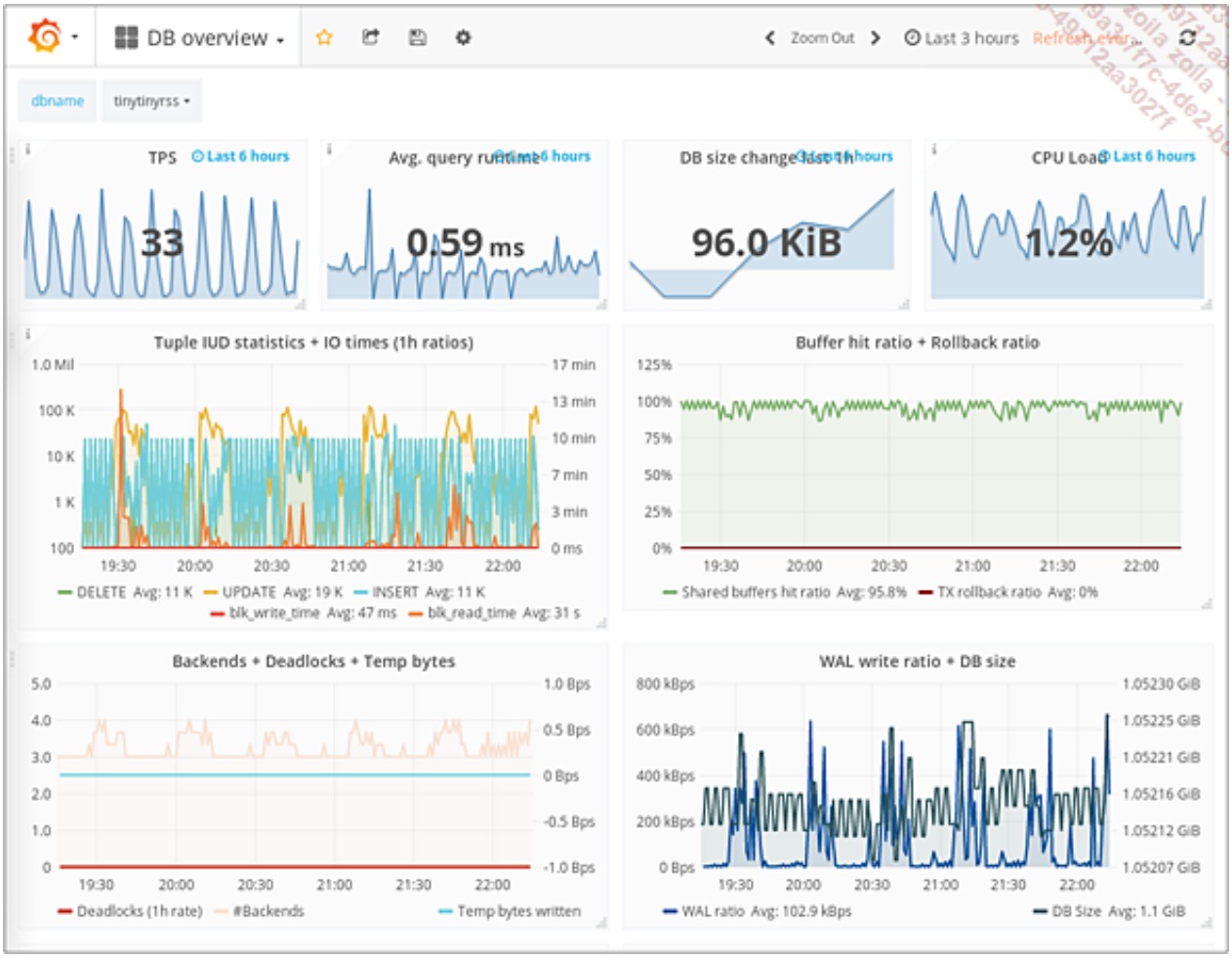
En la documentación de la herramienta se detallan numerosas opciones de visualización gráfica y de personalización de la zona de presentación.

