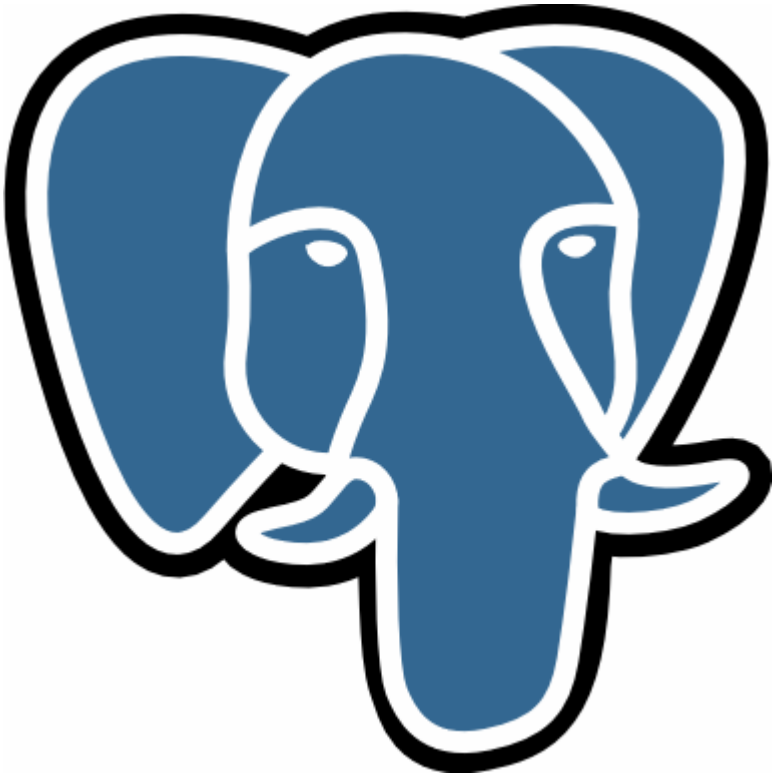


Manual PostgreSQL

Instalación, Creación, Mantención, PlpgSQL

Abril 2008
Versión 0.3

Jonathan Makuc
Cristian Molina
Armando Reyes

JM SOFTWARE

Índice

www.jm.cl // jm@jm.cl
Moneda 973, Of 536 – Fono-fax: 638 6201

1. Introducción

1.1. Presentación

El presente manual tiene por finalidad servir como una guía para iniciados y usuarios nivel medio en PostgreSQL. Sus contenidos desde la instalación del software hasta procedimientos almacenados en PlpgSQL, pretenden ser una base al lector para interiorizarse en el uso completo de uno de los motores de base de datos más completos del mercado.

El presente documento ha sido elaborado en base a PostgreSQL versión 8.1, en sistema operativo Linux Debian.

1.2. Glosario

DBMS Database Management System, Sistema de administración de base de datos. Formalmente la *base de datos* es la forma en que se almacenan estos en un medio (ej: disco duro), y el software que administra el acceso a tal repositorio de información es el DBMS, en este caso, PostgreSQL.

Encoding Palabra para *codificación* en inglés, hace referencia a la forma en que se representan en un computador los caracteres. Tiene una relevancia no menor en sistemas que trabajan con caracteres no tradicionales como acentos, ñ, etc; dadas las distintas formas de poder codificar un mismo elemento.

DDL Data Definition Language. Porción del lenguaje SQL que permite definir y modificar la estructura de una base de datos.

DML Data Manipulation Language. Porción del lenguaje SQL que permite manipular los datos de una base de datos.

2. Instalación, Configuración y Mantención

2.1. Instalación

2.1.1. Instalación APT-GET

En Linux Debian, y Ubuntu, es posible la instalación de PostgreSQL a través de la *automágica* aplicación APT. Esta permite la descarga, instalación, inicialización y configuración básica del DBMS con solo una línea de comando.

```
debian:~# apt-get install postgresql-8.1
Reading package lists... Done
Building dependency tree... Done
The following extra packages will be installed:
  libpq4 openssl postgresql-client-8.1 postgresql-client-common postgresql-common ssl-cert
Suggested packages:
  ca-certificates postgresql-doc-8.1
The following NEW packages will be installed:
  libpq4 openssl postgresql-8.1 postgresql-client-8.1 postgresql-client-common postgresql-common ssl-
cert
0 upgraded, 7 newly installed, 0 to remove and 0 not upgraded.
Need to get 6865kB/7143kB of archives.
After unpacking 18.4MB of additional disk space will be used.
Do you want to continue [Y/n]?
```

Al ingresar “Y” y presionar ENTER, APT realizará toda la instalación automáticamente.

```
Get:1 http://debian.ubiobio.cl etch/main openssl 0.9.8c-4etch1 [1001kB]
Get:2 http://debian.ubiobio.cl etch/main postgresql-client-common 71 [40.2kB]
Get:3 http://debian.ubiobio.cl etch/main postgresql-client-8.1 8.1.11-0etch1 [1422kB]
Get:4 http://debian.ubiobio.cl etch/main ssl-cert 1.0.14 [11.1kB]
Get:5 http://debian.ubiobio.cl etch/main postgresql-common 71 [102kB]
Get:6 http://debian.ubiobio.cl etch/main postgresql-8.1 8.1.11-0etch1 [4289kB]
Fetched 6865kB in 14s (461kB/s)
Preconfiguring packages ...
Selecting previously deselected package libpq4.
(Reading database ... 19720 files and directories currently installed.)
Unpacking libpq4 (from .../libpq4_8.1.11-0etch1_i386.deb) ...
Selecting previously deselected package openssl.
Unpacking openssl (from .../openssl_0.9.8c-4etch1_i386.deb) ...
Creating directory /etc/ssl
Selecting previously deselected package postgresql-client-common.
Unpacking postgresql-client-common (from .../postgresql-client-common_71_all.deb) ...
Selecting previously deselected package postgresql-client-8.1.
Unpacking postgresql-client-8.1 (from .../postgresql-client-8.1_8.1.11-0etch1_i386.deb) ...
Selecting previously deselected package ssl-cert.
Unpacking ssl-cert (from .../ssl-cert_1.0.14_all.deb) ...
Selecting previously deselected package postgresql-common.
Unpacking postgresql-common (from .../postgresql-common_71_all.deb) ...
Selecting previously deselected package postgresql-8.1.
Unpacking postgresql-8.1 (from .../postgresql-8.1_8.1.11-0etch1_i386.deb) ...
Setting up libpq4 (8.1.11-0etch1) ...

Setting up openssl (0.9.8c-4etch1) ...

Setting up postgresql-client-common (71) ...
Setting up postgresql-client-8.1 (8.1.11-0etch1) ...

Setting up ssl-cert (1.0.14) ...
```

```
Setting up postgresql-common (71) ...
adduser: Warning: that home directory does not belong to the user you are currently creating.

Setting up postgresql-8.1 (8.1.11-0etch1) ...
Creating new cluster (configuration: /etc/postgresql/8.1/main, data: /var/lib/postgresql/8.1/main)...
Moving configuration file /var/lib/postgresql/8.1/main/postgresql.conf to /etc/postgresql/8.1/main...
Moving configuration file /var/lib/postgresql/8.1/main/pg_hba.conf to /etc/postgresql/8.1/main...
Moving configuration file /var/lib/postgresql/8.1/main/pg_ident.conf to /etc/postgresql/8.1/main...
Configuring postgresql.conf to use port 5432...
Starting PostgreSQL 8.1 database server: main.

debian:~#
```

Al concluir la operación el instalador a creado varios directorios y archivos. Destacamos:

/etc/postgres/8.1/main/	Archivos de configuración
/etc/init.d/postgres-8.1	Ejecutable único
/var/lib/postgresql/8.1/main	Directorio de almacenamiento de la Base de datos inicial
/var/log/postgresql	Directorio de log

De la misma se ha creado un usuario de sistema **postgres**, el cual es el *superusuario* del DBMS con permisos completos sobre todas las bases de datos controladas. Este perfil es utilizado para la creación de Usuarios, ejecución de labores automáticas de mantención, etc.

2.1.2. Ingreso al sistema por primera vez

Al momento de inicialización de una ubicación de base de datos, se crean bases de ejemplo con el usuario *postgres* como dueño.

Para listar las bases de datos controladas por una instancia de Postgres se utiliza el siguiente comando:

```
postgres@debian:~$ psql -l
        List of databases
   Name    |  Owner   | Encoding
-----------+----------+----------
 postgres  | postgres | UTF8
 template0 | postgres | UTF8
 template1 | postgres | UTF8
(3 rows)
```

Para ingresar a una base de datos, se utiliza el siguiente comando. Detalles de cómo crear bases y las opciones asociadas, se pueden encontrar en la sección 3.1 del presente manual.

```
postgres@debian:~$ psql
Welcome to psql 8.1.11, the PostgreSQL interactive terminal.

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

postgres=#
```

Al no proporcionar parámetros, se asume que se esta intentando de acceder a la base de datos cuyo nombre coincide con el del usuario actual, en este caso *postgres*. Como postgres es superusuario, no solicita clave aunque la base estuviese configurada con esta protección.

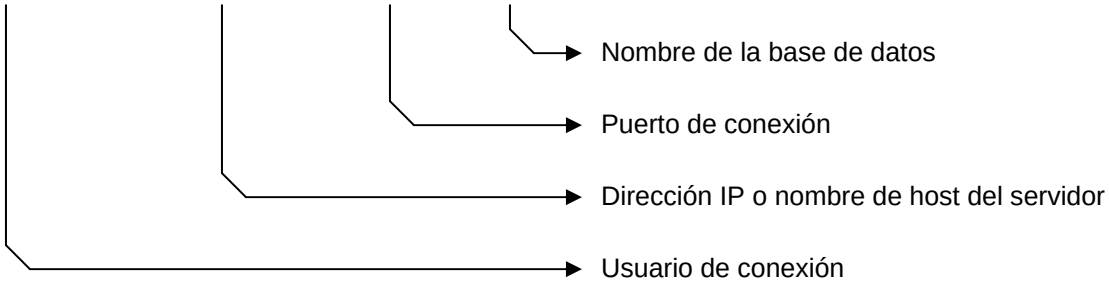
Notar que cuando se utiliza un superusuario en la conexión de consola al DBMS, también aparece el símbolo #, como sucede en un shell unix-like.

2.1.3. Ingreso vía línea de comando

Para ingresar a un motor de base de datos PostgreSQL local o remoto a través de línea de comando, se utiliza el comando **psql**. Este está disponible al utilizar la instalación del meta-paquete "postgres-8.1" como se describe anteriormente, pero también puede ser descargado de forma independiente vía el paquete "postgresql-client-8.1".

A continuación se presenta un comando psql con las opciones más comunes. Consulte psql –help para la lista completa de opciones.

```
Psql –U usuario –h 192.168.0.100 –p5432 nombre_base
```



Si se omite el parámetro de *host*, entonces la conexión se intenta vía socket Unix al valor establecido en postgresql.conf. También el parámetro host puede contener la ruta del socket Unix al cual se desea conectar.

De requerirse clave para ingresar a la base de datos indicada, el sistema la solicitará automáticamente.

Comandos interesantes

\i archivo.sql	Ejecuta los comandos almacenados en archivo.sql
\o output.txt	Copia la salida de la pantalla a output.txt
\d	Lista las tablas existentes
\d tabla	Describe la estructura de una tabla
\df funcion	Describe la función indicada
\df+ funcion	Muestra el código de la función indicada
\dn	Lista los esquema existentes
\z	Lista los permisos de una relación.
\q	Salir. También funciona CTRL + D.
\?	Despliegue de todos los comandos de consola (ayuda)

2.1.4. Inicio, detención y recarga del DBMS

Inicio de PostgreSQL

Inicia el motor de base de datos Postgres.

```
root@debian:/# /etc/init.d/postgresql-8.1 start
 * Starting PostgreSQL 8.1 database server                                    [ ok ]
```

A continuación la línea de comando completa para sistemas no-Debian.

```
root@debian:/# /usr/lib/postgresql/8.1/bin/postmaster -D /var/lib/postgresql/8.1/main -c
unix_socket_directory=/var/run/postgresql -c config_file=/etc/postgresql/8.1/main/postgresql.conf -c
hba_file=/etc/postgresql/8.1/main/pg_hba.conf -c ident_file=/etc/postgresql/8.1/main/pg_ident.conf
* Restarting PostgreSQL 8.1 database server                                   [ ok ]
```

Detención de PostgreSQL

Detiene el funcionamiento del motor de base de datos Postgres.

```
root@debian:/# /etc/init.d/postgresql-8.1 stop
 * Stopping PostgreSQL 8.1 database server                                    [ ok ]
```

Reinicio de PostgreSQL

De estar el servidor abajo, lo inicio. De estar corriendo, lo baja – matando todas las sesiones activas – para volverlo a subir.

```
root@debian:/# /etc/init.d/postgresql-8.1 restart
 * Restarting PostgreSQL 8.1 database server                                  [ ok ]
```

Recarga de PostgreSQL

Actualización de los valores en memoria de los parámetros definidos en los archivos de configuración. Este método es el recomendado para hacer efectivos los cambios en los archivos de configuración.

```
root@debian:/# /etc/init.d/postgresql-8.1 reload
 * Restarting PostgreSQL 8.1 database server                                  [ ok ]
```

Los dos últimos métodos son efectivos para que Postgres pueda recargar los parámetros de los archivos de configuración.

2.2. Archivo de configuración postgres.conf

2.2.1. Generalidades

El archivo de configuración *postgres.conf*, mantiene los parámetros de operación por defecto del motor PostgreSQL. Este permite indicar preferencias de conexión (no permisos), ajustes (*tweekings*) de optimización, etc; los cuales pueden ser modificados solo por el superusuario, a menos que se cambien los permisos de acceso al archivo.

Sintaxis:

- Primera palabra en la línea esa la variable
- Uso de signo igual (=) es optativo, se asume el primer espacio en blanco como delimitador.
- Comentarios con #. Están permitidos desde cualquier punto de la fila.

El archivo *postgres.conf* es leído al momento de partir el motor y los cambios realizados sobre él mientras este se ejecuta no serán reflejados hasta que este se reinicie. A continuación se recorrerá el archivo de configuración sección a sección comentando los puntos más interesantes.

2.2.2. Ubicación archivos de sistema

Al momento de inicializar el motor de base de datos, se crean archivos de configuración y los archivos donde se almacenarán la información propiamente tal. Cuando se levanta el DBMS, se puede especificar la ubicación de estos archivos o de lo contrario se tomarán los especificados en esta sección o aquellos presentes en el directorio de operación.

```
data_directory = 'ConfigDir'
```

Permite indicar otro directorio donde se almacenan los archivos de la base de datos

```
hba_file = 'ConfigDir/pg_hba.conf'
```

Permite indicar otro archivo de configuración de permisos de acceso. De no partir con /, se asume path relativo al directorio de trabajo indicado al partir el DBMS.

```
ident_file = 'ConfigDir/pg_ident.conf'
```

Permite indicar otro archive de configuración de autenticación IDENT (Descrito en 2.3). De no partir con /, se asume path relativo al directorio de trabajo indicado al partir el DBMS.

```
external_pid_file = '(none)'
```

Permite solicitar la creación de un archive PID adicional en la ubicación indicada. De no partir con /, se asume path relativo al directorio de trabajo indicado al partir el DBMS.

2.2.3. Conexiones y autenticación

En esta sección es posible configurar las opciones de acceso desde un punto de vista de los servicios levantados por el motor. La sección de conexiones se encuentra subdividida en 4 partes: tipo de conexiones, parámetros de autenticación, kerberos y TCP.

En su configuración inicial, Postgres solo puede ser accedido desde socket unix, sin puertos TCP/IP abiertos.

Conexiones

```
listen_addresses = 'localhost'
```

Listado separado por comas de los host desde los cuales se aceptarán conexiones. De estar comentado o con valor '*', acepta desde cualquier host.

```
port = 5432
```

Puerto en el cual esta escuchando esta instancia de Postgres. Notar que con esta directiva se pueden ejecutar múltiples instancias del DBMS.

```
max_connections = 100
```

Cantidad máxima de conexiones permitidas, sin considerar al superusuario. Aumentar este parámetro conlleva a mayor uso de memoria compartida.

```
superuser_reserved_connections = 2
```

Cantidad de slots de conexión reservados para el superusuario

```
unix_socket_directory = '/var/run/postgresql'
```

Directorio de los sockets unix para esta instancia de Postgres. Notar que con esta directiva se pueden ejecutar múltiples instancias del DBMS.

```
unix_socket_group = ''
```

Indica el grupo dueño de los sockets unix.

```
unix_socket_permissions = 0777
```

Indica los permisos en octal de acceso a los sockets unix

```
bonjour_name = ''
```

Permite establecer el nombre del DBMS. De estar comentado o en blanco, toma el nombre el equipo host.

