Data Base Administrator PostgreSQL

Ernesto Quiñones A.
ernesto@eqsoft.net

Sesión 10

Configuraciones Avanzadas

Manipulación de Grandes Volúmenes de Datos

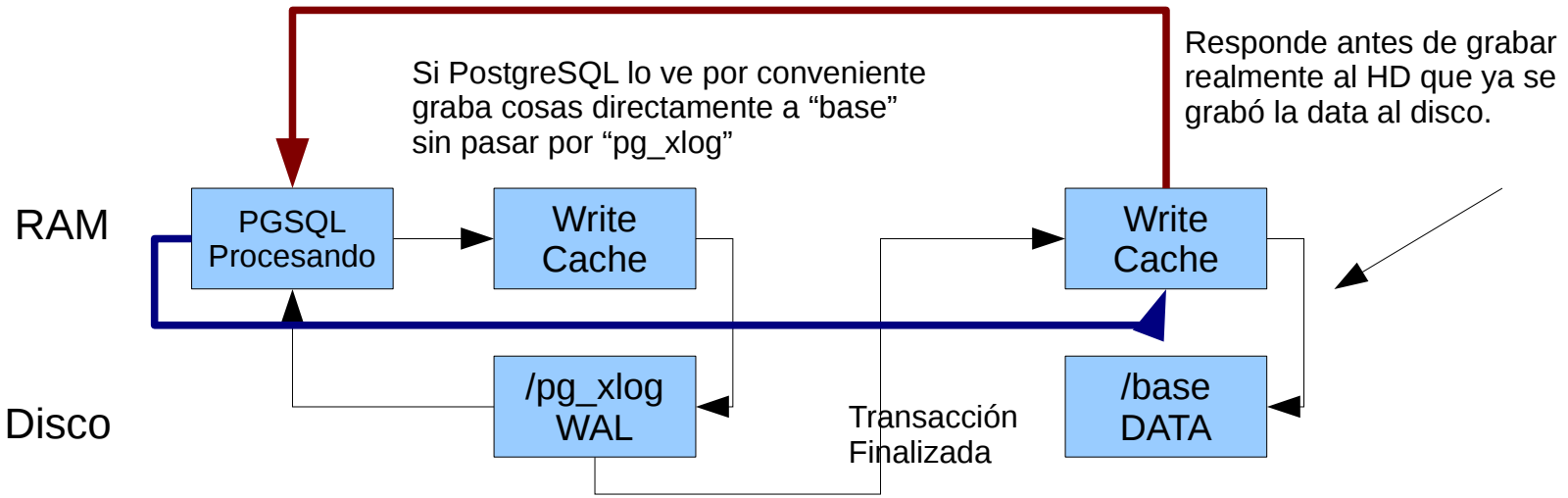
Tunning

a)Desactivar el Write-Back Cache

Toda la data siempre se trabaja entre la RAM y el Procesador antes de bajarse a disco y liberarlos para recibir y procesar otros datos.

Como RAM y Procesador son extremadamente mucho más rápidos que los discos, el sistema operativo almacena los datos en una espacio RAM del propio disco para luego grabarlos realmente a los platos del disco, en situaciones extremas el disco demorará varios minutos antes de completar la grabación.

PostgreSQL incorpora WAL como mecanismo de protección contra corrupciones a los datos antes de grabase en los repositorios de data final, en la practica sucede esto:



Tunning

a)Desactivar el Write-Back Cache

Para evitar la posibilidad de perder datos es necesario desactivar el Write Cache de los discos duros.

Para ver si tenemos el Write Cache activo hacemos esto:

```
[root@host]# hdparm -I /dev/sda | egrep "Write cache" <-- /dev/sda es la unidad
                                                         HD
          *    Write cache                            <-- si está activo aparece
                                                         con “*”

[root@host]# hdparm -W 0 /dev/sda                     <-- desactiva el write
                                                         Cache
/dev/sda:
 setting drive write-caching to 0 (off)
 write-caching =  1 (on)                              <-- problema en CentOS, no
                                                         se puede desactivar el
                                                         Write Cache
```

En RAIDs ó SANs los Write Cache pueden ser de 128MB hasta 512MB, en discos SATA generalmente están entre 8MB a 32MB, soluciones alternas son poder configurar el tamaño del Write Cache del disco ó eventualmente comprar unidades de disco que tengan baterías para casos de perdida de energía o Cache no volátil

Tunning

b)El límite de commits por segundo

Según la tecnología de las unidades de almacenamiento que utilicemos vamos a obtener cantidad máxima de commits que podemos hacer por segundo, estos dependerán de muchos factores, por ejemplo en HDs la velocidad de rotación de los platos (más rápida la rotación mayor será la cantidad de commits que se podrán procesar) y en SSDs la tecnología de almacenamiento y el cache de ram que dispongan.