La herramienta PGWatch2

PGWatch2 es una solución de supervisión que permite recoger métricas desde el catálogo de PostgreSQL e insertarlas en una base de datos de tipo «timeseries», como por ejemplo InfluxDB, para finalmente visualizarlas en Grafana, desde los cuadros de mando proporcionados por la herramienta. Aunque es posible instalar los diferentes componentes individualmente, como en el caso de los sistemas que ya utilizan InfluxDB y Grafana, la herramienta ofrece una imagen Docker que puede ser suficiente para evaluarla.

PGWatch2 se descarga en la siguiente dirección: https://github.com/cybertec-postgresql/pgwatch2

La siguiente copia de pantalla muestra el resultado de un cuadro de mandos de Grafana con los datos recogidos:



El administrador de base de datos dispone de toda la riqueza de Grafana para explorar los datos recogidos y, de esta manera, entender el comportamiento de PostgreSQL.

Replicación

Replicación en flujo

La replicación integrada en PostgreSQL, llamada «Streaming Replication», se basa en los mecanismos de copia de seguridad y restauración para poner en marcha una o varias instancias llamadas «standby», de una instancia principal de PostgreSQL.

Los procedimientos de copia de seguridad físicas integradas se basan en los archivos de traza de transacciones y, lógicamente, la replicación de los datos consiste en recrear de manera continua en las instancias «standby» el contenido de estos archivos de traza de transacciones.

Las primeras técnicas de replicación consistían en volver a leer los archivos de traza de transacciones archivados como resultado del argumento `archive_command`. Esta técnica siempre ha existido y se puede utilizar para permitir una mayor flexibilidad de la topología de los nodos PostgreSQL.

Con las versiones 9 de PostgreSQL, apareció el protocolo de replicación integrado, que permite a una instancia conectarse a otra instancia para leer de manera continuada el contenido de los archivos de traza de transacciones.

Esta técnica de replicación se puede utilizar en cascada, es decir, que cada uno de los servidores «standby» puede servir de fuente de replicación para otro servidor «standby», limitando de esta manera el peso de los servidores «standby» en el servidor principal.

1. La inicialización

La inicialización de una instancia «standby» consiste en hacer una copia de seguridad física con una herramienta como `pg_basebackup`, asegurándose de poder tener acceso siempre a los archivos de traza de transacciones intermedios entre el inicio de la copia de seguridad y el momento del arranque de la instancia «standby».

Este último punto es importante porque, como en el proceso de restauración PITR, la continuidad de los archivos de traza de transacciones es el medio utilizado para asegurar la continuidad de la replicación.

El comando `pg_basebackup` permite hacer esta copia de seguridad física inicial, simplemente a partir de la instancia principal.

Las herramientas de copia de seguridad como `pgbarman`, `pgbackrest` o `wal-e` se pueden utilizar como punto de partida porque las copias de seguridad realizadas permiten inicializar una instancia «standby».

La siguiente línea de comandos permite realizar una copia de seguridad física con un archivo `recovery.conf` mínimo:

```
pg_basebackup -R -h serverpg10 -U postgres -X stream -D 10/standby
```

La instancia principal debe haber sido preparada para autorizar las conexiones sobre el protocolo de replicación con los argumentos ajustados, como en el extracto de `postgresql.conf` siguiente:

```
wal_level = replica
max_wal_senders = 10
max_replication_slots = 10
```

Es el caso desde la versión 10, lo que hace que cualquier instancia de esta versión esté lista por defecto para aceptar los flujos de replicación. Es posible conectar varias instancias «standby» a un servidor «primary» en el límite del valor de la directiva `max_wal_senders`.

También se debe modificar el archivo `pg_hba.conf` para aceptar las conexiones de tipo REPLICATION, que es el caso por defecto desde la versión 10, incluso si falta por añadir las particularidades de la red o de los nombres de host realmente utilizados.

Estos cambios hacen necesario un rearranque de la instancia principal. Por este motivo es mejor anticiparlos para no tener que parar esta instancia. Los valores utilizados se deben adaptar en función del uso; por ejemplo, el número de servidores «standby».