Autenticación

```
#authentication_timeout = 60            # 1-600, in seconds
```

Tiempo en segundos de espera por la información de login. Desde 1 a 600.

```
#ssl = true
```

Booleano (true | false) que indica si el uso de SSL es forzado. De estar comentado es falso.

```
#password_encryption = on
```

Flag que indica si el uso de encriptación en el envío de la contraseña es obligatorio o no. De estar comentado es apagado.

```
#db_user_namespace = off
```

Habilita la funcionalidad donde los nombres de usuario pueden ser específicos de cada base de datos, es decir, usuario@baseEjemplo quiere decir que usuario solo se aplicaría a 'baseEjemplo' y no a todas como es por defecto.

2.2.4. Uso de Recursos

En esta sección se puede especificar la cantidad de recursos que esta instancia de PostgreSQL esta autorizada a consumir.

```
shared_buffers = 1000
```

El valor de shared buffers (buffers compartidos), esta en directa relación con la cantidad de conexiones permitidas en el sistema debiendo ser al menos el doble. Cada buffer es de 8kb, siendo este valor fijado al momento de "construir" (build) Postgres, no modificable vía parámetro de ejecución. Este valor corresponde a la cantidad de memoria compartida y semáforos solicitados al sistema operativo.

```
temp_buffers = 1000
```

Los buffers temporales hacen referencia a la cantidad de memoria utilizada por cada sesión de base de datos para acceder a tablas temporales.

```
work_mem = 1024
```

Este valor en kilobytes, indica la cantidad de memoria a utilizar por operaciones de ordenamiento y tablas de hash, antes de utilizar archivo de paginación. Cada operación de este tipo se le permitirá utilizar esta cantidad de memoria, aún dentro de una misma sesión.

```
maintenance_work_mem = 16384
```

Esta directiva en kilobytes, indica la cantidad de memoria a utilizar por las labores de mantención como VACUUM, CREATE INDEX, restauración de bases de datos, etc. Como no se suelen dar operaciones concurrentes de este tipo, conviene dejarlo alto.

```
max_fsm_pages = 20000
```

Cantidad máxima de páginas libres que son monitoreadas en el mapa de espacio libre[1]. Debe ser al menos la cantidad de relaciones multiplicado por 16, ocupando 6 bytes cada entrada.

```
max_fsm_relations = 1000
```

Cantidad de relaciones (tablas e índices) de las cuales se mantiene monitoreo del espacio libre en el mapa de espacio libre. Deben ser al menos 100.

[1] Una página es una unidad indivisible de espacio en disco, en la cual se realizan las operaciones de escritura y lectura, desde y hacia el disco duro. Por defecto en PostgreSQL este valor es 8Kb.

2.2.5. Ajuste de Consultas (Query Tunning)

El planificador de consultas es el módulo de un DBMS que decide como realizar físicamente la operación. Detalles sobre su operación se pueden encontrar en el punto 3.10 del presente manual.

Postgres permite la modificación del comportamiento por defecto del planificador, impidiendo que realice ciertas operaciones como búsqueda por índices, ordamiento, etc; de manera de poder realizar comparaciones de desempeño.

```
#enable_bitmapscan = on
#enable_hashagg = on
#enable_hashjoin = on
#enable_indexscan = on
#enable_mergejoin = on
#enable_nestloop = on
#enable_seqscan = on
#enable_sort = on
#enable_tidscan = on
```

Al cambiar el valor del parametro a “off”, se puede deshabilitar la característica.

Costos indicados al planificador

```
#effective_cache_size = 1000            # typically 8KB each
```

Valor que asume el planificador para tamaño efectivo del cache de disco utilizado por un escaneo de índice. Valores altos hacen más probable el uso de índices.

```
#random_page_cost = 4                   # units are one sequential page fetch
```

Costo estimado de recuperación de una página no-secuencial. Valores más altos hacen más probables el uso de escaneos secuenciales.

```
#cpu_tuple_cost = 0.01                  # (same)
```

Costo de procesar una fila en una consulta. El valor esta indicado en una fracción del costo de recuperación de una página secuencial.

```
#cpu_index_tuple_cost = 0.001           # (same)
```

Costo de proceso de una fila durante un escaneo vía índice. El valor esta indicado en una fracción del costo de recuperación de una página secuencial.

```
#cpu_operator_cost = 0.0025             # (same)
```

Costo estimado de procesar cada operador en una cláusula WHERE. El valor está indicado en una fracción del costo de recuperación de una página secuencial.

Optimizador Genético de Consulta (GEQO – Genetic Query Optimizar)

PostgreSQL incorpora dentro de su analizador de consultas, un algoritmo genético para optimización de consultas que incorporan JOIN. Las siguientes directrices permiten ajustar el comportamiento de este algoritmo.

Los algoritmos genéticos son una optimización heurística que opera de forma no-determinística. El set de soluciones posibles para el problema se denomina *población*, mientras que el grado de adaptación de un *individuo* se llama *ajuste*. Las coordenadas del *individuo* dentro del espacio de soluciones se representa por *cromosomas*. Un *gene* es una subsección de un *cromosoma* el cual codifica el valor de un único parámetro dentro de la optimización.

Para obtener la solución se simula la recombinación y mutación de los individuos, realizándose la *selección natural* en base a aquel individuo que presenta en promedio un mayor grado de adaptación que sus ancestros.

```
#geqo = on
```

Habilita o deshabilita el uso del algoritmo genético.

```
#geqo_threshold = 12
```

Cantidad de elementos en la cláusula FROM desde los cuales el algoritmo genético es utilizado.

```
#geqo_effort = 5                        # range 1-10
```

Controla en enteros de 1 a 10, la proporción de tiempo que el planificador gasta en planificar y la efectividad del mismo. Valores más altos resultarán en un tiempo mayor de planificación, pero también aumenta la probabilidad de encontrar un plan más eficiente.

```
#geqo_pool_size = 0                     # selects default based on effort
```

Cantidad de individuos en la población, debiendo haber al menos 2. De setearse a cero, entonces se elige un tamaño óptimo en base al esfuerzo indicado en *gego_effort*.

```
#geqo_generations = 0                   # selects default based on effort
```

Controla la cantidad de generaciones (iteraciones) usadas por GEQO. Debe ser al menos 1. Seteado en cero, hace que el motor escoja un valor adecuado en base al esfuerzo indicado en *gego_effort*.

```
#geqo_selection_bias = 2.0              # range 1.5-2.0
```

Ajusta la presión de selección dentro de la población, valor de 1.5 a 2.0.

```
#default_statistics_target = 10         # range 1-1000
```

Indica la cantidad de columnas a usar en el análisis, en consultas que no se especifica o no se ha indicado vía ALTER TABLE SER STATISTICS.

```
#constraint_exclusion = off
```

Al estar habilitado compara las condiciones de la consulta con las restricciones de la tabla impuestas en CHECK.

```
#from_collapse_limit = 8
```

El planificador juntará subconsultas hacia niveles superiores si la lista resultante en la cláusula FROM no tiene más elementos que este valor. Valores pequeños resultan en planificaciones más cortas, pero menos efectivas. Es conveniente dejar este valor bajo *gego_threshold*.

```
#join_collapse_limit = 8                # 1 disables collapsing of explicit
```

El planificador rescribirá los INNER JOINs de la consulta a items FROM, siempre y cuando el largo de la lista en esa cláusula no sobrepase este valor. No tiene efecto alguno en OUTTER JOINs. Setear su valor a 1, inhibe la reescritura haciendo el planificador se comporte como la versión 7.3.

2.2.6. Reportes de errores y log

PostgreSQL permite configurar de forma exhaustiva el como, cuando y donde *logear* lo que sucede en el motor. A continuación se revisan cada una de las opciones de cómo postgres notifica de los eventos y problemas que suceden.

Donde logear

```
#log_destination = 'stderr'
```

Por defecto toda la salida de log de Postgres es llevada a *stderr*. Desde aquí es llevada hacia los archivos de log. En este parámetro puede ser cambiarse a syslog en linux y eventlog en windows, entre otros.

```
#redirect_stderr = off
```

Permite la captura de stderr y su dirección a los archivos de log. Esto es útil para poder capturar los errores de extensiones de terceros a postgres integrados con el DBMS como bibliotecas enlazadas dinámicamente.

```
#log_directory = 'pg_log'
```

Cuando esta habilitado *redirect_stderr*, esta opción indica el directorio que contendrá los archivos de log. Puede ser especificado absoluto o relativo al directorio de trabajo.

```
#log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
```

Cuando esta habilitado *redirect_stderr*, esta opción indica el formato de nombre de archive utilizado para logear.

```
#log_truncate_on_rotation = off
```

Indica si truncar el archivo donde se escribirá el log a cero, en el caso de que ya exista.

```
#log_rotation_age = 1440
```

Setea el tiempo máximo en minutos que puede durar un archivo de log. Transcurrido este periodo se fuerza la creación de un nuevo archivo de log. Estando en cero, se asume ilimitado

```
#log_rotation_size = 10240
```

Setea el tamaño máximo en kilobytes que puede tener un archive de log. Alcanzada esta cantidad, se fuerza la creación de un nuevo archivo de log. Estando en cero, se asume ilimitado. Es poco recomendable dejar el archivo sin límite de espacio puesto que en teoría podría generar un archivo más grande que lo que permite el sistema de archivo, haciendo colapsar el módulo de log.

```
#syslog_facility = 'LOCAL0'
```

De estar logeando a syslog en *redirect_stderr*, este parámetro indica que canal utilizar. Posibilidades: LOCAL0 … LOCAL7. Se recomienda leer el manual de syslog antes de cambiar este parámetro.

```
#syslog_ident = 'postgres'
```

De estar logeando a syslog en *redirect_stderr*, este parámetro indica el usuario con que se logea.

Cuando logear

Los mensajes generados por PostgreSQL están catalogados en grados de severidad. Ordenados del menos importante hasta el más grave. Cada nivel entrega los mensajes de su grado y los superiores.

DEBUG5 **DEBUG4** **DEBUG3** **DEBUG2** **DEBUG1**	Mensajes de debug utilizados por los desarrolladores de PostgreSQL. Habilitar esta opción no tiene mucho sentido para usuarios comunes y bajará desempeño del motor considerablemente dada la cantidad de información generada.
NOTICE	Aviso. Provocado por un componente de software de Postgres, un script o cualquier otro componente. Se asume como un aviso, sin carácter nocivo.
WARNING	Advertencia. Algún evento ocurrido ha realizado una operación que no esta del todo bien. Suele darse en sentencias SQL cuya sintaxis no es perfecta, pero puede ser corregida.
ERROR	Error. Evento irreparable que provoca que el comando ejecutado y la actividad que se este realizando se lleve a cabo. Dentro de los scripts se conoce como “Exception”.
FATAL	Error en una sesión del DBMS que provoca la terminación inmediata de esta.
PANIC	Error de la mayor gravedad que provoca la terminación inmediata de todo el DBMS, matando todas las sesiones en todas las BdD

Existe un 11° estado **LOG** el cual varía su posición dependiendo de la directiva, la cual reporta información interesante a administradores del sistema como actividades de checkpoint.

Si bien es interesante poder tener información sobre que hace el DBMS y como, es importante saber sopesar la cantidad de información versus el desempeño del motor. Setear una gran cantidad de log puede entregar valiosa información al momento de desarrollar una aplicación, ayudando así a depurar más rápidamente. Por otra parte, una gran cantidad de log en un ambiente de producción no es muy útil, asumiendo que se tienen aplicaciones depuradas; en este caso solo se obtiene una disminución en el desempeño del motor sin un beneficio adecuado.

```
#client_min_messages = notice
```

Controla los mensajes que son enviados al cliente. Aquí LOG se encuentra entre DEBUG1 y NOTICE.

```
#log_min_messages = notice
```

Controla los mensajes que son logeados en el servidor. Aquí LOG se encuentra entre ERROR y FATAL. Esto es dado que en ambientes de producción solo se logean problemas, ante una caída es interesante saber en que puntos estaban ciertas operaciones administrativas como los checkpoints.

```
#log_error_verbosity = default
```

Controla el nivel de detalle que se maneja de cada mensaje. TERSE, DEFAULT y VERBOSE son valores válidos, cada uno agregando más campos a los mensajes entregados.

```
#log_min_error_statement = panic
```

Indica el nivel de error mínimo que debe tener una sentencia SQL para que esta sea logeada junto con el error que provocó. Para obtener mayor información sobre problemas con sentencias SQL como sintaxis, se puede bajar el nivel de este parámetro combinado con *log_error_verbosity.*

```
#log_min_duration_statement = -1
```

De estar seteado en un valor mayor que cero, indica la duración mínima que debe tener una consulta en milisegundos para que sea logeada. Este parámetro funciona en combinación con *log_min_error_statement.*

```
#silent_mode = off
```

Como su nombre lo indica, ejecuta el servidor en forma silenciosa redireccionando el log a /dev/null.

Que logear

```
#debug_print_parse = off
```

Opción de debug que permite habilitar la impresión del árbol de parseo de la consulta. El nivel de log debe estar al menos en DEBUG1 para que el parámetro tenga efecto.

```
#debug_print_rewritten = off
```

Opción de debug que permite habilitar la impresión de la consulta rescrita por el planificador. El nivel de log debe estar al menos en DEBUG1 para que el parámetro tenga efecto.

```
#debug_print_plan = off
```

Opción de debug que permite habilitar la impresión del plan de consulta. El nivel de log debe estar al menos en DEBUG1 para que el parámetro tenga efecto.

```
#debug_pretty_print = off
```

Opción de debug que habilita la identación de la salida de las opciones anteriores produciendo un formato más legible a personas, pero más extenso. El nivel de log debe estar al menos en DEBUG1 para que el parámetro tenga efecto.

```
#log_connections = off
```

De estar habilitado, logea cada conexión exitosa realizada al servidor.

```
#log_disconnections = off
```

De estar habilitado, logea cada desconexión del servidor.

```
#log_duration = off
```

De estar habilitado logea la duración de cada consulta SQL realizada.

```
#log_line_prefix = ''
```

String estilo *printf* que se coloca al inicio de cada línea de log. Detalles de los formatos permitido en la sección 17.7.3 del manual de usuario Postgres 8.1.

```
#log_statement = 'none'
```

Indica si logear las sentencias SQL ejecutadas, y cuales de ellas. Valores son: none (ninguna), ddl, mod (DML) y all (todas).

```
#log_hostname = off
```

Por defecto, en el log los mensajes de conexión son generados con la dirección IP de host entrante. Habilitar esta opción fuerza a postgres a buscar el reverso de la dirección IP. Dado que al timeout de una consulta de este tipo es altísimo comparado con los tiempos de respuesta requerido para una base de datos, se debe tener cautela al activar este parámetro.

2.2.7. Estadísticas de ejecución

Las siguientes opciones permiten configurar que estadísticas son recolectadas de forma constante por Postgres. Estas tienen la función de ayudar a administradores del sistema a tener una idea más acabada sobre que esta sucediendo en el DBMS. Si bien el apagar la producción de estadísticas conlleva una disminución en el overhead del servidor, esta es menor; el tradeoff de tener mayor información sobre que lo que ocurre al interior del motor suele ser más útil.

Más detalles en el capítulo 24 del manual de PostgreSQL 8.1.

```
#log_statement_stats = off
```

Habilita el log de las estadísticas completas de una consulta. Esta opción no puede habilitarse en conjunto con el log de estadísticas por-módulo de una consulta (parámetros siguientes)

```
#log_parser_stats = off
#log_planner_stats = off
#log_executor_stats = off
```

Habilitan el log de las estadísticas por modulo dentro de una consulta. Estos parámetros no pueden ser habilitados en conjunto con *log_statements_stats*.

```
#stats_start_collector = on
```

Controla si iniciar o no el subproceso colector de estadísticas.

```
#stats_command_string = off
```

Habilita la recolección de estadísticas en el comando en ejecución cada una de las sesiones activas. Los resultados son solo visibles al superusuario y al dueño de la sesión vía la vista de sistema *pg_stat_acivity*.

```
#stats_block_level = off
```

Habilita la recolección de estadísticas a nivel de bloque (I/O), asequibles vía la familia de vistas de sistema *pg_staty pg_statio*.

```
stats_row_level = on
```

Habilita la recolección de estadísticas a nivel de filas de tabla, asequibles vía la familia de vistas de sistema *pg_staty pg_statio*.

```
#stats_reset_on_server_start = off
```

Indica si resetear a cero las estadísticas recolectadas cada vez que el servidor se reinicia.