En un escenario básico, donde tenemos trabajando un disco duro, con el write cache desactivado tendremos estas tasas de operaciones máximas por segundo:

5400 RPM == 90 commit/segundo
7200 RPM == 120 commit/segundo
10000 RPM == 166 commit/segundo
15000 RPM == 250 commit/segundo

Para exceder estas tasas de operaciones deberíamos tener un método de cacheo de disco seguro.

En SSDs es posible llegar a tasas de 1,000 a 2,000 commits por segundo.

Usando arreglos de discos podemos superar ampliamente estas tasas de commits por segundo.

Configuración Avanzada - Afinando Postgresql

PostgreSQL y Gnu/Linux están muy integrados, un Sistema Operativo muy bien afinado dará como resultado un excelente rendimiento de la base de datos, sin embargo es posible configurar ciertos atributos del motor de la base de datos para optimizarla para diferentes escenarios.

Toda la configuración de PostgreSQL se realiza en el archivo "postgresql.conf" y cualquier cambio para que se active requerirá que se reinicie el servicio, en algunos casos bastará con hacer un reload.

```
[root@host]service postgresql restart  <-- baja el servicio totalmente y lo vuelve a
                                           iniciar.
[root@host]service postgresql reload   <-- el servicio no se baja, conexiones pueden
                                           seguir siendo aceptadas pero aparecerán
                                           con un rendimiento lento durante el
                                           periodo de tiempo que tome recargar el
                                           servicio, este tiempo dependerá de la
                                           concurrencia de usuarios cuando se haya
                                           ejecutado el Reload.
```

Otra forma de recargar las configuraciones modificadas en postgresql.conf es ejecutar este comando desde dentro de la base de datos.

```
template1=# select pg_reload_conf();
pg_reload_conf
---------------
t
```

Configuración Avanzada - Afinando Postgresql

a)SHARED_BUFFERS

El SHARED BUFFER es en si el área de intercambio de lectura y escritura de datos desde y hacia la base de datos.

El correr el INITDB el PostgreSQL detecta la cantidad de memoria del servidor y calcula esta valor, el problema es que asume que no deseas dar prioridad a la base de datos y coloca un valor muy bajo, 32mb o 24mb.

Los valores razonables deberían estar entre 25% a 40% del monto total de RAM, algunos expertos recomiendan que el 10% es más que necesario, pero en realidad dependerá del uso que le demos a la base de datos y que tan pesadas sean nuestras consultas,

Aumentar el valor de SHARED_BUFFER llevará generalmente a un aumento de su equivalente en el sistema operativo

Por ejemplo en este caso deseamos subir nuestro SHARED_BUFFER a 400mb

```
shared_buffers = 400MB
```

Acto seguido debemos configurar nuestro sistema operativo antes de que este nuevo valor pueda ser usado, sino tendremos un error del sistema, lo mejor es bajar la base de datos cuando se modifique este parámetro.

Configuración Avanzada - Afinando Postgresql

a)SHARED_BUFFERS - SHMMAX

Gnu/Linux por defecto levanta áreas de intercambio de datos en buffer de ram para cada aplicación que se esté ejecutando, por defecto los valores suelen ser pequeños por lo cual aumentar el SHARED_BUFFERS en el PostgreSQL hace que aumentemos el valor en el Gnu/Linux, tomar en cuenta que poner valores muy altos hará que si levantamos muchos servicios la RAM podría quedarnos corta, por lo cual no es buena idea levantar muchos servicios en un solo servidor.

Modificar /etc/sysctl.conf

Calculamos el nuevo valor de shmmax:

```
400mb x 1024kb = 409600   <-- cantidad de kb a utilizar por el buffer
409600 / 8 = 51200        <-- cantidad de páginas a utilizar por el buffer
51200 x 8192b = 419430400 <-- cantidad mínima de ram en bytes que se debe configurar
                              debe considerarse que hay otras aplicaciones que hacen
                              uso de esta cantidad de ram asignada.
```

Abrimos el archivo “/etc/sysctl.conf” y añadimos:

```
kernel.shmmax = 421527552  <-- añadimos un par de megas más para asegurarnos que no
                               Nos faltará ram.
```

Es necesario luego reiniciar el sistema operativo.