2. Configuración

El comando `pg_basebackup` ofrece un archivo `recovery.conf` mínimo, que se puede completar. Se pueden utilizar las siguientes directivas:

- `standby_mode`: desactivado por defecto, este argumento permite poner la instancia en estado «standby». Cuando no es el caso, la replicación no funciona y la instancia se comporta como un nuevo servidor principal.
- `primary_conninfo`: cadena de conexión al servidor principal. `pg_basebackup` utiliza sus propios argumentos de conexión para guardar este argumento.
- `primary_slot_name`: slot de replicación creado por la función `pg_create_physical_replication_slot()` sobre el servidor principal. Permite conservar los archivos de traza de transacciones intermedios cuando la instancia «standby» se desconecta.
- `recovery_min_apply_delay`: permite añadir un plazo de aplicación de los datos replicados para conservar un margen de error en caso de incidente de datos de tipo `DROPTABLE`, `DELETE` o `TRUNCATE` sobre el servidor principal.
- `recovery_target_timeline = 'latest'`: permite seguir la última línea de tiempo, que evoluciona cuando promocionamos una instancia, por ejemplo en casos de basculación.
- `restore_command`: comando inverso de `archive_command`. Permite al servidor «standby» saber adónde ir a buscar los archivos de traza de transacciones si el servidor principal deja de estar disponible, lo que no debe suceder con la utilización de los slots de replicación.

a. Conexión al servidor «standby»

Por defecto, el servidor «standby» no permite conexiones de aplicaciones cliente. Es necesario activar en el archivo de configuración `postgresql.conf` los siguientes argumentos:

```
hot_standby = on
hot_standby_feedback = on
```

b. Slot de replicación

Los slots de replicación permiten bloquear los archivos de traza de transacciones intermedios sobre el servidor principal. Cuando un servidor «standby» se desconecta por este mecanismo de slot, el servidor principal conoce la última posición del «standby» y conserva todos los archivos de traza de transacciones útiles.

Las funciones útiles para la manipulación de estos slots son:

- `pg_create_physical_replication_slot(slot_name name)`: crea un slot de replicación. El nombre se debe indicar en el argumento `primary_slot_name`.

- `pg_drop_replication_slot(slot_name name)` : elimina el slot de replicación. Cuando se elimina un nodo de la replicación, es importante no conservar el slot porque los archivos de traza de transacciones también se conservan.

La vista `pg_replication_slots` permite al administrador conocer el estado de los slots.

c. Arranque

La instancia «standby» arranca como una instancia normal de PostgreSQL.

La sola presencia del archivo `recovery.conf` configurado correctamente es suficiente para hacer de la instancia PostgreSQL un servidor «standby».

Cuando la instancia «standby» se arranca correctamente, los procesos en segundo plano dedicados a la replicación están creados.

Un proceso «`wal sender`» aparece en el servidor principal, y un proceso «`wal receiver`», en el servidor «standby».

Debian

En los sistemas Debian, el comando `pg_basebackup` no copia los archivos de configuración `postgresql.conf`, `pg_hba.conf` y `pg_ident.conf`. Por lo tanto, hay que copiarlos independientemente para crear la instancia de forma correcta con `pg_createcluster`.

Atención: el archivo de configuración `postgresql.conf` contiene el argumento `data_directory`, que se puede modificar para que se corresponda con la ruta de los datos de la instancia «standby».

Red Hat

En los sistemas Red Hat, el script de arranque se debe adaptar para indicar la ruta del directorio de los datos.