2.2.8. Parámetros de Autovacuum

A partir de la versión 8.1 de PostgreSQL, existe un subproceso separado llamado *demonio autovacuum*. Este es el encargado de revisar periódicamente la tablas con modificación considerables a su tuplas. Los datos utilizados por el demonio son aquellos generados al estar *stats_start_collector*i y *stats_row_level* encendidos. El proceso se conecta a la base de datos utilizando alguno de los slots de superusuario reservados.

El subproceso realiza 2 tareas: VACUUM y ANALYZE sobre las tablas que hayan alcanzado el nivel de cambio indicado por *autovacuum_vacuum_threshold* y *autovacuum_analyze_threshold* respectivamente. Mas detalles de estos comandos en la sección 3.10 del presente manual.

```
autovacuum = on
```

Flag que indica si ejecutar el proceso autovacuum.

```
#autovacuum_naptime = 60
```

Tiempo que duerme el proceso de autovacuum entre ejecuciones.

```
#autovacuum_vacuum_threshold = 1000
```

Umbral para determinar cuando se aplica VACUUM a una tabla. Cuando el valor calculado de la base más la cantidad de filas modificadas multiplicadas por el factor supera este valor, entonces se ejecuta VACUUM sobre la tabla.

```
#autovacuum_analyze_threshold = 500
```

Umbral para determinar cuando se aplica ANALYZE a una tabla. Cuando el valor calculado de la base más la cantidad de filas modificadas multiplicadas por el factor supera este valor, entonces se ejecuta ANALYZE sobre la tabla.

```
#autovacuum_vacuum_scale_factor = 0.4
```

Factor por el cual se multiplican la cantidad de filas modificadas de una tabla para obtener el umbral de ejecución de vacuum sobre una tabla.

```
#autovacuum_analyze_scale_factor = 0.2
```

Factor por el cual se multiplican la cantidad de filas modificadas de una tabla para obtener el umbral de ejecución de analyze sobre una tabla.

2.2.9. Parámetros de conexión de clientes por defecto

Los siguientes parámetros indican la configuración aplicada al contectarse al DBMS incluso vía consola.

```
#search_path = '$user,public'
```

El search path indica el orden en que los schemas son barridos en busca de las relaciones involucradas en una consulta. Este debe ser un listado separado por coma. Cuando se desea hacer referencia al esquema con el mismo nombre del usuario conectado se utiliza *$user*. Los esquemas pg_temp_nnn y pg_catalog son revisados aun cuando no aparezcan en el search_path; de no estar son revisados antes de los schemas indicados en este parámetro.

```
#default_tablespace = ''
```

Indica el tablespace por defecto a utilizar.

```
#check_function_bodies = on
```

Al habilitarse, se checkea la sintaxis del cuerpo de los procedimientos almacenados al momento de crearlos, entregando los errores correspondientes en caso de existir.

```
#datestyle = 'iso, mdy'
```

Indica el formato de fechas aceptado. El primer valor hace referencia al formato de output predeterminado (ISO, Postgres, SQL o German), mientras que el segundo establece el método de entrada/salida para dia-mes-año (DMY, MDY o YMD)

```
#timezone = unknown
```

Establece la zona horaria donde se encuentra el servidor. No ingresarse “unknown” se toma la zona horaria del sistema operativo.

2.2.10. Manejo de Locks

Esta sección permite ajustar 2 parámetros importantes dentro de la provisión de atomicidad.

```
#deadlock_timeout = 1000                # in milliseconds
```

Cantidad de tiempo cada cuanto se verifica si ha ocurrido un deadlock. Este procedimiento es lento y no conviene ejecutarlo muy seguido. En ambientes de producción, donde se asume que los deadlocks no son comunes, se recomienda aumentar el valor de manera de aumentar el desempeño del servidor; el valor a ingresar se recomienda sea al menos tiempo promedio de las transacciones reales, checkear antes solo provoca una pérdida de tiempo.

```
#max_locks_per_transaction = 64          # min 10
```

Valor utilizado para el cálculo del tamaño de la tabla compartida de locks. Este valor no es un límite duro de la cantidad de lock por sesión, sino un promedio. Aumentar el valor puede resultar en la solicitud de más memoria compartida de la que soporte el sistema operativo.

2.3. Permisos de acceso

2.3.1. Generalidades

Los permisos de acceso en postgres se realizan en 2 capas. La primera es aquella que controla que quien, como y desde donde puede conectarse a una base de datos; esto es configurado en el archivo **pg_hba.conf.** Por otra parte están los permisos particulares que tiene un usuario sobre las entidades de una base de datos; estos son manejados por los comandos GRANT y REVOKE.

2.3.2. Archivo pg_hba.conf

Los permisos de acceso en Postgres son manejados en el archivo **pg_hba.conf** el cual se encuentra en el directorio de trabajo de la base de datos o en */etc/postgres/8.1/main* en sistemas Debian-like. Cada línea en este archivo permite un tipo de acceso al servidor, el cual esta completamente cerrado inicialmente, hasta para el usuario *postgres*.

Existen 2 tipos posibles de conexión: *local* y *host*. Local hace referencia a conexiones realizadas a través de socket Unix, mientras que la segunda es a través de TCP/IP aun cuando sea a *localhost (127.0.0.1)*.

Hay 7 formatos para una fila de pg_hba.conf:

```
local      database  user  auth-method  [auth-option]
host       database  user  CIDR-address  auth-method  [auth-option]
hostssl    database  user  CIDR-address  auth-method  [auth-option]
hostnossl  database  user  CIDR-address  auth-method  [auth-option]
host       database  user  IP-address  IP-mask  auth-method  [auth-option]
hostssl    database  user  IP-address  IP-mask  auth-method  [auth-option]
hostnossl  database  user  IP-address  IP-mask  auth-method  [auth-option]
```

El primer parámetro indica el tipo de conexión. La primera fila indica el formato para conexiones vía socket unix, mientras que el resto se refiere a TCP/IP. En estas últimas se puede indicar si se requiere el uso de SSL (hostssl), no se desea en absoluto (hostnossl) o es opcional (host). Múltiples valores pueden ingresarse en esta columna separados por coma. Ejemplos a continuación.

```
local      base1                usuario          trust
local      base1,base2,base3    usuario          trust
```

El segundo parámetro es el nombre de la base de datos, mientras que el tercero es el nombre de usuario con el cual se esta conectando. Para ambos valores se puede utilizar la palabra reservada *all*, la cual hace referencia a "cualquier base de datos" o "cualquier usuario" según corresponda. Múltiples valores pueden ingresarse en esta columna separados por coma, sin espacios en blanco. Ejemplos a continuación.

```
local      base1      usuario                trust
local      base1      usuario,usuario2       trust
```

En el caso de conexiones TCP/IP, a continuación del usuario se especifica el segmento de red o IP sobre el cual aplica esta regla. El valor puede estar en formato CIDR o en dirección / máscara ya sea en IPv4 o IPv6. A continuación un ejemplo de una entrada análoga:

```
host            usuario    192.168.0.0/24                    trust
host            usuario    192.168.0.0      255.255.255.0    trust
```

El tipo de autenticación -- *auth-method* -- indica que método de seguridad aplicar a la conexión entrante. Las posibilidades a continuación.

Método	Descripción	Opciones
trust	Permite la conexión sin clave, *confiando* en ella.	Sin opciones
rejected	Rechaza la conexión inmediatamente.	Sin opciones
md5	Requiere que la clave de acceso sea enviada en MD5	Sin opciones
crypt	Requiere que la clave de acceso sea encriptada con el método *crypt()*. Solo recomendado para Postgres 7.2 o inferior, siendo md5 la alternativa en versiones actuales.	Sin opciones
password	Requiere clave de autenticación en texto plano.	Sin opciones
krb5	Autenticación vía Kerberos V5, solo disponible para conexiones TCP/IP. Su configuración se establece en postgres.conf.	Sin opciones
ident	Autenticación vía el usuario de sistema operativo que esta haciendo la consulta al archivo mapa de identidades provisto como parámetro.	Archivo de mapa de identidades. Si se omite, se utiliza pg_ident.conf. Al utilizar el mapa especial *sameuser*, la autenticación se realiza usando el mismo nombre entregado en el login.
pam	Autenticación vía PAM (Pluggable Authentication Modules) provisto por el sistema operativo. Recibe como parámetro el nombre del servicio contra el cual autenticar.	Nombre del servicio PAM contra el cual autenticar. Si se omite es 'postgres'

Las reglas en el archivo son aplicadas secuencialmente desde arriba hacia abajo; la primera fila que calce con el tipo de conexión, base de datos y usuarios utilizado, será ejecutada. En el siguiente se muestra un error típico donde se rechaza la conexión en la primera línea, no permitiendo que se revisen las siguientes.

```
host            usuario1    192.168.0.0/24          reject
host            usuario1    192.168.0.2             ident sameuser
host            usuario1    192.168.0.3             trust
local           all         reject
local           usuario4    md5
local           usuario3    md5
```

Aquí las reglas para permitir el acceso desde las direcciones 192.168.0.2 y 192.168.0.3 ni siquiera se alcanzan a leer puesto que la primera regla coincide y rechaza la conexión. Mismo caso con la conexión vía socket-unix, donde la primera regla rechaza la conexión aún cuando abajo hay permisos establecidos para algunos usuarios.

NOTA: En caso de eliminar la línea que autoriza al usuario *postgres* a ingresar libremente sin ingreso de password, no se podrán realizar las labores de mantención.

2.3.3. Archivo pg_hba.conf

Al utilizarse autenticación **ident** en el archivo pg_hba.conf, Postgres lee el archivo de identidades especificado en postgres.conf en la directiva *ident_file* que por defecto es pg_ident.conf en el mismo directorio de postgres.conf.

Este archivo contiene las correspondencias entre usuarios de postgres y usuarios del sistema operativo ambos asociados a un “mapname” o mapa de correspondencia. Para determinar cual usar, consulta al sistema operativo cual es el usuario de la sesión actual

Supongamos que se tiene la siguiente entrada en pg_hba.conf:

```
host            all   192.168.0.2           ident mapa_ejemplo
```

Esto quiere decir que cada vez que un usuario intente entrar a la base de datos “”, se ira a leer al archive pg_ident.conf, a buscar aquellas filas que tengan mapname “ejemplo” el nombre de usuario indicado para verificar si hay correspondencia y realizar la verificación de identidad contra el sistema operative. Un ejemplo de archivo pg_ident.conf para este caso seria el siguientes.

```
mapa_ejemplo        juanlinux       juanpsql
mapa_ejemplo        francisco       pancho
mapa_ejemplo2       fcojavier       pancho
```

El formato mapa – nombre SO – nombre Psql, hace la correspondencia entre uno y otro usuario. Cada mapa de nombres es independiente pudiéndose repetir los usuarios entre mapas.

2.3.4. Comandos GRANT y REVOKE

Los usuarios dueños de las entidades, por defecto quien las crea, poseen permisos completos sobre las mismas, pudiendo consultarlas, modificarlas y eliminarlas. El dueño puede decidir revocarse alguno de los privilegios por seguridad.

Para entregar permisos, se utiliza el comando SQL GRANT. Este en Postgres tiene la siguiente sintaxis

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] tablename [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
    ON DATABASE dbname [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTION funcname ( [ [ argmode ] [ argname ] argtype [, ...] ] ) [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE langname [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schemaname [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespacename [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT OPTION ]

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

El comando tiene 2 variantes básicas. La primera hace referencia a la entrega de permisos sobre entidades de la base de datos, mientras que la segunda incorpora a un usuario a un rol.

Al igual que los permisos de conexión entregados en pg_hba.conf, un usuario no dueño de una relación no tiene permiso alguno sobre esta; es necesario entregar uno a uno los privilegios.

Las variantes del comando GRANT que entregan privilegios sobre entidades de la base de datos, aplican sobre bases de datos, esquemas o tablas (inclúyase aquí índices, vistas, etc).

Privilegios posibles

SELECT	Autoriza SELECT en cualquier columna de la tabla, vista o secuencia en cuestión. Autoriza también el uso de COPY TO y el uso de *currval* para secuencias.
INSERT	Autoriza el INSERT en una tabla, así como el COPY FROM.
UPDATE	Autoriza el UPDATE de cualquier columna en la tabla indicada. SELECT FOR UPDATE también requiere de este privilegio. En secuencias, entrega acceso a *nextval* y *setval*.
DELETE	Autoriza el uso de DELETE en una tabla.
RULE	Autoriza la creación de RULEs en una tabla.
REFERENCES	Autoriza la creación de llaves foráneas en una tabla
TRIGGER	Autoriza la creación de Triggers en la tabla indicada.
CREATE	En bases de datos, autoriza la creación de esquemas, en esquemas la creación de nuevos objetos (tablas, vistas, etc). Para renombrar un objeto se debe ser dueño y además tener este permiso para el schema. Para tablespaces, permite la creación de tablas e índices, así como la creación de nuevas bases de datos con el tablespace en cuestión.
TEMPORARY ***TEMP***	Autoriza la creación y uso de tablas temporales.
EXECUTE	Autoriza la ejecución de la función indicada.
USAGE	Autoriza el uso de un determinado lenguaje procedural (Ej: plpgsql)
ALL PRIVILEGES	Entrega todos los permisos anteriores.

Al agregar el modificador WITH GRANT OPTION, entonces el receptor de los privilegios puede a su vez autorizar a otros usuarios a realizar la operación que él ha recibido.

En el caso de GRANT ROLE, permite entregar los permisos de rol a otro rol. Un rol es un usuario o grupo de la base de datos, que esta recibiendo los privilegios de otro grupo o usuario.

Para comprobar los permisos en un objeto, se utiliza el comando de consola \z OBJETO.

```
=> \z articulos
                    Access privileges for database ""
 Schema |   Name    | Type  |                Access privileges
--------+-----------+-------+---------------------------------------------------
 public | articulos | table | {yo=arwdRxt/yo,otro=r/ublog,otromas=rw/ublog}
(1 row)
```

Los permisos se interpretan de la siguiente manera.

```
            =xxxx -- privilegios entregados a PUBLIC (todos)
       usuario=xxxx -- privilegios entregados a un usuario
         grupo=xxxx -- privilegios entregados a un grupo

                  r -- SELECT ("leer")
                  w -- UPDATE ("escribir")
                  a -- INSERT ("escribir al final")
                  d -- DELETE
                  R -- RULE
                  x -- REFERENCES
                  t -- TRIGGER
                  X -- EXECUTE
                  U -- USAGE
                  C -- CREATE
                  T -- TEMPORARY
             arwdRxt -- ALL PRIVILEGES (para tablas)
                  * -- con opción grant en el privilegio precedente

               /yyyy -- usuario que concedió este privilegio
```

Los comandos GRANT que generan los permisos anteriores son:

```
GRANT SELECT ON articulos TO otro;
```

```
GRANT SELECT ON articulos TO otromas;
```

```
GRANT UDPATE ON articulos TO otromas;
```

Los distintos permisos deben colocarse en filas separas, pero un mismo permiso puede aplicarse a varias tablas o usuarios. Rescribiendo:

```
GRANT SELECT ON articulos TO otro, otromas;
```

```
GRANT UDPATE ON articulos TO otromas;
```

Revocar permisos es un procedimiento simple. Solo se debe aplicar el comando REVOKE al permiso específico que se desea remover.

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] tablename [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
    ON DATABASE dbname [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTION funcname ( [ [ argmode ] [ argname ] argtype [, ...] ] ) [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE langname [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schemaname [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespacename [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

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

Los privilegios pueden ser revocados por el dueño del objeto en cuestión o por usuarios que tenga WITH GRANT OPTION en ese objeto.