Configuración Avanzada - Afinando Postgresql

b)KERNEL SEMAPHORES

Los semáforos son objetos usados para procesar comunicaciones.

```
[root@host]# ipcs -l
```

<-- para ver la configuración de los semáforos actual del sistema.

```
---- Límites memoria compartida ----      Ubuntu 32bits
max number of segments = 4096
max seg size (kbytes) = 32768
max total shared memory (kbytes) = 8388608
tamaño mín. segmento (bytes) = 1

------ Límites semáforo --------
número máximo de matrices = 128
máx. semáforos por matriz = 250
máx. semáforos sistema = 32000
máx. oper. por llamada semop = 32
valor máx. semáforo = 32767

------ Mensajes: límites -------
máx. colas sistema = 7660
tamaño máx. mensaje (bytes) = 8192
tamaño máx. predeterminado cola (bytes) = 16384

------ Shared Memory Limits --------      Centos 64bits
max number of segments = 4096
max seg size (kbytes) = 18014398509465599
max total shared memory (kbytes) = 18014398442373116
min seg size (bytes) = 1

------ Semaphore Limits --------
max number of arrays = 128
max semaphores per array = 250
max semaphores system wide = 32000
max ops per semop call = 32
semaphore max value = 32767

------ Messages Limits --------
max queues system wide = 3675
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384
```

```
Ubuntu 64bits

------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 18014398509465599
max total shared memory (kbytes) =
18014398509481980
min seg size (bytes) = 1

------ Semaphore Limits --------
max number of arrays = 32000
max semaphores per array = 32000
max semaphores system wide = 1024000000
max ops per semop call = 500
semaphore max value = 32767

------ Messages Limits --------
max queues system wide = 32000
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384
```

Configuración Avanzada - Afinando Postgresql

b)KERNEL SEMAPHORES

PostgreSQL usa un total de 16+1 semáforos por cada conexión activa, 16 son de uso real y uno de control, mientras más conexiones requeriremos en nuestra base de datos más semáforos deberíamos tener disponibles para esto.

Para modificar los semáforos editamos el archivo /etc/sysctl.conf

```
número máximo de matrices = 128   <-- SEMMNI = (conexiones + procesos autovacumm) / 16
                                       redondeado hacia arriba
máx. semáforos por matriz = 250   <-- SEMMSL = como mínimo debería ser 17
máx. semáforos sistema = 32000    <-- SEMMNS = SEMMNI * SEMMSL
máx. oper. por llamada semop = 32
valor máx. Semáforo = 32767       <-- SEMVMX no es necesario cambiarlo generalmente
```

Para activar los nuevos valores requerirá reiniciar el sistema operativo.

Configuración Avanzada - Afinando Postgresql

c)CHECKPOINTS

Un CHECKPOINT es un área de almacenamiento de datos donde se almacena la información que está lista para bajarse al repositorio de datos final, esto se hace para asegurar que ante caídas del sistema el repositorio principal no se corrompa.

```
checkpoint_segments = 3        <-- cada segmento es de 16MB, 3 segmentos significa que
                                   tenemos 48MB de historia de procesos almacenados
                                   para reconstruir la información en caso de
                                   Desastres, más segmentos hará un sistema mas seguro
                                   Pero con un overhead mayor.
checkpoint_timeout = 5min      <-- tiempo máximo que el checkpoint podrá estar sin ser
                                   Pasado al repositorio principal.
checkpoint_warning = 30s       <-- lanza un mensaje al sistema indicando que se están
                                   Sucediendo checkpoints demasiado rápido en
                                   frecuencia de  creación, es decir que se genera un
                                   checkpoint en menos de 30s nos alertará
```

Configuración Avanzada - Afinando Postgresql

d)WORK_MEM

work_mem es el espacio de memoria que se usará para los ordenamientos de los datos cuando se ejecutan consultas, no existe una formula exacta de cálculo dependerá de que tanta data se mueva en las consultas que ejecutamos y cual es la concurrencia de ejecución de las consultas, por defecto el valor es muy bajo, solo 1MB.