3. Administración

Para monitorizar correctamente la replicación, existen dos vistas accesibles en el servidor principal:

- `pg_stat_replication`: muestra para cada uno de los servidores «standby», la información de la replicación, principalmente los movimientos de la aplicación de los datos de replicación. La siguiente consulta muestra este movimiento, expresado en bytes, y desde la versión 10, en tiempo:

```
select pid, backend_start, state, sync_state,
pg_size_pretty(pg_wal_lsn_diff(sent_lsn, write_lsn)) as write_size_lag,
pg_size_pretty(pg_wal_lsn_diff(sent_lsn, flush_lsn)) as flush_size_lag,
pg_size_pretty(pg_wal_lsn_diff(sent_lsn, replay_lsn)) as replay_size_lag,
write_lag, flush_lag, replay_lag  from pg_stat_replication;
-[ RECORD 1 ]---+------------------------------
pid             | 18196
backend_start   | 2017-08-30 21:05:23.425949+02
state           | streaming
sync_state      | async
write_size_lag  | 34 kB
flush_size_lag  | 34 kB
replay_size_lag | 146 kB
write_lag       | 00:00:00.001046
flush_lag       | 00:00:00.002642
replay_lag      | 00:00:00.016767
```

Estas medidas se deben recoger para monitorizar el correcto funcionamiento de la replicación. Los movimientos expresan tres estados de los datos en el servidor «standby»: `write` cuando se reciben los datos, `flush` cuando se escriben y se sincronizan en el almacenamiento y, para terminar, `replay` cuando son visibles en la instancia «standby».

- `pg_replication_slots`: muestra los slots de replicación, en particular el atributo `active`, que debe ser verdadero (`t`), indicando que el servidor «standby» se ha conectado correctamente.

Replicación en cascada

Es posible elaborar topologías de replicación complejas, principalmente poniendo las instancias «standby» unas junto a otras. Esta elección se hace en función de las posibilidades de la red, de las necesidades reales y de las evoluciones previstas en el marco del plan de recuperación o de continuidad de la actividad (PRA/PCA).

El número de instancias «standby» que es posible poner en cascada no tiene límite.

Cambio de topología

No existe un mecanismo de cambio automático integrado en PostgreSQL. Sí hay herramientas dedicadas, pero se incluyen en los planes de alta disponibilidad, que sobrepasan el marco de este libro.

Existe un comando que permite a una instancia «standby» convertirse en una instancia «primary»: se trata del comando `promote` de la herramienta `pg_ctl`, que simplemente va a detener la replicación y permitir las transacciones en modo lectura/escritura. En esta ocasión, la instancia va a cambiar de línea de tiempo (visible en los nombres de los archivos de traza de transacciones) y entonces ya no es posible conectarse a la instancia promovida a su antigua instancia «primary». El comando se recupera en el script de control `pg_ctlcluster`, disponible en los sistemas Debian.

Las otras instancias «standby» se pueden conectar a esta nueva instancia «primary» modificando el argumento `host` de la directiva `primary_conninfo` del archivo `recovery.conf` e indicando la nueva línea de tiempo que hay que seguir con la directiva `recovery_target_timeline` o, más simplemente, indicando `latest` como valor.

Replicación síncrona

Por defecto, la replicación integrada es asíncrona, es decir, que la transacción en el servidor «primary» se valida antes de ser replicada. Es posible configurar la replicación en modo síncrono para replicar los datos antes de validar la transacción. Por defecto, los datos se replican, pero no son visibles inmediatamente en la instancia «standby». Desde de la versión 9.6 de PostgreSQL, es posible configurar la replicación para que sea síncrona hasta la visibilidad de los datos en la instancia «standby».

Una instancia «standby» se debe declarar en la directiva `synchronous_standby_names` de la instancia «primary». Esta directiva puede contener una lista de nombres separados por comas o combinaciones con los operadores `FIRST` o `ANY`, como se ha explicado en el capítulo Explotación. Cada instancia «standby» se identifica utilizando el argumento `application_name` de la cadena de conexión `primary_conninfo` del archivo `recovery.conf`.

Una vez arrancada la instancia «standby» para tener en cuenta esta configuración, es necesario verificar el estado de la columna `sync_state` de la vista `pg_stat_replication` para la conexión deseada, que debe tener el valor `sync`, y no `async`.