Ejemplos a continuación.

```
REVOKE INSERT ON articulos FROM otro;
```

```
REVOKE UPDATE ON articulos FROM otro, otromas;
```

Cuando se ha entregado un privilegio con WITH GRANT OPTION, este puede ser revocado independiente del permiso al cual aplica. En el siguiente ejemplo se muestra como se entrega el permiso con la opción de propagarlo, luego se elimina el permite de propagación pero el privilegio de SELECT continua.

```
=>GRANT SELECT ON articulos TO otro WITH GRANT OPTION;
GRANT
=>REVOKE GRANT OPTION FOR SELECT ON articulos FROM otro;
REVOKE
=>\z articulos
                           Access privileges for database ""
 Schema |   Name    | Type  |           Access privileges
--------+-----------+-------+---------------------------------------
 public | articulos | table | {yo=arwdRxt/yo, otro=a/yo}
(1 row)
```

2.4. Respaldo y Recuperación

2.4.1. Generalidades

PostgreSQL, al igual que la mayoría de los DBMS de buen nivel, trae incorporadas las herramientas adecuadas para realizar los respaldos de la base de datos y aquellas para poder reestablecer las bases desde los respaldos.

Los respaldos en PostgreSQL son realizados en SQL del motor, conteniendo el DDL y DML de una base de datos completa o de parte ella.

2.4.2. Respaldo

El respaldo se realiza a través de la herramienta **pg_dump**. Esta realiza una conexión al DBMS de la misma forma en que lo hace la consola **psql** para poder recuperar la base de datos o una porción de esta. Al utilizar la misma metodología de conexión, esta sujeta a las mismas reglas de autenticación que psql, debiéndose utilizar un usuario válido desde un host permitido.

Los parámetros de conexión son análogos a los de psql, agregando la información de base de datos a respaldar.

```
pg_dump [opciones] nombre_base_de_datos
```

A continuación, algunas líneas ejemplo tradicional.

```
pg_dump –Uusuario –hlocalhost
pg_dump –a –Uusuario –hlocalhost
pg_dump –d –Uusuario –hlocalhost  –t mi_tabla1
```

Un punto importante a considerar, es que por defecto pg_dump entrega los datos en formato COPY, es decir, separados con coma para insertar en una sola operación al restaurar. Si bien esto acelera considerablemente el desempeño tanto al respaldar como al restaurar, tiene el gran problema que no es posible detectar problemas de forma precisa; si ocurre un error en la línea 2 de 10.000, no será posible detectarlo sino hasta que termine. En bases de datos que contengan caracteres latinos o extraños, es poco recomendable dado los problemas de *encoding* que se suelen dar.

Parámetros interesantes:

-a	Entrega solo los datos, sin la información sobre el modelo (CREATE TABLE)
-s	Entrega solo el modelo de datos, sin datos.
-d	Entrega los datos en formato INSERT INTO en lugar que por COPY
-t TABLA	Entrega solo la tabla indicada, en lugar de la base completa.
-f ARCHIVO	Escribe la salida del comando al archivo indicado

Al ejecutarse el comando, el respaldo de la base de datos será entregado al *stdout*. Para almacenarlo en un archivo se utiliza alguno de las siguientes alternativas:

```
pg_dump base > archivo.sql
```

```
pg_dump –f archive.sql base
```

2.4.3. Recuperación

La recuperación de un archivo de respaldo es tan simple como la ejecución del script SQL dentro de una consola (**\i archivo.sql**) o alimentándolo como stdin.

Ejecución vía consola

```
usuario@debian:~$ psql base
Welcome to psql 8.1.10, the PostgreSQL interactive terminal.

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

base=> \i respaldo.sql
```

Alimentación vía stdin

```
usuario@debian:~$ psql base < respaldo.sql
```

3. Base de datos

3.1. Creación / Eliminación Base de datos

3.1.1. Generalidades

Las operaciones administrativas básicas como lo son la creación de usuarios, bases de datos, etc; pueden ser realizadas en línea de comando a través del usuario de sistema "postgres", así como por consola PSQL. A continuación se revisará como son los comandos utilizando el usuario de sistema postgres.

Todas las operaciones a continuación DEBEN ser realizadas con el usuario *postgres*.

Todos los comandos de esta sección soportan los modificadores de conexión de *psql* descritos en 2.1.3.

3.1.2. Creación de usuarios

El comando **createuser** permite la creación de nuevos roles en Postgres. Los roles hacen referencia de forma genérica a un usuario o a un grupo. Su sintaxis a continuación.

```
createuser [opciones] … nombre_rol
```

Opciones Interesantes

-s	Rol será superusaurio
-d	Rol puede crear bases de datos
-r	Rol puede crear otros roles
-P	asigna al rol su clave. Puede ir en blanco, caso en el cual solo podrá logearse con TRUST.

En caso de no proporcionarse alguna de las opciones anteriores, excepto –P, el sistema lo preguntara interactivamente de la siguiente forma.

```
postgres@debian:~$ createuser ejemplo
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
CREATE ROLE
```

3.1.3. Creación de una base de datos

Para crear una base de datos en el sistema, se utiliza el comando **createdb**. Sintaxis a continuación.

```
createdb [opciones] [nombre_base_de_datos] [descripción]
```

Interesantemente el nombre de la base de datos puede ir en blanco, asumiéndose que esta tendra el mismo nombre que el usuario de sistema actual.

Opciones Interesantes

-D Indica el tablespace de esta base de datos

-E Indica la codificación de la base de datos. El detalle de los distintos juegos de caracteres que soporta Postgres están en la sección 21.2.1 del manual de la versión 8.1. Alguno son: LATIN1 (ISO88591), UTF-8, SQL_ASCII.

-O Indica el usuario dueño de la base de datos. De omitirse se asume *postgres*.

Ejemplo de la creación de la base de datos “ejemplo” cuyo dueño es el usuario “ejemplo”:

```
postgres@debian:/etc/postgresql/8.1/main$ createdb -Oejemplo
CREATE DATABASE
```

Ejemplo de la creación de la base de datos “base1” cuyo dueño es el usuario “ejemplo”:

```
postgres@debian:/etc/postgresql/8.1/main$ createdb -Oejemplo base1
CREATE DATABASE
```

3.1.4. Creación de lenguajes procedurales

Por defecto al momento de crear un base de datos, esta no tiene habilitado el uso de lenguajes procedurales como lo es el **plpgsql**. Para poder incorporarlos basta con ejecutar el siguiente comando.

```
createlang lenguaje nombre_base_de_datos
```

Ejemplo:

```
postgres@debian:/etc/postgresql/8.1/main$ createlang plpgsql mibase
postgres@debian:/etc/postgresql/8.1/main$
```

Como se puede apreciar, el comando no lanza salida alguna de completarse correctamente.

3.2. Tipos de datos

3.2.1. Generalidades

Postgres pone a disposición del usuario una gran variedad de tipos de datos pensados en cubrir necesidades específicas. Suele suceder que al modelar una base de datos se utilizan los tipos conocidos y no se da provecho a las características especiales de Postgres; mientras esto provee compatibilidad, es menudo más conveniente sacar provecho a aquellas definiciones adicionales del motor.

A continuación se describen los tipos de datos mas utilizados y la sugerencia de cuando es conveniente utilizarlos.

3.2.2. Tipos String

Tipo	Descripción
char(n)	Datos de caracteres no Unicode de longitud fija
Text	Cantidad no predefinida
varchar(n)	Datos de caracteres no Unicode de longitud variable.

El tipo *char(n)* almacena n caracteres en formato ASCII, un byte por cada letra. Cuando almacenamos datos en el tipo char, siempre se utilizan los *n* caracteres indicados, incluso si la entrada de datos es inferior. Por ejemplo, si en un *char(5)*, guardamos el valor **'A'**, se almacena **'A ',** ocupando los cinco bytes.

Por su parte *varchar(n)* almacena n caracteres en formato ASCII, un byte por cada letra. Cuando almacenamos datos en el tipo varchar, únicamente se utilizan los caracteres necesarios, Por ejemplo, si en un *varchar(255)*, guardamos el valor **'A'**, se almacena **'A'**, ocupando solo un byte.

El tipo de dato *text* permite el almacenamiento de un string en cualquier formato de largo no predefinido. Este es el tipo más versátil de string. Considerando que hoy en día los espacios de almacenamiento son enormes, este tipo se recomienda en todo caso por sobre varchar. Cabe señalar que dado que text no tiene límite predefinido, entonces al momento de llevar los datos a otros sistemas con largos predefinidos no hay aviso previo de que estamos sobrepasando algún largo establecido, siendo este un punto a considerar al momento de seleccionar el tipo de dato.

3.2.3. Tipos Numéricos

Tipo	Tamaño	Descripción	Rango
bool		Booleano	trae / false – (‘t’ / ‘f’)
decimal	variable	Especificadas por el usuario	no limite
float4	4 bytes	Número de punto flotante con precisión	6 lugares de decimales
float8	8 bytes	Número de punto flotante con doble precisión	15 lugares de decimales
bigint	8 bytes	Valores enteros.	-9,233,372,036,854,775,808 a +9,233,372,036,854,775,807
int2	2 bytes	Precisión fija (entero)	-32768 a +32767
int4	4 bytes	opción para determinada precisión (entero)	-2147483648 a +2147483647
int8	8 bytes	amplia gama determinada de precisión (entero)	+/- > 18 lugares de decimales
numeric	variable	Especificadas por el usuario	no limite
serial	4 bytes	Identicador o referencia cruzada	0 a +2147483647

Postgres contiene los tipos de datos nativos comúnmente conocidos ***int*** y ***float***, para números enteros y reales respectivamente. Estos dependiendo del dígito que acompañe su nombre, indica la precisión que soporta la columna.

Para grandes números se utiliza *bigint*, el cual suele ser utilizado para las columnas de identificadores.

Cuando se requiere de gran precisión en números, virtualmente sin limitación (1000 dígitos), el tipo a utilizar es sin duda ***numeric***. Este es recomendado para valores monetarios u cualquier dato donde la precisión es crucial. Este tipo además soporta el valor ‘NaN’ (Not A Number – No Un Numero) para indicar tal caso, además de NULL.

El tipo de dato SERIAL es en realidad un atajo. En Postgres no existe un tipo autoincremental como es el caso de MySQL. Aquí por un lado se genera la columna con tipo BIGINT asociando al valor por defecto el siguiente valor de una secuencia (Ver punto 3.6) creada especialmente, provocándose el efecto de auto incremento.

3.2.4. Tipos de Fecha y Hora

Tipo	Descripción
date	Fecha, con día, mes y año
time	Hora, minuto, segundos
timestamp(n)	Fecha y hora con milisegundos. El parámetro *n*, indica la cantidad de milisegundo a almacenar.

En tipos de datos para fecha y hora, Postgres conserva los tipos tradicionales. Aquí el único parámetro interesante es aquel que se puede pasar a timestamp para indicar la cantidad de milisegundos a almacenar que por defecto son 6.

3.3. Esquemas (Schemas)

3.3.1. Generalidades

PostgreSQL provee un mecanismo para poder agrupar objetos dentro de una misma base de datos, de manera permitir un mayor orden en modelos con un gran número de elementos. El uso de esquemas permite, entre otras cosas, que muchos usuarios interactúen con una misma base de datos sin interferirse, organizar objetos lógicamente según su relación, tener tablas de un mismo nombre en una misma base de datos sin toparse, etc.

Los esquemas son grupos de objetos. Por defecto PostgreSQL provee esquemas que existen en toda base de datos:

- public: contexto publico por defecto donde se crean los objetos si no se especifica otro.
- information_schema: conjunto de vistas que contienen información sobre los objetos definidos en la base de datos actual.
- pg_catalog: schema que mantiene las tablas del sistema y los tipos de datos, funciones y operadores incorporados en PostgreSQL.

3.3.2. Creación y uso

Los esquemas pueden ser creados por el usuario dueño de la base de datos, aquellos que tengan permisos de superusuario sobre la base y los superusuarios; sintaxis a continuación.

```
CREATE SCHEMA nombre [AUTHORIZATION usuario];
```

Si se omite el nombre de esquema y se provee authorization, entonces se crea un esquema con el mismo nombre del usuario.

Para acceder a un objeto dentro de un esquema, se utiliza la sintaxis **esquema.objeto**.

Ejemplos:

```
Mibase=> CREATE TABLE miesquema.tabla ( ... );
Mibase=> DROP TABLE miesquema.tabla;
Mibase=> CREATE SEQUENCE miesquema.secuencia;
Mibase=> SELECT * FROM miesquema.tabla as t1, miotroesquema.tabla as t2 WHERE t1.id = t2.id ;
```

Para eliminar un esquema se utiliza DROP SCHEMA.

```
Mibase=> DROP SCHEMA miesquema
```

Si se tiene dependencia de las tablas del esquema a eliminar y se quiere propagar la eliminación, se usa CASCADE.

```
MiBase=> DROP SCHEMA miesquema CASCADE;
```

3.3.3. Search Path

Cuando en una consulta se hace referencia a un objeto (tabla, vista, etc), el nombre de este se busca dentro de los esquemas definidos dentro de la variable ***search_path***. Esta contiene el nombre de los esquemas que el usuario puede acceder sin hacer referencia explicita, y el orden en el cual se realizará la búsqueda.

Un search_path por defecto contiene: $user, public. Esto significa que por defecto, PostgreSQL busca en el esquema con el mismo nombre del usuario que esta conectado (si no existe, se pasa al siguiente) y a continuación intenta el esquema publico. Adicionalmente, y de forma transparente, PostgreSQL busca en el esquema de sistema pg_catalog antes de $user, de manera que las tablas de sistemas siempre estén accesibles; colocar explícitamente pg_catalog al final de la línea o en otra parte, permitirá crear tablas que reemplacen a aquellas del sistema al momento de intentar de acceder sin especificar esquema.

Para mostrar el search_path actual, se usa la siguiente instrucción.

```
mibase=> SHOW search_path;
 search_path
--------------
 $user,public
(1 row)
```

El search_path puede ser modificado en postgres.conf para entregar un valor por defecto a todos los usuarios, así como también dentro de la sesión a través del siguiente comando.

```
mibase=> SET search_path TO miesquema, public;
SET
```

3.4. Tablas (DDL – Data Definition Language)

3.4.1. Creación

3.4.1.1. Generalidades

En Postgres existen dos formas de crear una tabla: declarándola y seleccionando un set de datos vía query.

Para crear una tabla vía declaración, se tiene la siguiente sintaxis en forma simplificada. Referirse al manual de Postgres para la sintaxis completa.

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name ( [
  { column_name data_type [ DEFAULT default_expr ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE parent_table [ { INCLUDING | EXCLUDING } DEFAULTS ] }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace ]
```

La cláusula opcional INHERITS especifica una colección de nombres de tabla de las cuales esta tabla hereda todos los campos. Si algún campo heredado aparece más de una vez, Postgres informa de un error. Postgres permite automáticamente a la tabla creada heredar funciones de las tablas superiores a ella en la jerarquía de herencia. Notar que cuando se utiliza INHERITS, la tabla que hereda cambia si la tabla padre lo hace.

A través de LIKE se puede obtener el mismo resultado que INHERITS, pero con la salvedad que la tabla creada no cambia cuando el padre lo hace.

La opción ON COMMIT permite establecer el comportamiento de las tablas temporales en una transacción.

- PRESERVE ROWS indica que ninguna acción es realizada (por defecto).
- DELETE ROWS elimina todas las filas de la tabla.
- DROP elimina la tabla completa.

Ejemplo:

```
CREATE TABLE miesquema.ejemplo1 (
       Id SERIAL PRIMARY KEY,
       Texto text
);
```

Esto creará la tabla *ejemplo1* en el esquema *miesquema* con las 2 columnas especificadas, siendo la llave primaria la columna *id*.

```
CREATE TEMP TABLE temporal1 (
       Id SERIAL PRIMARY KEY,
       Texto text
);
```

Esto creará la tabla temporal *temporal1* con la misma estructura de la anterior. Nótese que aquí no especificamos esquema puesto que las tablas temporales existen en un esquema especial para este efecto.

3.4.1.2. Opciones de tabla y table_constraints

La opción *TEMPORARY* indica que la tabla se crea solo para esta sesión, y es eliminada automáticamente con el fin de la sesión. Las tablas permanentes existentes con el mismo nombre no son visibles mientras la tabla temporal existe. En Postgres los modificadores GLOBAL y LOCAL no tienen efecto, siendo proporcionados solo por compatibilidad.

Mediante la sección de CONSTRAINT se pueden especificar restricciones de una forma alternativa a crearlas directamente en cada fila, obteniéndose el mismo resultado.