La regla dice, si las consultas se ejecutan con poca concurrencia entonces asignar entre 2% a 4% de la ram disponible será ideal, pero si la concurrencia es alta entonces debe asignarse menos memoria debido a que cada consulta consume la misma cantidad de ram.

work_men = 163MB <-- 4% de 4gb (redondeado)

Configuración Avanzada - Afinando Postgresql

e)WAL

WAL (Write-Ahead Log) son páginas de datos que se generan para trabajar los datos en un área reservada antes de pasar al repositorio principal de datos, cuando un WAL culmina su labor se le cambia el identificador y se convierte en un CHECKPOINT y de ahí recién pasa en ciertos intervalos de tiempo al repositorio principal.

WAL_BUFFERS = 8 <-- es el valor por defecto, significa que tiene 8 * Longitud de página disponibles (el mínimo de una página de 8k hasta 32k), debería ser tan grande como lo necesiten nuestras transacciones más comunes.

wal_buffers = 64KB <-- podemos especificar el valor indicando una cantidad de KB pero Tomar en cuenta el peso de las páginas en KB.

wal_writer_delay = 200ms <-- lapso de tiempo en que el WAL realizará escrituras hacía disco después de informado por el "cliente" que la transacción ha concluido, un número muy bajo causará un incremento de overhead en los discos.

Configuración Avanzada - Afinando Postgresql

f)temp_buffers

Es la cantidad de memoria que tendrá asignada cada conexión al dbms para manejo de tablas temporales, esta no se libera hasta que la sesión muera, cada sesión podría manejar un tamaño diferente y mayor al de default (8mb) pero solo hasta antes de generar la primera tabla temporal.

g)max_locks_per_transaction

Permite indicar cuantos objetos se pueden bloquear en una transacción, un registro no se considera un objeto, el default es 64 solo valdría la pena subirlo si es que tenemos transacciones que bloquean muchos más objetos únicos (tablas) que este valor, obviamente el consumo de memoria crecerá.

h)max_connections

Determina cuantas conexiones soportará el dbms, subirlo significará un consumo de recursos adicional en el OS (sistema operativo), y en realidad no servirá de mucho si es que el OS no soporta esta carga de trabajo.

Configuración Avanzada - Afinando Postgresql

i)random_page_cost

Determina en que momento se van a utilizar índices ó búsquedas secuenciales, el valor por defecto es 4, se recomienda aumentarlo para discos lentos y bajarlo para discos rápidos.

Considere que los índices son efectivos si la variedad de datos es alta, si es baja entonces es probable que aunque se mueva la configuración se ejecuten búsquedas secuenciales.

j)effective_cache_size

Setea el tamaño de cache en RAM que se usará en una consulta, el valor por defecto es 128Mb, un valor alto facilitará el uso de índices, un valor bajo hará que se prefiera el uso de búsquedas secuenciales, se recomienda como máximo 50% del total de la RAM.

Configuración Avanzada - Afinando Postgresql

h)statement_timeout

Configura cuanto tiempo puede demorar un query en ejecutarse, evita que se lancen querys extremadamente pesados que demoren mucho tiempo en ejecutarse, por defecto es 0 (desactivado) pero se expresa en milisegundos, OjO configurarlo afectará todos los querys.

MANIPULACION DE GRANDES VOLUMENES DE DATOS

La solución para mejorar la manipulación de grandes volúmenes de datos es siempre objeto de mucho estudio en las bases de datos relaciones.

Este es un problema que no se resuelve solo con ampliaciones de hardware, es necesario desde el Diseño Físico de nuestra base de datos tomar consideraciones técnicas que nos faciliten el manejo de volúmenes grandes de datos.

Segmentación De Tablas

Las tablas particionadas nos permiten crear diferentes espacios de almacenamiento para una data en común, básicamente es la aplicación de una tabla heredada.

```
template1=# create table tbl_padre(cuenta char(10));
template1=# create table tbl_hija10(cuenta char(10)) inherits (tbl_padre);
template1=# create table tbl_hija20(cuenta char(10)) inherits (tbl_padre);

template1=# insert into tbl_hija20 values('20.01');
template1=# insert into tbl_hija10 values('10.01');
template1=# select * from tbl_padre;

   cuenta
------------
   10.01
   20.01

template1=# select * from tbl_hija10;

   cuenta
------------
   10.01
```

Vamos a poner el caso de una tabla que contendrá miles de registros que estará particionada.

```
python=# create table padre (id int);
python=# create table hijo10 (origen int) inherits (padre);
python=# create table hijo20 (origen int) inherits (padre);
```