Además de esta configuración inicial, es posible ajustar de manera más fina el sincronismo, modificando la directiva `synchronous_commit` en la configuración de la instancia «primary», como se ha explicado en el capítulo Explotación. Los valores `remote_write` y `remote_apply` permiten modificar el ámbito del sincronismo de la replicación; `remote_apply` permite validar la transacción solo cuando el dato es visible en la instancia «standby».

El valor `local` desactiva la replicación síncrona, manteniendo la sincronicidad de los datos locales. Entonces es posible modificar la directiva localmente de una transacción, que permite un ajuste fino de la granularidad de la replicación síncrona, controlada al cierre de la transacción. La siguiente secuencia muestra la utilización de la directiva en una transacción:

```
begin:
set local synchronous_commit = remote_apply;
...
commit;
```

Replicación lógica integrada

Desde la versión 10 de PostgreSQL, es posible implementar una replicación lógica sin herramienta adicional, es decir, sin utilizar Slony o Londiste. Esta funcionalidad se basa en los archivos de traza de transacciones enriquecidos y decodificados por las funciones de decodificación lógica de PostgreSQL. En los casos de la replicación lógica, independientemente de la técnica, se habla de «provider» para designar la instancia que ofrece el dato y de «subscriber» para la instancia que lo recibe.

Para permitir esta replicación, es necesario modificar el valor de la directiva de configuración `wal_level` a `logical`, en lugar de `replica` por defecto. Esta modificación hace necesario un rearranque de la instancia.

Una vez que se aplica esta modificación y el archivo `pg_hba.conf` se adapta para aceptar las conexiones, las siguientes sentencias `CREATE PUBLICATION` y `CREATE SUBSCRIPTION` permiten establecer un juego de tablas para replicar, después de abonarse a ellas desde otra instancia.

La siguiente sentencia permite crear la publicación en la instancia donde los datos de las tablas se modifican:

```
CREATE PUBLICATION clientes_repli FOR TABLE public.clientes,
public.facturas, public.prestaciones;
```

Luego, desde la instancia donde los datos se deben leer, indicando la conexión a la instancia «provider»:

```
CREATE SUBSCRIPTION sub_clientes_repli CONNECTION 'host=servidorpg10
port=5432 dbname=clientes' PUBLICATION clientes_repli;
```

Entonces, esta replicación se puede monitorizar desde la vista `pg_stat_replication`.

Replicación lógica con Slony

La herramienta Slony-I es un sistema de replicación lógica asíncrona maestro-esclavo. Permite replicar datos entre varios servidores PostgreSQL, sabiendo que solo uno de estos servidores será el servidor maestro, y el resto, los servidores esclavos. Estos servidores esclavo se pueden poner en cascada, pero la cantidad de servidores esclavos debe permanecer razonable para no tener impacto negativo en el rendimiento.

El servidor maestro es el que debe utilizar las aplicaciones para todas las modificaciones de datos. Los servidores esclavos reciben una copia de estas modificaciones de manera asíncrona. Esta replicación asíncrona implica que el servidor maestro no espera la validación de los servidores esclavos para validar su propia transacción a la aplicación cliente. La validación de una transacción en el servidor maestro solo tiene en cuenta el estado del servidor maestro, y no el de los servidores esclavos.

La replicación se puede realizar en una red local o en las redes extendidas (WAN). Esto se debe al hecho de que Slony utiliza conexiones TCP entre los servidores.

Las diferentes posibilidades de Slony son:

- La posibilidad de añadir servidores al grupo de replicación durante el funcionamiento de este grupo, así como la posibilidad de sustituir un servidor maestro por otro.
- La posibilidad de sustituir el servidor maestro durante un funcionamiento erróneo, por ejemplo no disponibilidad, temporal o no.
- La posibilidad de modificar la estructura de los datos durante el funcionamiento y propagar estas modificaciones a todos los servidores esclavos.

La unidad de replicación de Slony es la tabla. En un mismo grupo de servidores, Slony-I copia los datos de una tabla utilizando los triggers ubicados en estas tablas durante la inicialización de Slony. Es posible usar varias tablas, con la condición de que todas existan dentro de la misma base de datos.