```
[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ... ] ) [ USING INDEX TABLESPACE tablespace ] |
  PRIMARY KEY ( column_name [, ... ] ) [ USING INDEX TABLESPACE tablespace ] |
  CHECK ( expression ) |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

- UNIQUE permite la creación de un índice en la(s) columna(s) indicada(s) que fuerza a que no existan valores repetidos
- PRIMARY KEY define las columnas que componen la llave primaria
- CHECK permite la definición de reglas a verificar al momento de insertar o modificar la información de la tabla. La expresión en esta opción debe evaluar booleano.
- FOREIGN KEY establece las llaves foráneas de la tabla.
- DEFERRABLE o NOT DEFERRABLE establece que esta condición puede ser chequeada al final de una transacción en lugar de en cada fila. Actualmente solo FOREIGN KEY soporta NOT DEFERRABLE.

Ejemplo:

```
CREATE TABLE ciudadanos (
       ciudadano_id SERIAL,
       ciudadano_nombre text,
       ciudadano_rut VARCHAR(10),
       PRIMARY KEY (ciudadano_id),
       CONSTRAINT rut UNIQUE (ciudadano_rut),
       CONSTRAINT check1 CHECK ( ciudadano_nombre <> '' ciudadano_rut <> ''),
       CONSTRAINT FOREIGN KEY (ciudadano_rut) REFERENCES otra_tabla (persona_rut),
);
```

3.4.1.3. Opciones de columna y column constraints

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  UNIQUE [ USING INDEX TABLESPACE tablespace ] |
  PRIMARY KEY [ USING INDEX TABLESPACE tablespace ] |
  CHECK (expression) |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

Cada columna se especifica como “nombre tipo”, pudiendo el tipo incluir especificaciones de array (Ver ejemplos). Adicionalmente cada columna puede tener las siguientes opciones.

- NOT NULL que indica que su valor no puede ser nulo
- UNIQUE, su valor no se puede repetir en la tabla
- DEFAULT ‘valor’, dice que valor tiene la columna en caso de proporcionarse NULL al insertar.
- “column_constraint” permite indicar una restricción a los valores que esta puede tomar.
- PRIMARY KEY: indica que la columna es parte de la llave primaria de la tabla.

Ejemplo: (Análogo al ejemplo anterior)

```
CREATE TABLE ciudadanos (
    ciudadano_id SERIAL PRIMARY KEY,
    ciudadano_nombre text CHECK (ciudadano_nombre <> ''),
    ciudadano_rut VARCHAR(10) UNIQUE CHECK (ciudadano_rut <> '') REFERENCES otra_tabla (persona_rut),
);
```

3.4.1.4. Creación de tabla vía SELECT

Existe 2 formas de crear tablas a partir de un set de resultados: SELECT INTO y CREATE TABLE AS.

SELECT INTO crea una nueva tabla a partir del resultado de una query. Típicamente, esta query recupera los datos de una tabla existente, pero se permite cualquier query de SQL.

```
SELECT [ ALL | DISTINCT [ ON ( expresión [, ...] ) ] ]
    expresión [ AS nombre ] [, ...]
    [ INTO [ TEMPORARY | TEMP ] [ TABLE ] nueva_tabla ]
    [ FROM tabla [ alias ] [, ...] ]
    [ WHERE condición ]
    [ GROUP BY columna [, ...] ]
    [ HAVING condición [, ...] ]
    [ { UNION [ ALL ] | INTERSECT | EXCEPT } select ]
    [ ORDER BY columna [ ASC | DESC | USING operador ] [, ...] ]
    [ FOR UPDATE [ OF Nombre_de_clase [, ...] ] ]
    LIMIT { contador | ALL } [ { OFFSET | , } incio ]
```

Ejemplo:

```
SELECT clientes.*, cuentas.saldo INTO clientes_premium
FROM cuentas, clientes
WHERE cuentas.cliente_rut = clientes.cliente_rut
AND cuentas.saldo > 10000000;
```

CREATE TABLE AS produce el mismo efecto que el comando anterior con otra sintaxis.

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name
    [ (column_name [, ...] ) ] [ [ WITH | WITHOUT ] OIDS ]
    AS query
```

Ejemplo:

```
CREATE TABLE clientes_premium AS
SELECT clientes.*, cuentas.saldo INTO clientes_premium
FROM cuentas, clientes
WHERE cuentas.cliente_rut = clientes.cliente_rut
AND cuentas.saldo > 10000000;
```

3.4.2. Modificación de Tablas

Para modificar una tabla, se utiliza el comando ALTER TABLE, sintaxis a continuación.

```
ALTER TABLE [ ONLY ] name [ * ]
    action [, ... ]
ALTER TABLE [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column TO new_column
ALTER TABLE name
    RENAME TO new_name
ALTER TABLE name
    SET SCHEMA new_schema
```

Donde “action” puede ser alguna de las siguientes opciones.

```
    ADD [ COLUMN ] column type [ column_constraint [ ... ] ]
    DROP [ COLUMN ] column [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column TYPE type [ USING expression ]
    ALTER [ COLUMN ] column SET DEFAULT expression
    ALTER [ COLUMN ] column DROP DEFAULT
    ALTER [ COLUMN ] column { SET | DROP } NOT NULL
    ADD table_constraint
    DROP CONSTRAINT constraint_name [ RESTRICT | CASCADE ]
    { DISABLE | ENABLE } TRIGGER [ trigger_name | ALL | USER ]
    OWNER TO new_owner
```

Revisemos algunas de las opciones. Para el listado completo de opciones referirse al manual PostgreSQL.

ADD COLUMN	Agrega una columna a la tabla al final. Para sintaxis referirse a sección 3.4.1.
DROP COLUMN	Elimina una columna de la tabla. Notar que esto no elimina físicamente los datos del sistema de archivos; se debe llamar a VACUUM.
ALTER COLUMN TYPE	Cambia el tipo de dato de la columna propaganda el cambio a los constraints para la columna. Agregando USING se puede indicar una expresión de conversión a aplicar, de lo contrario se utilizará la conversión por defecto interna de Postgres
SET/DROP DEFAULT	Permite setear o eliminar el valor por omisión de una columna.
SET/DROP NOT NULL	Permite indicar si la columna acepta valores NULL o no. Se setearse, se debe asegurar que ninguna fila tenga valor NULL o haber especificado el valor por defecto.
ADD table_constraint	Agrega una restricción a la tabla. Detalles en secciones 3.4.1.2 y 3.4.1.3.
DROP CONSTRAINT	Elimina una restricción de la tabla.
DISABLE/ENABLE TRIGGER	Permite habilitar o deshabilitar un trigger específico ya sea a un usuario o a todos.
OWNER	Cambia el dueño de la tabla
RENAME	Cambia el nombre de la columna o la tabla dependiendo del caso.
SET SCHEMA	Cambia el esquema al cual pertenece la tabla.

Ejemplos:

```
ALTER TABLE tabla RENAME TO tabla1;
```

```
ALTER TABLE tabla1 ALTER COLUMN columna1 RENAME TO columna2;
```

```
ALTER TABLE tabla1 ADD COLUMN nueva_columna NUMERIC DEFAULT 0;
```

```
ALTER TABLE tabla1 ALTER COLUMN nueva_columna SET NOT NULL;
```

```
ALTER TABLE tabla1 DROP COLUMN nueva_columna CASCADE;
```

```
ALTER TABLE tabla1 ADD CHECK ( valor > 0 );
```

```
ALTER TABLE tabla1 DROP tabla1_valor_check;
```

3.4.3. Eliminación de Tablas

DROP TABLE [Eliminar Tabla] -- Elimina tablas de una base de datos

```
DROP TABLE name [, ...] [ CASCADE | RESTRICT ]
```

Nombre: nombre de una tabla a eliminar. Pueden proveerse varios nombres separados por coma.

Al indicar CASCADE, se eliminarán además los objetos que dependen de esta tabla, como las vistas asociadas (por defecto es RESTRICT, que no propaga la eliminación).

A diferencia de DELETE, DROP borra completamente (datos y definiciones de la tabla), no solo los datos de la tabla (rows) como el DELETE.

Ejemplo:

```
DROP TABLE mitabla;
```

```
DROP TABLE mitabla1, mitabla2 CASCADE;
```

3.5. Consultas

3.5.1. Generalidades

Una de las características más interesantes que los distinguía hace unos años de otras bases de datos gratuitas como MySQL, era su capacidad para poder realizar complejas consultas, incorporando elementos que hasta el momento solo bases de datos comerciales como Oracle tenían.

Hoy en día el motor de consultas sigue siendo muy poderoso, aunque otros DBMS han logrado incorporar la mayoría de las características de Postgres.

A continuación se muestra la sintaxis de una consulta SELECT SQL en Postgres.

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
    [ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] ]
```

Donde from_item puede ser uno de los siguientes:

```
    [ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
    function_name ( [ argument [, ...] ] ) [ AS ] alias [ ( column_alias [, ...]
| column_definition [, ...] ) ]
    function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
    from_item [ NATURAL ] join_type from_item [ ON join_condition | USING (
join_column [, ...] ) ]
```

En terminos más simples:

```
SELECT elementos FROM tablas WHERE condiciones ORDER BY criterio LIMIT
cantidad_máxima.
```

Donde *elementos* son las columnas de *tablas* a recuperar, las cuales cumplen las *condiciones*. Al terminar la operación son ordenadas por *criterio* y finalmente se entrega *cantidad_maxima* de filas en el set de resultados.

3.5.2. Consultas Simples

Las consultas más simples son aquellas donde se unen algunas tablas, recuperando algunas de las filas de cada tabla involucrada.

Sean las siguientes tablas en una base de datos

Clientes	Sucursales	Ventas
Cliente_rut	Sucursal_id	Sucursal_id
Cliente_nombre	Surcursal_nombre	Cliente_rut
Cliente_telefono	Sucursal_direccion	Fecha
		Monto

Si se desea conocer todas las ventas de todas las sucursales, con fecha superior al 10 de enero de 2008, entregando solo los 10 mayores montos; se puede hacer la siguiente consulta SELECT SQL.

```
SELECT
        sucursal_nombre,
        sucursal_direccion,
        Fecha,
        cliente_nombre,
        cliente_teléfono,
        monto
FROM
        Clientes, Sucursales, Ventas
WHERE
        Clientes.cliente_rut = Ventas.cliente_rut
        Sucursales.sucursal_id = Ventas.sucursal_id
ORDER BY Monto DESC, Fecha ASC LIMIT 10;
```

3.5.3. Consultas con agregación

Las funciones de agregación son aquellas que permiten agrupar datos bajo un criterio obteniéndose un valor representativo. Funciones comunes son:

- min – mínimo del valor de la columna al cual se aplica
- max – máximo del valor de la columna al cual se aplica
- count – número de resultados obtenidos. No se ve afectado por la columna al cual se aplica.
- sum – Suma de los valores de la columna indicada.
- avg – promedio de los valores de la columna indicada.

Ejemplos de consultas con agregación:

Máximo de un folio:

```
SELECT max(folio) FROM ordenes_compra;
```

Mínimo precio de las acciones de LAN Chile en la bolsa:

```
SELECT min(precio) FROM acciones WHERE nemotecnico = 'LAN';
```

Suma de las ventas por fecha del mes de enero:

```
SELECT fecha, sum(monto) FROM ventas WHERE fecha >= '2008-01-01' AND fecha <
'2008-02-01' GROUP BY fecha;
```

En esta última se hace uso de la sentencia GROUP BY, la cual como indica su nombre *agrupa* los resultados bajo alguna columna, permitiendo que todos los datos asociados a usa columna se *resuman* mediante una funciona de agregación. Se debe usar GROUP BY cada vez que acompañemos una función de agregación de otras columnas, debiendo aparecer todas estas en la cláusula de agregación.

3.5.4. Consultas Anidadas

Una de las características más útiles dentro del lenguaje SQL, es la posibilidad de anidar consultas para simplificar la sentencia o poder realizar operaciones que de otra manera serían imposibles o requerirían de la creación de tablas temporales explicitas.

Las consultas anidadas funcionan para cualquier motivo igual que una tabla. Aquí se da el nombre "tabla2" al resultado.

```
SELECT * FROM (SELECT * FROM tabla1) as tabla2;
```

Es necesario darle un nombre cada subconsulta para que Postgres pueda identificarlo y referenciarlo inequívocamente.

3.5.5. Join

Join es un operador utilizado para unir tablas en una consulta. Para explicarlo se utilizara un ejemplo con tres tablas que se muestran a continuación.

```
clase=> select * from
tabla1;
 id | dato1
----+-------
  1 | aaa
  1 | bbb
  2 | ccc
  2 | ddd
  2 | eee
  6 | fff
  7 | ggg
(7 rows)
```

```
clase=> select * from tabla2;
 id |   dato2
----+------------
  1 | 2008-07-12
  1 | 2008-03-14
  2 | 2008-09-24
  2 | 2008-07-30
  2 | 2008-04-22
  3 | 2008-01-11
  4 | 2008-05-03
(7 rows)
```

```
clase=> select * from tabla3;
 id | dato3 | dato4
----+-------+-------
  1 | t     |   2.4
  1 | t     |   3.6
  3 | f     |   2.8
  5 | t     |   2.4
(4 rows)
```

3.5.5.1. Inner Join

El *Inner Join* se utiliza para unir 2 tablas en base a un columna. En este tipo, se unirán datos de la primera tabla que tengan una entrada en la segunda tabla, entregando como resultado filas en las que solamente hay datos que se puedan unir entre las dos tablas a través de un valor común. La sintaxis utilizada es la siguiente

```
SELECT [datos] FROM [tabla1] INNER JOIN [tabla2] ON tabla1.parametro =
tabla2.parametro;
```

Utilizando las dos primeras tablas que tenemos de ejemplo se obtiene lo siguiente:

```
clase=> SELECT * FROM tabla1 INNER JOIN tabla2 ON tabla1.id = tabla2.id;
 id | dato1 | id |   dato2
----+-------+----+------------
  1 | aaa   |  1 | 2008-07-12
  1 | aaa   |  1 | 2008-03-14
  1 | bbb   |  1 | 2008-07-12
  1 | bbb   |  1 | 2008-03-14
  2 | ccc   |  2 | 2008-09-24
  2 | ccc   |  2 | 2008-07-30
  2 | ccc   |  2 | 2008-04-22
  2 | ddd   |  2 | 2008-09-24
  2 | ddd   |  2 | 2008-07-30
  2 | ddd   |  2 | 2008-04-22
  2 | eee   |  2 | 2008-09-24
  2 | eee   |  2 | 2008-07-30
  2 | eee   |  2 | 2008-04-22
(13 rows)
```

Utilizar el INNER JOIN logra mezclar todas las entradas de tabla1 con todos las entradas de tabla2 en las que el parámetro id sea igual. También se puede ver como, por ejemplo, se tienen 9 resultados con el id = 2. Esto es por la cantidad de entradas con id = 2 que hay en la tabla1 (3 entradas) y la cantidad que hay en la tabla2 (3 entradas). Siempre la cantidad será la multiplicación de la cantidad de la tabla1 multiplicado por la cantidad de la tabla2, para cada parámetro coincidente.

3.5.5.2. Outer Join

El *Outer Join* es una extensión del *Inner Join*, pero que permite agregar valores al set de resultados. La sintaxis utilizada es la siguiente:

```
SELECT [datos] FROM [tabla1] [LEFT | RIGHT | FULL] OUTER JOIN [tabla2] ON
tabla1.parametro = tabla2.parametro;
```

Si aplicamos un *Left Outer Join* a las tablas de ejemplo se obtiene el siguiente resultado:

```
clase=> SELECT * FROM tabla1 LEFT OUTER JOIN tabla2 ON tabla1.id = tabla2.id;
 id | dato1 | id |   dato2
----+-------+----+------------
  1 | aaa   |  1 | 2008-07-12
  1 | aaa   |  1 | 2008-03-14
  1 | bbb   |  1 | 2008-07-12
  1 | bbb   |  1 | 2008-03-14
  2 | ccc   |  2 | 2008-09-24
  2 | ccc   |  2 | 2008-07-30
  2 | ccc   |  2 | 2008-04-22
  2 | ddd   |  2 | 2008-09-24
  2 | ddd   |  2 | 2008-07-30
  2 | ddd   |  2 | 2008-04-22
  2 | eee   |  2 | 2008-09-24
  2 | eee   |  2 | 2008-07-30
  2 | eee   |  2 | 2008-04-22
  6 | fff   |    |
  7 | ggg   |    |
(15 rows)
```

Como podemos apreciar, se obtienen los mismos resultados del *Inner Join*, pero agregando aquellos datos de la tabla1 que no se encuentran en la tabla2, en este caso las entradas con Id = 6 y 7. Al no tener valores en la tabla2, se agregan valores en NULL para las columnas de resultado.