Esta tabla almacenará números del 1 al 30,000, las tablas particionadas solo tienen sentido cuando los datos se distribuyen de forma lógica entre ellas, esto podemos hacerlo a través de RULEs o TRIGGERs, en este caso vamos a usar los siguientes RULEs:

```
python=# create rule regla10 as on insert to padre where new.id > 9999
and  new.id < 19999 do instead insert into hijo10 (id) values (new.id);
```

```
python=# create rule regla20 as on insert to padre where new.id > 19999
and  new.id < 29999 do instead insert into hijo20 (id) values (new.id);
```

Esta función solo se crea para llenar la base de datos:

```
create or replace function poblardata() returns integer
as $$
declare
    i int;
    b float;
begin
    for i in 1..30000 loop
        b = i * random();
        insert into padre (id) values (b);
    end loop;
    return 1;
end;
$$ language plpgsql;

python=# select * from poblardata();
 poblardata
------------
          1
(1 row)
```

Aquí podemos ver que la tabla PADRE tienen los 30,000 registros ingresados

```
python=# select count(*) from padre;
 count
-------
 30000
```

Sin embargo las tablas hijas almacenan dentro de si la información que les corresponde según las reglas creadas:

```
python=# select count(*) from hijo10;
 count
-------
  7127
python=# select count(*) from hijo20;
 count
-------
  1906
```

Esto deja, a nivel físico, a la tabla PADRE solo con su segmento de dato

```
python=# select  count(*) from only padre ;
 count
-------
 20967
```

Cuando ejecutamos una búsqueda podemos notar que trabajan las 3 tablas como si fuera una sola.

```
python=# explain analyze select * from padre where id > 1 and id < 10;
                                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
Append  (cost=0.00..584.00 rows=84 width=4) (actual time=0.008..3.123 rows=68
loops=1)
   ->  Seq Scan on padre  (cost=0.00..407.50 rows=82 width=4) (actual
                          time=0.008..2.147 rows=68 loops=1)
         Filter: ((id > 1) AND (id < 10))
         Rows Removed by Filter: 20899
   ->  Seq Scan on hijo10  (cost=0.00..138.91 rows=1 width=4) (actual
                          time=0.727..0.727 rows=0 loops=1)
         Filter: ((id > 1) AND (id < 10))
         Rows Removed by Filter: 7127
   ->  Seq Scan on hijo20  (cost=0.00..37.59 rows=1 width=4) (actual
                          time=0.241..0.241 rows=0 loops=1)
         Filter: ((id > 1) AND (id < 10))
         Rows Removed by Filter: 1906
 Planning time: 0.109 ms
 Execution time: 3.143 ms
(12 rows)
```

Según la búsqueda trabajará una u otra tabla, pero hasta este momento las búsquedas no son óptimas porque se desperdician demasiados recursos filtrando datos no útiles.

```
python=# explain analyze select * from padre where id > 10000 and id < 10500;
------------------------------------------------------------------------------
Append  (cost=0.00..584.00 rows=516 width=4) (actual time=1.627..2.746 rows=525
loops=1)
…
   ->  Seq Scan on hijo10  (cost=0.00..138.91 rows=514 width=4) (actual
time=0.005..0.863 rows=525 loops=1)
         Filter: ((id > 10000) AND (id < 10500))
         Rows Removed by Filter: 6602

python=# explain analyze select * from padre where id > 20000 and id < 20500;
------------------------------------------------------------------------------
Append  (cost=0.00..584.00 rows=192 width=4) (actual time=2.335..2.566 rows=189
loops=1)
…
   ->  Seq Scan on hijo20  (cost=0.00..37.59 rows=190 width=4) (actual
time=0.007..0.220 rows=189 loops=1)
         Filter: ((id > 20000) AND (id < 20500))
         Rows Removed by Filter: 1717
```

Las búsquedas como ya sabemos se aceleran gracias a los índices, por ello debemos indexar nuestras tablas, podemos indexar una o todas las tablas heredadas, eso dependerá de nuestras necesidades.

```
python=# create index idx1 on hijo10(id);
python=# create index idx2 on hijo20(id);
python=# explain analyze select * from padre where id > 10000 and id < 10500;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Append  (cost=0.00..469.09 rows=517 width=4) (actual time=1.800..1.985
rows=525 loops=1)
   ->  Seq Scan on padre  (cost=0.00..407.50 rows=1 width=4) (actual
                           time=1.743..1.743 rows=0 loops=1)
         Filter: ((id > 10000) AND (id < 10500))
         Rows Removed by Filter: 20967
   ->  Bitmap Heap Scan on hijo10  (cost=13.56..53.29 rows=515 width=4) (actual
                                    time=0.056..0.182 rows=525 loops=1)
         Recheck Cond: ((id > 10000) AND (id < 10500))
         Heap Blocks: exact=32
         ->  Bitmap Index Scan on idx1  (cost=0.00..13.43 rows=515 width=0)
             (actual time=0.047..0.047 rows=525 loops=1)
               Index Cond: ((id > 10000) AND (id < 10500))
   ->  Index Only Scan using idx2 on hijo20  (cost=0.28..8.30 rows=1 width=4)
       (actual time=0.003..0.003 rows=0 loops=1)
         Index Cond: ((id > 10000) AND (id < 10500))
         Heap Fetches: 0
```