Además, cada tabla debe tener una clave primaria; si el modelo de datos no define una clave primaria sobre una tabla, Slony-I puede añadir una columna para esto.

1. Instalación de Slony

Slony está disponible en forma de paquete en los sistemas Debian y Red Hat. El comando de instalación bajo Red Hat es el siguiente:

```
yum install slony1-10
```

En un sistema Debian, el comando es el siguiente:

```
apt-get install postgresql-10-slony1-2 slony1-2-bin
```

Los paquetes instalan el programa `slon`, que es el demonio utilizado para la replicación; el programa `slonik`, herramienta de línea de comandos y de administración, y las librerías de funciones necesarias para PostgreSQL.

El programa `slon` se arranca en cada sistema donde haya un nodo del grupo de servidores y se utiliza para la comunicación entre los servidores. El programa `slonik` es la herramienta de administración que permite controlar el comportamiento de la replicación.

2. Configuración del grupo de servidores

La primera etapa es copiar la estructura de la base de datos existente en todos los servidores utilizados en el grupo de servidores. El siguiente comando permite copiar la base de datos `clientes`, usando los comandos `pg_dump` y `psql`:

```
[postgres]$ createdb -h subscriberhost1 clientes
[postgres]$ pg_dump -h providerhost -s -p 5432 clientes | psql -h
subscriberhost -p 5432 clientes
```

Esta etapa se repite para cada servidor esclavo.

Es esencial permitir las conexiones a los diferentes servidores PostgreSQL desde todos los hosts, utilizando por ejemplo un rol dedicado para la replicación que tenga permisos suficientes para realizar las diferentes transacciones y después ajustando el archivo `pg_hba.conf` de manera adecuada.

3. Inicialización

El siguiente script utiliza el comando `slonik` para inicializar el grupo de servidores; aquí, con un maestro y un esclavo. Este script se debe registrar en un archivo, por ejemplo `slonik_clientes_init.sh`, y ejecutar para guardar en un esquema específico la siguiente información:

```
#!/bin/sh
#
# Ejecutar ./slonik_clientes_init.sh
#
/usr/local/slony1/bin/slonik < _END_

# Definición del nombre del grupo de servidores

cluster name = clientes;

# Definición de los nodos

node 1 admin conninfo = 'dbname=clientes
host=providerhost port=5432
user=slonyuser';
node 2 admin conninfo = 'dbname=clientes host=subscriberhost1
port=5432 user=slonyuser';

# Inicialización del grupo de servidores

init cluster (id=1, comment='Node 1');

# Registro del segundo nodo

store node (id=2, comment='Node 2');

# Creación de las rutas. Cada servidor debe conocer
el resto de los servidores.

store path (server=1, cliente=2, conninfo='dbname=clientes
host=providerhost port=5432 user=slonyuser');
store path (server=2, cliente=1, conninfo='dbname=clientes
host=subscriberhost1 port=5432 user=slonyuser');

# Activación de la escucha de los eventos del resto de los servidores.

store listen (origin=1, receiver=2, provider=1);
store listen (origin=2, receiver=1, provider=2);
_END_
```

Las etapas `store path` y `listen` deben permitir a cada servidor escuchar cualquier otro servidor. Esta configuración permite cambiar de maestro en el grupo de servidores sin tener que añadir las rutas.

4. Arranque del programa slon

El demonio `slon` permite la replicación entre los nodos. Para cada servidor del grupo, se debe arrancar un demonio. El archivo de configuración del demonio `slon` contiene la cadena de conexión a la instancia local, por ejemplo:

```
conn_info='port=5432 user=postgres'
```