Si aplicamos un *Right Outer Join* a las tablas de ejemplo se obtiene el siguiente resultado:

```
clase=> SELECT * FROM tabla1 RIGHT OUTER JOIN tabla2 ON tabla1.id = tabla2.id;
 id | dato1 | id |   dato2
----+-------+----+------------
  1 | aaa   |  1 | 2008-07-12
  1 | aaa   |  1 | 2008-03-14
  1 | bbb   |  1 | 2008-07-12
  1 | bbb   |  1 | 2008-03-14
  2 | ccc   |  2 | 2008-09-24
  2 | ccc   |  2 | 2008-07-30
  2 | ccc   |  2 | 2008-04-22
  2 | ddd   |  2 | 2008-09-24
  2 | ddd   |  2 | 2008-07-30
  2 | ddd   |  2 | 2008-04-22
  2 | eee   |  2 | 2008-09-24
  2 | eee   |  2 | 2008-07-30
  2 | eee   |  2 | 2008-04-22
    |       |  3 | 2008-01-11
    |       |  4 | 2008-05-03
```

En este caso, tenemos los resultados del *Inner Join* sumados a las entradas de la tabla2 que no se encuentran en la tabla1, id = 3 y 4. Al igual que el *Left Inner Join*, aquellos valores que no se encuentran en tabla1 son reemplazados con valores en NULL.

Si aplicamos un *Full Outer Join* a las tablas de ejemplo se obtiene el siguiente resultado:

```
clase=> SELECT * FROM tabla1 FULL OUTER JOIN tabla2 ON tabla1.id = tabla2.id;
 id | dato1 | id |   dato2
----+-------+----+------------
  1 | aaa   |  1 | 2008-07-12
  1 | aaa   |  1 | 2008-03-14
  1 | bbb   |  1 | 2008-07-12
  1 | bbb   |  1 | 2008-03-14
  2 | ccc   |  2 | 2008-09-24
  2 | ccc   |  2 | 2008-07-30
  2 | ccc   |  2 | 2008-04-22
  2 | ddd   |  2 | 2008-09-24
  2 | ddd   |  2 | 2008-07-30
  2 | ddd   |  2 | 2008-04-22
  2 | eee   |  2 | 2008-09-24
  2 | eee   |  2 | 2008-07-30
  2 | eee   |  2 | 2008-04-22
    |       |  3 | 2008-01-11
    |       |  4 | 2008-05-03
  6 | fff   |    |
  7 | ggg   |    |
(17 rows)
```

Aquí se obtienen los resultados de un *Inner Join*, más las entradas que aportan el *Left* y *Outer Join* juntos.

El *Join* se puede aplicar a más de una tabla y mezclado de la manera que resulte conveniente, como se muestra a continuación:

```
clase=> SELECT * FROM tabla1 INNER JOIN tabla2 ON tabla1.id = tabla2.id LEFT OUTER JOIN tabla3 ON
tabla1.id = tabla3.id;
 id | dato1 | id |   dato2    | id | dato3 | dato4
----+-------+----+------------+----+-------+-------
  1 | aaa   |  1 | 2008-07-12 |  1 | t     |   2.4
  1 | aaa   |  1 | 2008-07-12 |  1 | t     |   3.6
  1 | aaa   |  1 | 2008-03-14 |  1 | t     |   2.4
  1 | aaa   |  1 | 2008-03-14 |  1 | t     |   3.6
  1 | bbb   |  1 | 2008-07-12 |  1 | t     |   2.4
  1 | bbb   |  1 | 2008-07-12 |  1 | t     |   3.6
  1 | bbb   |  1 | 2008-03-14 |  1 | t     |   2.4
  1 | bbb   |  1 | 2008-03-14 |  1 | t     |   3.6
  2 | ccc   |  2 | 2008-09-24 |    |       |
  2 | ccc   |  2 | 2008-07-30 |    |       |
  2 | ccc   |  2 | 2008-04-22 |    |       |
  2 | ddd   |  2 | 2008-09-24 |    |       |
  2 | ddd   |  2 | 2008-07-30 |    |       |
  2 | ddd   |  2 | 2008-04-22 |    |       |
  2 | eee   |  2 | 2008-09-24 |    |       |
  2 | eee   |  2 | 2008-07-30 |    |       |
  2 | eee   |  2 | 2008-04-22 |    |       |
(17 rows)
```

Los *Joins* se van resolviendo desde primero al último. Por lo tanto, tomando el ejemplo, se resuelve el *Inner Join* de la tabla1 con la tabla2 y luego un *Left Outer Join* del set de datos resultante de la primera operación con la tabla3.

3.5.6. Union, Intersect y Except

Los comandos *union*, *intersect* y *except* permiten juntar resultados de 2 consultas separadas en un solo set. Todos estos comandos requieren que ambas consultas retornen el mismo número de columnas, considerando que los tipos correspondientes de columnas sean de tipos de datos compatibles.

```
(SELECT …) UNION (SELECT …)
```

UNION [ALL]	Une ambos sets de resultados entregando todas filas sin duplicar. No se pueden usar SENTENCIAS que tengan ORDER BY, LIMIT, FOR UPDATE o FOR SHARE. Order by y limit pueden venir en subconsultas entre paréntesis.
INTERSECT [ALL]	Entrega el set donde los resultados son los valores presentes en ambos sets de resultados. No se pueden usar SENTENCIAS que tengan ORDER BY, LIMIT, FOR UPDATE o FOR SHARE. Order by y limit pueden venir en subconsultas entre paréntesis.
EXCEPT [ALL]	Entrega las filas que se encuentran en la consulta de la izquierda, pero no en la de la derecha. No se pueden usar SENTENCIAS que tengan ORDER BY, LIMIT, FOR UPDATE o FOR SHARE. Order by y limit pueden venir en subconsultas entre paréntesis.

Todos los comandos no entregan filas duplicadas a menos que se agregue la opción ALL.

Ejemplos:

```
SELECT monto, cliente_rut, 'dolar' as tipo FROM ordenes_compra_dolares
UNION
SELECT monto, cliente_rut, 'pesos' as tipo FROM ordenes_compra_pesos
```

```
SELECT monto, cliente_rut, 'dolar' as tipo FROM ordenes_compra_dolares
UNION ALL
SELECT monto, cliente_rut, 'pesos' as tipo FROM ordenes_compra_pesos
```

```
SELECT votante_rut, votante_nombre FROM votantes_total
EXCEPT
SELECT persona_rut as votante_rut, persona_nombre as votante_nombre FROM penas_aflictivas;
```

```
SELECT cliente_rut FROM clientes_premium
INTERSECT
SELECT persona_rut as cliente_rut FROM habitantes_las_condes;
```

3.6. Secuencias

3.6.1. Generalidades

Las secuencias son objetos dentro de una base de datos cuya función es generar números. Son utilizadas principalmente para proporcionar auto-incremento a columnas llave.

Las secuencias parten por defecto desde 1, incrementándose en 1 cada vez que son consultadas. De alcanzarse el valor máximo establecido, cualquier llamada a la secuencia falla a menos que se establezca que cicla.

3.6.2. Creación (DDL)

La sintaxis de creación es la siguiente.

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
```

Ejemplos:

CREATE SEQUENCE misecuencia;
Secuencia normal partiendo de 1, incrementándose en 1.

CREATE SEQUENCE misecuencia2 INCREMENT by -1 START 10000;
Secuencia que parte en 10.000 y decrece de uno en uno.

CREATE SEQUENCE misecuencia3 MAXVALUE 100 CYCLE;
Secuencia que parte de 1, incrementa en uno, llega hasta 100 y cicla.

Formalmente Postgres general una tabla de 1 sola fila con el nombre de la secuencia donde se mantiene la información relacionada con la secuencia.

```
mibase=> select * FROM articulos_articulo_id_seq ;
       sequence_name       | last_value | increment_by |      max_value      | min_value | cache_value | log_cnt | is_cycled | is_called
---------------------------+------------+--------------+---------------------+-----------+-------------+---------+-----------+-----------
 articulos_articulo_id_seq |          1 |            1 | 9223372036854775807 |         1 |           1 |       1 | f         | f
(1 row)
```

En este caso la secuencia, articulos_articulo_id_seq, fue autogenerada para la columna articulo_id de la tabla articulos, todavía no ha sido llamada (is_called) ni ha ciclado (is_cycled), partiendo de 1 incrementando en 1.

3.6.3. Uso, modificación y eliminación

Para utilizar una secuencia se llama al método *nextval* pasando como parámetro el nombre de la secuencia.

```
=> select nextval('articulos_articulo_id_seq');
 nextval
---------
       1
(1 row)
```

Asimismo se puede consultar el valor de una secuencia sin tener que incrementar el valor de esta a través del comando *currval*.

```
=> select currval('articulos_articulo_id_seq');
 currval
---------
       1
(1 row)
```

Para modificar algún valor de la secuencia, se utiliza el comando *setval*, el cual puede recibir 2 o 3 parámetros.

```
SELECT setval('misecuencia', 31);
```

Setea el valor de *misecuencia* a 31.

```
SELECT setval('misecuencia', 1, false);
```

Setea el valor de *misecuancia* a 1 e indica que esta no ha sido llamada nunca. Esto provoca que la primera llamada a *nextval* retorne 1.

Para eliminar una secuencia se usa DROP SEQUENCE.

```
DROP SEQUENCE misecuencia;
```

3.7. Vistas

3.7.1. Generalidades

Una vista es una forma de resumir una consulta SQL en una *tabla virtual* la cual se presenta al usuario como una tabla más. La vista no es materializada físicamente, sino que la consulta con la cual fue creada es ejecutada cada vez que se consulta la vista.

Existen beneficios asociados al uso de vistas.

Seguridad: Las vistas pueden proporcionar un nivel adicional de seguridad. Cada vista al comportarse como una tabla puede tener permisos independientes de las tablas que figuran en la consulta base.

Simplicidad: Las vistas permiten ocultar la complejidad de los datos.

Exactitud en los datos solicitados: Permiten acceder a un subconjunto de datos específicos, omitiendo datos e información innecesaria e irrelevante para el usuario.

Amplía Perspectivas de la base de datos: Proporciona diversos modelos de información basados en los mismos datos, enfocándolos hacia distintos usuarios con necesidades especificas. El mostrar la información desde distintos ángulos nos ayuda a crear ambientes de trabajo y operación acordes a los objetivos de la empresa. Debe evaluarse el perfil y requerimientos de información de los usuarios destino de la vista.

3.7.2. Creación y Eliminación (DDL)

La sintaxis de creación es la siguiente.

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] VIEW name [ ( column_name [, ...] ) ]
    AS query
```

Cuando se reemplaza una vista, solo se puede hacer por una consulta que genere la misma cantidad y tipos de columnas que la consulta anterior.

Ejemplo:

```
CREATE VIEW mivista AS SELECT nombre, telefono FROM clientes WHERE edad > 18;
```

Para eliminar una vista se usa DROP VIEW.

```
DROP VIEW mivista;
```

3.8. Índices

3.8.1. Generalidades

En la práctica, una base de datos es solo tan útil como las formas en que esta provee acceso a la información almacenada. Si una consulta demora mucho tiempo en ejecutarse, entonces la usabilidad de los datos se reduce, dado que se desperdicia mucho tiempo en la espera.

Los índices son objetos que permiten ordenar los datos en una tabla, acelerando las búsquedas considerablemente. Crear índices pensando en la forma en que son consultados los datos puede reducir los tiempos de búsqueda de minutos a fracciones de segundo.

3.8.2. Creación (DDL)

La sintaxis de creación de un índice es la siguiente.

```
CREATE [ UNIQUE ] INDEX name ON table [ USING method ]
    ( { column | ( expression ) } [ opclass ] [, ...] )
    [ TABLESPACE tablespace ]
    [ WHERE predicate ]
```

Parámetros:

UNIQUE	Indica que el valor, o valores en caso de un índice por varias columnas, debe ser único en la tabla. Al modificar o insertar datos se verifica la condición de unicidad, lanzándose un error en caso de duplicados.
USING METHOD	Indica el método ton el cual realizar el índice, que por defecto es B-Tree[2].
TABLESPACE	Indica en que *tablespace* será almacenado este índice. Se recomienda que sea en el más veloz de todos.
WHERE	Condición que deben satisfacer las columnas parte del índice para pertenecer a él.

[2] B-Tree o árbol B, es una estructura de datos que permite almacenar una gran cantidad de valores, a la vez que permite recorrer tales valores en muy pocos pasos.

Ejemplos:

```
CREATE INDEX articulos_titulos_idx ON articulos (articulo_titulo);

CREATE INDEX bitacora_nombre_idx ON bitacora (empleado_rut);

CREATE INDEX calces_idx ON acciones_calces (calce_fecha, calce_folio);
```

Se debe tener en consideración que un índice agrega carga al DBMS al momento de insertar una fila en la tabla, dado que tiene que realizar las labores de creación de la entrada en el índice o su modificación, para cada inserción u modificación en los casos que corresponda. ***El índice degrada el desempeño de las inserciones en pos de acelerar las búsquedas.***

Una tabla con un alto volumen de inserciones por unidad de tiempo, puede sufrir una baja considerable en su desempeño si se tienen muchos índices creados en ella. Si no se toman las precauciones adecuadas, la cantidad de índices podría tornar la tabla lenta al punto que el delay de inserción sea mayor al tiempo de llegada de los datos, caso en el cual el sistema colapsa dado que la cola de filas a insertar aumentará constantemente hasta que el volumen de datos decrezca.

Para eliminar un índice se utiliza el comando DROP INDEX.

```
DROP INDEX nombre;
```

3.9. Transacciones y locks

3.9.1. Generalidades

Uno de los puntos más importantes en un motor de base de datos es proporcionar los mecanismos adecuados para manejar la concurrencia, es decir, como asegurar la integridad de los datos y la correctitud de las operaciones realizadas cuando existen múltiples personas intentando de acceder a la misma información.

Para ejemplificar el concepto, usemos un ejemplo cotidiano. Los cajeros automáticos (Redbanc / ATM) realizan descuentos del saldo de la cuenta corriente cuando se obtiene dinero de ellos. Supongamos que en un momento determinado existe un usuario sacando dinero y simultáneamente se esta realizando un cobro electrónico sobre la misma cuenta.

Cajero Automático		***Cobro Electrónico***	
Saldo: 100.000	Cobro: 17.000	Saldo:100.000	Cobro: 30.000
Saldo: 83.000		Saldo: 70.000	

El mecanismo seria: Cuanto dinero hay en la cuenta? Del saldo descuente esta cantidad de dinero. Si esta operación puede ser realizada EXACTAMENTE al mismo tiempo, entonces se obtendrá el resultado de la tabla anterior. Así es necesario proveer mecanismos de bloqueo temporal para asegurar que solo una instancia tenga acceso a los datos *concurrentemente* (simultáneamente).

3.9.2. Anatomía de una Transacción

Una transacción es una operación que se realiza por completo, de forma atómica[3]. En caso de fallar algo, los cambios se revierten y todo vuelve al estado anterior. Postgres sigue el estándar SQL para la sintaxis de transacciones.

```
BEGIN WORK;
INSERT INTO usuarios VALUES (‘acampos’, ‘Andres Campos’);
INSERT INTO usuario_saldo VALUES (‘acampos’, 10000);
COMMIT;
```

Cada transacción comienza con el comando BEGIN WORK, aunque WORK es opcional. Al terminar el trabajo se debe invocar al comando COMMIT para indicar que la transacción ha concluido.

En cualquier punto de la transacción se puede invocar a ROLLBACK, comando que deshace todos los cambios realizados en la base de datos, dejándola en el estado previo al inicio de la transacción.

```
BEGIN WORK;
INSERT INTO usuarios VALUES (‘acampos’, ‘Andres Campos’);
ROLLBACK;
INSERT INTO usuarios VALUES (‘acampos’, ‘Andres Campos’);
INSERT INTO usuario_saldo VALUES (‘acampos’, 10000);
COMMIT;
```

En el caso de que un comando SQL falle, Postgres ignorará las siguientes sentencias hasta el fin de la transacción, la cual se indica con COMMIT o ROLLBACK.

Las transacciones en Postgres pueden ser grabadas en un punto intermedio, de manera de poder re-ejecutar o deshacer solo una porción de la transacción. Para esto se utiliza SAVEPOINT.

```
BEGIN WORK;
INSERT INTO usuarios VALUES (‘acampos’, ‘Andres Campos’);
SAVEPOINT s1;
INSERT INTO usuario_saldo VALUES (‘acampos’, 10000);
ROLLBACK s1;
INSERT INTO usuario_saldo VALUES (‘acampos’, 10000);
COMMIT;
```

[3] El concepto de atómico o atomicidad, hace referencia a una operación que es *indivisible*, y por tanto se realiza por completo o no se realiza en absoluto.

3.9.3. Locks

Hasta ahora hemos visto como una operación de múltiples comandos se realiza como una sola, pero no hemos provisto mecanismos para realmente proteger los datos en casos como el expuesto al inicio de este punto.

Para realmente proveer protección sobre los datos, estos se deben *bloquear* para que otros no los utilicen mientras se esta en la *región crítica*[4].

```
LOCK [ TABLE ] name [, ...] [ IN lockmode MODE ] [ NOWAIT ]
```

Donde *lockmode* es una de las siguientes opciones.

```
    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
    | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

Las más importantes son 2: SHARE y EXCLUSIVE. La primera provee bloqueo sobre modificaciones solicitadas por otras instancias, mientras que permite que la tabla sea consultada. La segunda impide además que la tabla sea consultada.

Cuando se especifica el modificador NOWAIT, entonces el intento de bloqueo de una tabla no espera en caso de existir un LOCK anterior. En este caso el comando lanza un error y la transacción es abortada.

Ejemplo:

```
BEGIN;
LOCK TABLE saldo IN SHARE MODE;
UPDATE saldo SET saldo = saldo – 10000 WHERE cliente = ‘2985673’;
COMMIT;
```

Existe además la posibilidad de realizar bloqueos más detalladamente a nivel de filas a través de **SELECT FOR UPDATE**. Esto permite que otras transacciones puedas bloquear otras parte de la tabla y realizar operaciones que no interfieren con la actual de forma concurrente.

```
BEGIN;
SELECT * FROM saldo WHERE cliente = ‘2985673’ FOR UPDATE;
UPDATE saldo SET saldo = saldo – 10000 WHERE cliente = ‘2985673’;
COMMIT;
```

[4] Se define como región crítica aquella porción del código que de ejecutarse simultáneamente por más de una instancia, los resultados de la computación son inciertos.

3.10. Query Planner

3.10.1. Generalidades

Todos los DBMS poseen una pieza de software que planifica la estrategia de recolección de datos en una consulta, de manera de optimizar de la mejor forma posible los tiempos de respuesta.

El *Planificador de Consultas* o *Query Planner* es la pieza en Postgres que realiza tal labor. Su comportamiento puede ser modificado vía el archivo de configuración postgres.conf como se describe en la sección 2.2.5. Este se vale de las estadísticas recolectadas por el proceso para este efecto, para poder determinar cual es la mejor estrategia a seguir en cada caso.

Postgres provee 2 comandos para desplegar el plan de consulta y observar los tiempos que se están utilizando en cada uno de los pasos.

3.10.2. EXPLAIN y EXPLAIN ANALYZE

El comando EXPLAIN permite desplegar el plan de consulta utilizado en una sentencia SQL. Se utiliza anteponiendolo a la consulta que se desea explicar. Los resultados son expresados en cantidad de páginas de disco que se requieren recuperar.

Al utilizar EXPLAIN, la consulta en si no es realmente realizada, sino solo se despliega el plan que se utilizaría en caso de ejecutarla.

Usando EXPLAIN ANALYZE, Postgres ejecuta realmente la consulta entregando los resultados en tiempo verdadero (milisegundos). Aquí se debe tener la precaución que si se están realizando operaciones que modifican la base de datos, como INSERT, UPDATE, DELETE; y se requiere resguardar la información original, la sentencia debería estar contenida en una transacción para realizar ROLLBACK al final.

Ejemplo Explain:

```
mibase=> explain select * FROM ordenes, usuarios WHERE ordenes.ordenes_usuario_creador = usuarios.usuario_id;
                                QUERY PLAN
------------------------------------------------------------------------------
 Hash Join  (cost=1.46..18.41 rows=398 width=223)
   Hash Cond: ("outer".ordenes_usuario_creador = "inner".usuario_id)
   ->  Seq Scan on ordenes  (cost=0.00..10.98 rows=398 width=139)
   ->  Hash  (cost=1.37..1.37 rows=37 width=84)
         ->  Seq Scan on usuarios  (cost=0.00..1.37 rows=37 width=84)

(5 rows)
```

En este caso el plan es:

- Realizar una búsqueda secuencial en la tabla usuarios
- Aplicar hash a los valores
- Realizar una búsqueda secuencial en la tabla ordenes
- Aplicas hash a los valores
- Unir ambas tablas utilizando los valores hash obtenidos.

Explain indica en cada fila el costo estimado en base a la cantidad de filas de una tabla, valor obtenido de las estadísticas de Postgres. Un comando explain puede dar resultados muy diferentes en base a la cantidad de filas de una tabla, la antigüedad de las estadísticas y la versión de Postgres que se esté utilizando.

En cada fila se muestra el costo que se lleva acumulado hasta ese comando y el costo al terminar, estimando la cantidad de filas de la tabla y el ancho estimado (en bytes) de cada fila recuperada.

Ejemplo Explain Analyze:

```
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=5.72..5.72 rows=1 width=22) (actual time=68.802..68.930 rows=25 loops=1)
   Sort Key: ods.ods_codigo_empresa
   ->  Nested Loop  (cost=0.02..5.71 rows=1 width=22) (actual time=7.136..68.574 rows=25 loops=1)
         ->  HashAggregate  (cost=0.02..0.03 rows=1 width=4) (actual time=0.914..1.047 rows=25 loops=1)
               ->  Result  (cost=0.00..0.01 rows=1 width=0) (actual time=0.688..0.806 rows=25 loops=1)
         ->  Index Scan using ods_pkey on ods  (cost=0.00..5.61 rows=1 width=22) (actual time=0.009..0.015 rows=1 loops=25)
               Index Cond: (ods.ods_id = "outer".getempleadoods)
 Total runtime: 69.142 ms
```

- Búsqueda a través del índice “ods_pkey” en la tabla ods, tiempo 0.015 ms
- Agrupación de resultados y aplicación de hash.
- Recorrido de set obtenido
- Ordenado a través del campo ods_codigo_empresa

A través del análisis de estos resultados, se puede establecer claramente si hay un cuello de botella en algún punto de la consulta, y rescribirla o agregar índices para acelerar los tiempos de respuesta.

4. Procedimientos Almacenados

4.1. Generalidades

Los procedimientos almacenados son funciones que se crean dentro de una base de datos. Estas, como su nombre lo indica, están almacenadas dentro del modelo de datos haciendo que su ejecución sea lo más ágil posible para el manejo de la información.

Los procedimientos pueden estar escritos en varios lenguajes. PostgreSQL soporta SQL, C, PL/pgSQL, PL/PerlSQL, etc; permitiéndose la sobrecarga.

La sintaxis de creación de un procedimiento almacenado para cualquier lenguaje es la siguiente.

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
    [ RETURNS rettype ]
  { LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
    [ WITH ( attribute [, ...] ) ]
```

Los parámetros.

name	Nombre de la función
argmode	Modo del argumento (IN , OUT), por defecto IN. Los parámetros definidos con OUT son realmente retorno en lugar de entradas a la función.
argname	Nombre que tendrá la variable dentro de la función. (Solo disponible en PL/pgsql)
rettype	Tipo de dato de retorno de la función. Si se omite se asume VOID.
langname	Lenguaje en el cual esta escrita la función.

Al utilizar el modificador OR REPLACE, solo se puede modificar una función que tenga exactamente los mismos parámetros y tipo de retorno. Si no, entonces se estaría creando una nueva función.

Los parámetros INMUTABLE, STABLE, VOLATIL, sirven para indicar al sistema si es seguro reemplazar múltiples evaluaciones de la función con una sola para optimización de la ejecución. En detalle:

INMUTABLE indica que la función siempre retorna el mismo valor cuando se le entregan los mismos argumentos, es decir, no realiza ninguna consulta a otras tablas o depende de variables como la hora. Cuando se entrega este argumento, en donde se llame a la función se reemplaza directamente su valor si se ha calculado.

STABLE indica que dentro de un recorrido por una tabla, el valor de la función no cambia – no cambia fila a fila. Tipo recomendado para funciones que dependen del valor de parámetros, consultas a tablas, timezone, etc;

VOLATILE se utiliza para aquellas funciones que cambian en cada llamado, como *nextval* de las secuencias. Esta opción es aquella que se asume por defecto.

La opción CALLED ON NULL INPUT – seleccionada por defecto – permite que esta sea llamada con algunos o todos sus parámetros en NULL. RETURNS NULL ON NULL INPUT dice que se entregue NULL como retorno si alguno de los parámetros es NULL.

Con la opción SECURITY DEFINER se especifica que la función se ejecutará con los permisos del creador, en lugar de los del invocador como es por defecto (SECURITY INVOKER).

Dentro de una función los parámetros son accedidos a través de posición en la definición anteponiendo $.

Ejemplo:

```
CREATE FUNCTION ejemplo(varchar, int) RETURNS boolean AS ‘
       SELECT $1 = ‘ejemplo && $2 < 0:
‘ LANGUAGE sql;
```

En este ejemplo el retorno es booleano y se construye de la verificación de los valores de los parámetros.

4.2. Procedimientos Almacenados en SQL

En PostgreSQL existe la posibilidad de crear procedimientos almacenados en lenguaje SQL. Esta característica permite la reutilización de código y ordenamiento del mismo, pudiendo concentrar parte de la lógica en un solo lugar desde donde se le puede cambiar transparentemente. Dentro el código debe ser construido

Ejemplo:

```
CREATE FUNCTION getPrecio(INT) RETURNS NUMERIC AS ‘
       SELECT precio FROM bitacora_precio WHERE producto_id = $1 ORDER BY timestamp DESC LIMIT 1;
‘ LANGUAGE SQL;
```

4.3. Procedimientos Almacenados en Plpgsql

4.3.1. Introducción

PL/pgSQL es un lenguaje procedural cargable al DBMS PostgreSQL, es decir, no viene incorporado dentro del *core* del sistema, sino que se carga cuando se requiere.

Del manual de Postgres, se conocen los objetivos al momento de diseñar PL/pgSQL, entre los cuales estan:

- Poder ser utilizado para la creación de funciones y triggers.
- Añadir estructuras de control al lenguaje SQL
- Realizar computaciones complejas
- Heredar todos los tipos, funciones y operadores definidos por el usuario

4.3.2. Sintaxis

4.3.2.1. Anatomía de un programa PL/PgSQL

Es un lenguaje orientado a bloques. Un bloque se define como:

```
[ <<etiqueta>> ]
[ DECLARE
  declaraciones ]
BEGIN
  estamentos
END;
```

Puede haber múltiples subbloques en la sección de sentencia de un bloque. Los subbloques pueden usarse para ocultar variables a otros bloques de sentencias.

Las variables declaradas en la sección de declaraciones que preceden a un bloque, son inicializadas a sus valores por defecto cada vez que el bloque es introducido, no sólo una vez por cada llamada a la función.

```
CREATE FUNCTION funcion() RETURNS INTEGER AS '
DECLARE
  cantidad INTEGER;
BEGIN
  -- LOGICA DEL PROGRAMA AQUI
  RETURN 1;
END;
' LANGUAGE 'plpgsql';
```

Un ejemplo con subbloques:

```
CREATE FUNCTION funcion() RETURNS INTEGER AS '
DECLARE
  cantidad INTEGER;
BEGIN
  -- LOGICA DEL PROGRAMA AQUI

  DECLARE
    cantidad;
  BEGIN
    -- LOGICA DEL SUBBLOQUE
    RETURN 0;
  END;

END;
' LANGUAGE 'plpgsql';
```

Es importante no confundir el uso de BEGIN/END para la agrupación de estamentos en PL/pgSQL con los comandos de base de datos para el control de transacciones. Los comandos BEGIN/END de PL/pgSQL son sólo para agrupar; ellos no inician o finalizan una transacción. Los procedimientos de funciones y triggers son siempre ejecutados dentro de una transacción establecida por una consulta externa -no pueden iniciar o validar transacciones, ya que PostgreSQL no tiene transacciones anidadas.

4.3.2.2. Declaración y asignación

Todas las variables, filas y registros usados en un bloque deben estar declaradas en la sección de declaraciones del bloque.

Las variables PL/pgSQL pueden ser de cualquier tipo de datos SQL. Aquí se tienen algunos ejemplos de declaración de variables:

```
user_id INTEGER;
cantidad NUMERIC(5);
url VARCHAR;
myrow nombretabla%ROWTYPE;
myfield nombretabla.nombrecampo%TYPE;
unafila RECORD;
```

Los primeros 3 son tipos de datos nativos de PostgreSQL. El cuarto tipo indica que *myrow* tendrá la misma configuración que una fila de la tabla *nombretabla*; análogamente *myfield*hereda el tipo de datos del cambpo *nombrecampo* de *nombretabl*a. El último tipo RECORD indica que cualquier tipo de fila – múltiples campos – puede ser colocado en la variable.

La sintaxis general de una declaración de variables es:

```
nombre [ CONSTANT ] tipo [ NOT NULL ] [ { DEFAULT | := } expresión ];
```

La cláusula DEFAULT, si existe, especifica el valor inicial asignado a la variable cuando el bloque es introducido. Sin la cláusula DEFAULT no es facilitada entonces la variable es inicializada al valor SQL NULL.

La opción CONSTANT previene la asignación de otros valores a la variable, así que su valor permanece durante la duración del bloque.

Si se especifica NOT NULL, una asignación de un valor NULL resulta en un error en tiempo de ejecución. Todas las variables declaradas como NOT NULL deben tener un valor por defecto no nulo especificado.

El valor por defecto es evaluado cada vez que el bloque es introducido. Así, por ejemplo, la asignación de now a una variable de tipo timestamp provoca que la variable tenga la fecha y hora de la llamada a la actual función, y no la fecha y hora de su compilación.

Ejemplo:

```
cantidad INTEGER DEFAULT 32;
url varchar := ''http://www.jm.cl'';
user_id CONSTANT INTEGER := 10;
```

Para asignar un valor a una variable se utiliza := como operador de asignación.

```
variable := valor;
```

Ejemplo:

```
CREATE FUNCTION funcion() RETURNS INTEGER AS '
DECLARE
  cantidad INTEGER := 15;
BEGIN
  cantidad := 30;
  RAISE NOTICE ''La cantidad aquí es %'',cantidad; -- La cantidad aquí es 30
  RETURN cantidad;
END;
' LANGUAGE 'plpgsql';
```

Cuando deseamos realizar una asignación de un valor obtenido de una consulta, se utiliza la sentencia SELECT INTO. Ejemplo a continuación

```
CREATE FUNCTION funcion() RETURNS INTEGER AS '
DECLARE
  cantidad INTEGER := 15;
BEGIN
  SELECT cantidad_total INTO cantidad FROM ventas WHERE folio = 12;
  RETURN cantidad;
END;
' LANGUAGE 'plpgsql';
```

4.3.2.3. Parámetros

Los parámetros pasados son nominados con los identificadores $1, $2, etc. Opcionalmente, se pueden declara alias para los nombres de los parámetros, con el objeto de incrementar la legibilidad del código. Tanto el alias como el identificador numérico pueden ser utilizados para referirse al valor del parámetro. Algunos ejemplos:

```
CREATE FUNCTION tasa_ventas(REAL) RETURNS REAL AS '
DECLARE
  subtotal ALIAS FOR $1;
BEGIN
  return subtotal * 0.06;
END;
' LANGUAGE 'plpgsql';
```

```
CREATE FUNCTION usa_muchos_campos(tablename) RETURNS TEXT AS '
DECLARE
  in_t ALIAS FOR $1;
BEGIN
  RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
' LANGUAGE 'plpgsql';
```

4.3.2.4. Mensajes y Errores (Notice, Exceptions)

Dentro del lenguaje SQL de Postgres existen varias formas de generar mensajes que queden en el log del sistema, se muestren vía consola o incluso sean propagados a la siguiente capa, como es el caso de los errores.

Por una parte tenemos *NOTICE* que despliega un mensaje cuando se esta ejecutando vía consola, quedando tal mensaje en el archivo log, al menos cuando están las opciones por defecto de postgres.conf.

```
RAISE NOTICE ‘’El usuario % tiene un saldo de $%””, usuario.nombre, saldo.cantidad;
```

Aquí vemos el uso de variables dentro de la expresión. Utilizando el comodín % se puede reemplazar un valor dentro del string, obtenido de las variables a continuación en el mismo orden correspondiente.