5. Creación del juego de tablas

Una vez que los demonios `slon` se arrancan, el siguiente script crea un juego de tablas existentes que se deben replicar:

```
#!/bin/sh
# ./clientes_tables.sh
/usr/local/slony1/bin/slonik < _END_

cluster name = clientes;
node 1 admin conninfo = 'dbname=clientes host=providerhost
port=5432 user=slonyuser';
node 2 admin conninfo = 'dbname=clientes host=subscriberhost1
port=5432 user=slonyuser';

create set (id=1, origin=1, comment='tablas Clientes');

# adición de las tablas

set add table (set id=1, origin=1, id=1, full qualified
name = 'public.clientes', comment='tabla de Clientes');
set add table (set id=1, origin=1, id=3, full qualified
name = 'public.facturas', comment='tabla de Facturas');
set add table (set id=1, origin=1, id=5, full qualified
name = 'public.prestaciones',
comment='Tabla de Prestaciones');

# Creación de las claves primarias para las tablas que no la tienen
Tabla add key (node id = 1, fully qualified name =
'public.contactos');
table add key (node id = 1, fully qualified name =
'public.registros_facturas');

set add table (set id=1, origin=1, id=2, full qualified
name = 'public.contactos', comment='tabla de Contactos',
key = serial );
set add table (set id=1, origin=1, id=4, full qualified
name = 'public.registros_facturas',
comment='Registros de Facturas', key = serial );

# Suscripción del esclavo ( 2 ) al juego de tablas en el maestro ( 1 )
subscribe set (id=1, provider=1, receiver=2, forward=yes);
_END_
```

Como para la inicialización, estos comandos se deben guardar en un script después de ser ejecutados.

Una vez que termina esta etapa, la replicación debe funcionar: las modificaciones de los datos en las tablas del servidor maestro se deben replicar en el servidor esclavo.

Este servidor «subscriber» es una copia del servidor «provider», por lo que las copias de seguridad se pueden realizar sobre este. Además, Pgpool sabe utilizar los dos servidores para repartir las consultas de los clientes, usando el servidor esclavo para las consultas en modo lectura.

6. Modificación del esquema

Puede suceder que aparezcan modificaciones durante el ciclo de vida de una base de datos, por ejemplo la adición de una columna a una tabla. Estas modificaciones no se propagan automáticamente a los diferentes nodos del grupo de servidores. Si solo se realizan en el servidor maestro, la replicación fallará y las tablas dejarán de ser las mismas.

Slony permite propagar estas modificaciones ejecutando un script que contiene, por ejemplo, un comando `ALTER TABLE`.

Por ejemplo, cuando las aplicaciones necesitan saber si un cliente está activo o no, se puede añadir un campo `activo` de tipo booleano y se debe ejecutar en todos los servidores, por ejemplo:

```
ALTER TABLE clientes ADD COLUMN active bool default false;
```

Este script se puede ejecutar con el comando `execute script`:

```
#!/bin/sh
/usr/local/slony1/bin/slonik <_EOF_
cluster name = clientes;
node 1 admin conninfo = 'dbname=clientes host=providerhost
port=5432 user=slonyuser';
node 2 admin conninfo = 'dbname=clientes host=subscriberhost1
port=5432 user=slonyuser';

execute script ( set id = 1, filename = 'addcolumnactif.sql' , event
node = 1);
_EOF_
```

Se pueden lanzar muchos otros comandos, como la adición de nuevos servidores en el grupo, el cambio de servidores maestros (principalmente en casos de fallo grave) y la eliminación de un nodo. Estos comandos utilizan el comando `slonik` y se ejecutan de la misma manera que en los ejemplos anteriores.

Evolución de las soluciones de replicación

Las evoluciones técnicas que han traído las últimas versiones de PostgreSQL permiten considerar los desarrollos alrededor de la replicación lógica integrada en PostgreSQL sin sobrecargar ni hacer doble escritura y, por lo tanto, sin impacto sobre el rendimiento.

Hay otras soluciones que ven la luz y van a permitir completar las posibilidades actuales de PostgreSQL

BDR permite crear replicaciones de servidores principales a servidores principales, algunas veces llamadas replicación multimaestros. Esta solución ya es funcional y se debería integrar en una futura versión de PostgreSQL, posiblemente en la versión 11, que sale a finales de 2018.