En el caso de querer lanzar un error que se propague a la siguiente capa, como por ejemplo un programa PHP, se utiliza EXCEPTION. Estos errores son análogos a los generados nativamente por PostgreSQL en todo sentido, cortando la ejecución de una transacción inclusive.

```
RAISE EXCEPTION ‘’El ciudadano RUT % ya tiene un voto registrado en el sistema’’,
votante.votante_rut;
```

4.3.2.5. Retorno

```
RETURN expression;
```

RETURN con una expresión es usado para retornar desde una función PL/pgSQL. La función termina y el valor de la expresión es retornado.

Para retornar un valor compuesto (una fila), se debe escribir una variable RECORD en la cual podrá almacenar la tupla o registro que desea retornar como expresión. Cuando retorne un tipo entero, cualquier expresión pude ser usada. El resultado de la expresión será automáticamente convertido al tipo de retorno de la función (si ha declarado la función para que retorne void (nulo), entonces la expresión puede ser omitida, y se ignorará en cualquier caso).

El valor de retorno de una función no puede quedar sin definir. Si el control llega al final del bloque de mayor nivel de la función sin detectar un estamento RETURN, ocurrirá un error en tiempo de ejecución.

Cuando una función PL/pgSQL es declarada para que retorne un SETOF de algún tipo, el procedimiento a seguir es algo diferente. En ese caso, los elementos individuales a retornar son especificados en comandos RETURN NEXT, y luego un comando final RETURN sin argumentos que es usado para indicar que la función ha terminado su ejecución. RETURN NEXT puede ser usado tanto con tipos de datos enteros como compuestos; en el último caso, una ``tabla'' entera de resultados. Ejemplo a continuación

```
CREATE FUNCTION mi_funcion() RETURNS SET OF INT AS ‘

	DECLARE
		aux INT;
	BEGIN
		FOR aux IN SELECT usuario_id FROM usuarios LOOP
			RETURN NEXT aux;
		END LOOP;

		RETURN;
	END;
‘ LANGUAGE plpgsql;
```

Las funciones que usen RETURN NEXT deberían ser llamadas de la siguiente forma:

```
SELECT * FROM mi_funcion();
```

Esto es porque al usar RETURN NEXT estamos devolviendo desde la función un registro completo (una fila), por ende es como si fuese una tabla que contiene un solo registro (en algunos casos pueden ser muchos más). Podríamos realizar también consultas anidadas con funciones que retornen los datos de esta manera.

4.3.3. Elementos de Control

4.3.3.1. Generalidades

Se denominan *elementos* o *estructuras de control*, aquellas instrucciones que permiten controlar la linea de ejecución de un programa, es decir, si se efectúan unas u otras instrucciones.

4.3.3.2. IF THEN

```
IF boolean-expression THEN
  estamentos
END IF;
```

Los estamentos IF-THEN son el formato más simple del IF. Los estamentos entre THEN y END IF serán ejecutados si la condición se cumple. En caso contrario, serán ignorados.

```
IF var_usuario_id <> 0 THEN
  UPDATE usuarios SET nombre = 'Nuevo Nombre' WHERE usuario_id = var_usuario_id;
END IF;
```

```
IF NOT FOUND THEN
  return NULL;
END IF;
```

En este último ejemplo se muestra el uso de **NOT FOUND**, el cual hace referencia a los resultados de la última sentencia SELECT dentro de la funcion.

4.3.3.3. IF THEN ELSE

```
IF boolean-expression THEN
  estamentos
ELSE
  estamentos
END IF;
```

Los estamentos IF-THEN-ELSE añadidos al IF-THEN permiten especificar qué hacer si la condición del IF se evalúa como falsa.

```
IF nombre_completo IS NULL OR nombre_completo = '''' THEN
  return nombre_completo;
ELSE
  return rut;
END IF;
```

```
IF dinero > 0 THEN
  INSERT INTO personas (dinero) VALUES(dinero);
  RETURN ''t'';
ELSE
  RETURN ''f'';
END IF;
```

Los estamentos IF pueden anidarse, tal como en el siguiente ejemplo:

```
IF usuario.sexo = ''m'' THEN
   usuario.sexo := ''masculino'';
ELSE
  IF usuario.sexo = ''f'' THEN
     usuario_sexo := ''femenino'';
  END IF;
END IF;
```

Cuando use este formato, estará anidando un operador IF dentro de la parte ELSE de un operador IF superior. Necesitará un cerrar dicho IF con END IF por cada IF anidado, y uno para el IF-ELSE padre. Esto puede ser un poco tedioso, pero es parte de la sintaxis, además ayuda a mantener el orden de las operaciones que estamos realizando.

4.3.3.4. IF THEN ELSEIF ELSE

```
IF boolean-expression THEN
  estamentos
[ ELSIF boolean-expression THEN
  estamentos
[ ELSIF boolean-expression THEN
  estamentos ...]]
[ ELSE
  estamentos ]
END IF;
```

IF-THEN-ELSEIF-ELSE proporciona un método más conveniente para chequear muchas alternativas en un estamento. Formalmente es equivalente a comandos IF-THEN-ELSE-IF-THEN anidados, pero sólo se necesita un END IF.

Aquí hay un ejemplo de su uso:

```
IF numero = 0 THEN
  resultado := ''cero'';
ELSIF numero > 0 THEN
  resultado := ''positivo'';
ELSIF numero < 0 THEN
  result := ''negativo'';
ELSE
  -- la otra posibilidad es que sea nulo
  resultado := ''NUL0'';
END IF;
```

La sentencia ELSE final es opcional.

El beneficio de la instrucción ELSEIF es que nos permite realizar una comparación si es que la instrucción anterior resulta falsa así menos estamentos, esto nos ayuda a mantener un código más legible mejorando el mantenimiento de este.

4.3.4. Bucles / ciclos

4.3.4.1. Generalidades

Los *bucles* o *ciclos* permiten realizar una tarea múltiples veces en un código fuente. A continuación se revisan los 3 tipos básicos de ciclos – loop, while, for – y a continuación la forma de ciclar a través de resultados de búsqueda.

4.3.4.2. LOOP

```
LOOP
[<<etiquetal>>]
LOOP
  estamentos
END LOOP;
```

LOOP define un ciclo incondicional, esto quiere decir que es repetido indefinidamente hasta que sea terminado por un estamento EXIT o RETURN. La etiqueta opcional puede ser usada por estamentos EXIT en bucles anidados para especificar qué nivel de anidación debería ser terminado.

www.jm.cl // jm@jm.cl
Moneda 973, Of 536 – Fono-fax: 638 6201

EXIT

```
EXIT [ etiqueta ] [ WHEN expresión ];
```

Si no se proporciona etiqueta, el ciclo más al interior es terminado y el estamento más cercano a END LOOP es ejecutado a continuación. Si se proporciona etiqueta, esta debe ser la etiqueta del bucle o bloque más actual o de nivel externo. Entonces el ciclo o bloque nombrado es terminado, y el control continúa con el estamento que sigue al correspondiente END del ciclo/bloque.

Si WHEN está presente, sólo ocurrirá la salida del ciclo si la condición especificada es cierta, de lo contrario pasa al estamento tras EXIT.

Ejemplos:

```
LOOP
  IF contador > 0 THEN
     EXIT; -- exit loop
  END IF;
END LOOP;
```

```
BEGIN
  IF stocks > 100000 THEN
     EXIT;
  END IF;
END;
```

Nótese que en el primer ejemplo, usamos *EXIT* para terminar un ciclo pero dentro de un estamento LOOP. En el segundo ejemplo lo hacemos dentro de un IF lo cual provocara error ya que no se puede realizar dicha acción fuera de un ciclo.

4.3.4.3. WHILE

```
WHILE
[<<etiqueta>>]
WHILE expresión LOOP
  estamentos
END LOOP;
```

El estamento WHILE repite una secuencia de instrucciones mientras que la condición sea verdadera. La condición es chequeada justo antes de cada entrada al cuerpo del ciclo.

Ejemplo:

```
WHILE dinero_bolsillo > 0 AND dinero_billetera > 0 LOOP
  -- realizamos algunas acciones aqui
END LOOP;
```

En este ejemplo, se repetirá el ciclo WHILE hasta que una de las 2 condiciones sean menor o igual a cero.

4.3.4.4. FOR

```
[<<etiqueta>>]
FOR nombre IN [ REVERSE ] expresión .. expresión LOOP
  estamentos
END LOOP;
```

Este formato de FOR crea un ciclo que itera sobre un rango de valores enteros. La variable nombre es automáticamente definida como de tipo integer y existe sólo dentro del ciclo. Las dos expresiones dando el menor y mayor valor del rango son evaluadas una vez se entra en el ciclo. El intervalo de iteración es de 1 normalmente, pero es -1 cuando se especifica REVERSE.

Ejemplos:

```
FOR i IN 1..10 LOOP
  -- algunas expresiones aquí

  RAISE NOTICE ''i is %'',i;
END LOOP;
```

```
FOR i IN REVERSE 10..1 LOOP
  -- algunas expresiones aquí
END LOOP;
```

Cuando se coloca instrucción REVERSE el ciclo itera de atrás hacia adelante, si empezamos en 10, ira descontando 1 en 1 hasta llegar a 1.

Nota: RAISE NOTICE se utiliza para enviar mensajes a la pantalla.

4.3.4.5. Bucles a través de los resultados de una búsqueda

Usando un tipo diferente de ciclo FOR, se puede iterar a través de los resultados de una consulta y manipular sus datos. La sintaxis es:

```
[<<etiqueta>>]
FOR registro | fila IN select_query LOOP
  estamentos
END LOOP;
```

En la variable registro o fila le son sucesivamente asignadas todas las filas resultantes de la consulta SELECT. Aquí un ejemplo:

```
CREATE FUNCTION refrescar_usuarios () RETURNS INTEGER AS '

DECLARE
  uvista RECORD;
BEGIN

  RAISE NOTICE  ''Refrescando la Usuarios :D ...'' ;

  FOR uvista IN SELECT * FROM vista_usuarios ORDER BY usuario_rut LOOP

     -- Ahora   "uvista" tiene un registro de la vista  vista_usuarios
     RAISE NOTICE ''Refrescando la fecha de registro del usuario '' || uvista.usuario_nombre;
     UPDATE vista_usuarios SET fecha_ingreso = now() WHERE usuario = uvista.usuario;

  END LOOP;

  RAISE NOTICE  ''Vista Actualizada.'';
  RETURN 1;
END;
' LANGUAGE 'plpgsql';
```

En este caso se muestra el uso de un *registro* (RECORD) para almacenar la fila actual de resultado. Las variables RECORD pueden contener una fila de cualquier formato. Alternativamente también se puede usar una variable cuyo tipo de dato sea el de una tabla, restringiéndose el formato a aquel de la tabla en cuestión.

```
DECLARE
 uvista vista_usuarios;
BEGIN
```

Si el ciclo es terminado por un estamento EXIT, el valor de la última fila asignada es todavía accesible tras la salida del bucle.

El estamento FOR-IN-EXECUTE es otra forma de iterar sobre registros:

```
[<<etiqueta>>]
FOR registro | fila IN EXECUTE expresión_texto LOOP
  estamentos
END LOOP;
```

Esto es como el anterior formato, excepto porque el estamento origen SELECT es especificado como una expresión de texto, la cual es evaluada y re planificada en cada entrada al bucle FOR. Esto permite al programador seleccionar la velocidad de una consulta pre planificada o la flexibilidad de una consulta dinámica, tal como con un estamento plano EXECUTE.

Ejemplo:

```
PREPARE funcionpreparada(tabla, valor) AS
       SELECT * FROM $1 WHERE id = $2;

FOR registro IN EXECUTE funcionpreparada(''usuarios_registrados'', 3985) LOOP
       …
END LOOP
```

4.4. Triggers

4.4.1. Generalidades

Los *triggers* son procedimiento almacenados los cuales se ejecutan de forma automática ante una acción – INSERT, UPDATE, DELETE – sobre una tabla en la base de datos.

La utilización de triggers permite dar vida al modelo de una base de datos, dando la posibilidad de actualizar varias tablas con una sola inserción como por ejemplo propagación de consistencia, creación de resúmenes, checkeos complejos, etc.

Esta libertad y apertura de posibilidades también conlleva problemas. Si no se es cuidadoso con la cantidad de triggers que se utiliza, se puede llegar a un punto en donde cada operación sobre las tablas involucradas desencadene tal numero de funciones que el desempeño del sistema se vea degradado considerablemente. Más aun, si no se documento apropiadamente los triggers que se agregan al modelo, puede llegar a ser una tarea no menor averiguar la traza completa de acciones gatilladas sobre una acción en una tabla.

4.4.2. Creación y Eliminación

Un trigger se crea a partir de una función – procedimiento almacenados – que ha sido declarado con algunas características en particular:

- Se declara sin argumento, estos son manejados de otra forma dentro del procedimiento.
- Se declara con tipo de retorno *trigger*.

Ejemplo:

```
CREATE FUNCTION check_saldo() RETURNS trigger AS ‘
	DECLARE
		saldo_disponible NUMERIC;
	BEGIN
		SELECT cliente.saldo INTO saldo_disponible
		FROMcuentas_corriente WHERE cta_numero = NEW.cta_numero;

		IF NOT FOUND
			RAISE EXCEPTION ‘’Cuenta corriente % no existe’’, NEW.cta_numero;
		ELSE
			IF saldo_disponible > NEW.valor_cargo THEN
				RETURN NEW;
			ELSE
				RAISE EXCEPTION ‘’Saldo en cuenta % es insuficiente’’, NEW.cta_numero;
			END;
		END;
	END;
‘ LANGUAGE plpgsql;
```

En este caso se tiene una función que verificará que la cuenta indicada en la nueva fila (variable NEW) exista y que tenga saldo antes de permitir la inserción.

Para asociar esta función a una se utiliza la siguiente sintaxis

```
CREATE TRIGGER nombre { BEFORE | AFTER } { INSERT | UPDATE | DELETE [ OR …] }
ON tabla FOR EACH { ROW | STATEMENT }
EXECUTE PROCEDURE nombrefuncion;
```

Los parámetros BEFORE y AFTER permiten indicar si el trigger es ejecutado antes o después de que la operación se aplique a la tabla. Al usar BEFORE se pueden modificar los valores de la nueva fila en caso de inserción y modificación. En ambos casos un trigger que lance un error cancelará la operación que lo disparó.

A continuación se indica sobre que operación sobre la tabla se lanzará el trigger: INSERT, UPDATE, DELETE. Se pueden especificar múltiples acciones separadas por OR.

El parámetro ROW o STATEMENT permite indicar si la función se ejecutará para cada fila afectada en la operación o solo al final de la ejecución. Si por ejemplo un operación de UPDATE cambia 10 filas, al seleccionar EACH ROW, el procedimiento se ejecutará 10 veces (una por cada fila), mientras que con EACH STATEMENT lo hará al final.

En nuestro ejemplo:

```
CREATE TRIGGER check_saldo BEFORE INSERT ON giros
FOR EACH ROW EXECUTE PROCEDURE check_saldo();
```

Esto provocara que al momento de insertar una fila en la tabla *giros* se ejecute la función *check_saldo* ANTES de que se realice la inserción propiamente tal.

4.4.3. Variables especiales

Cuando una función plpgsql es un trigger, se tienen una serie de variables especiales disponibles dentro del procedimiento las cuales permiten conocer el estado de la operación que ejecuta el trigger.

NEW	Variable de tipo RECORD que contiene la nueva fila en caso de INSERT o UPDATE. Es null en caso de EACH STATEMENT.
OLD	Variable de tipo RECORD que contiene la antigua fila en caso de UPDATE o DELETE. Es null en caso de EACH STATEMENT.
TG_NAME	Nombre del trigger que se esta ejecutando
TG_WHEN	Contiene el *cuando* se esta ejecutando (BEFORE / AFTER)
TG_LEVEL	Indica si el procedimiento se ejecuta fila a fila o por statement.
TG_OP	Operación sobre la cual se gatilló el evento (INSERT, UPDATE, DELETE)
TG_RELID	ID de la relación sobre la cual se disparó el trigger
TG_RELNAME	Nombre de la relación sobre la cual se disparó el trigger
TG_NARGS	Número de argumentos pasados como parámetros en CREATE TRIGGER.
TG_ARGV[]	Arreglo con los argumentos pasados en CREATE TRIGGER.