Se deben indexar todas las tablas por separado para tener la mejor ejecución de nuestras consultas.

```
python=# create index idx0 on padre(id);

python=# explain analyze select * from padre where id > 10000 and id < 10500;
                                 QUERY PLAN
-------------------------------------------------------------------------
Append  (cost=0.29..69.89 rows=517 width=4) (actual time=0.055..0.261 rows=525
loops=1)
   ->  Index Only Scan using idx0 on padre  (cost=0.29..8.31 rows=1 width=4)
                                    (actual time=0.003..0.003 rows=0 loops=1)
         Index Cond: ((id > 10000) AND (id < 10500))
         Heap Fetches: 0
   ->  Bitmap Heap Scan on hijo10  (cost=13.56..53.29 rows=515 width=4) (actual
                                     time=0.052..0.202 rows=525 loops=1)
         Recheck Cond: ((id > 10000) AND (id < 10500))
         Heap Blocks: exact=32
         ->  Bitmap Index Scan on idx1  (cost=0.00..13.43 rows=515 width=0)
                                  (actual time=0.040..0.040 rows=525 loops=1)
               Index Cond: ((id > 10000) AND (id < 10500))
   ->  Index Only Scan using idx2 on hijo20  (cost=0.28..8.30 rows=1 width=4)
                                     (actual time=0.002..0.002 rows=0 loops=1)
         Index Cond: ((id > 10000) AND (id < 10500))
         Heap Fetches: 0
 Planning time: 0.244 ms
 Execution time: 0.311 ms
```

Indexar la tabla padre no indexa por defecto la data en las tablas hijas.

```
python=# drop index idx1; drop index idx2;

python=# explain analyze select * from padre where id > 10000 and id < 10500;
                                  QUERY PLAN
-------------------------------------------------------------------------------
Append  (cost=0.29..184.80 rows=516 width=4) (actual time=0.012..1.159 rows=525
loops=1)
   ->  Index Only Scan using idx0 on padre  (cost=0.29..8.31 rows=1 width=4)
                                    (actual time=0.002..0.002 rows=0 loops=1)
         Index Cond: ((id > 10000) AND (id < 10500))
         Heap Fetches: 0
   ->  Seq Scan on hijo10  (cost=0.00..138.91 rows=514 width=4) (actual
                                      time=0.008..0.850 rows=525 loops=1)
         Filter: ((id > 10000) AND (id < 10500))
         Rows Removed by Filter: 6602
   ->  Seq Scan on hijo20  (cost=0.00..37.59 rows=1 width=4) (actual
                                        time=0.247..0.247 rows=0 loops=1)
         Filter: ((id > 10000) AND (id < 10500))
         Rows Removed by Filter: 1906
 Planning time: 0.200 ms
 Execution time: 1.225 ms
```

Índices Parciales

Los índices parciales nos permiten tener indexada una porción de la data de una tabla.

```
Create index idx_nombre on tabla1 (campo1)
             where campo1 > 100 and campo1 <= 100000;
```

Si la búsqueda sobre esta tabla escapa a los valores especificados entonces no se usará el índice.

Podemos especificar un índice de este tipo donde los valores WHERE sean diferentes a los que estamos indexando.

```
Create index idx_nombre on tabla1 (campo1)
             where campo2 > 100 and campo2 <= 100000;

Select * from tabla1 where campo1 = 10 and campo2 = 1000;   <-- usará el índice

Select * from tabla1 where campo1 = 10;                     <-- no usará el
                                                                índice

Select * from tabla1 where campo2 > 1000 and campo3 = 1000; <-- podría usar el
                                                                índice
```

Índices Parciales

```
python=# create index idx0;
python=# create index idx01 on padre(id) where id >0  and id <= 2000;
python=# create index idx02 on padre(id) where id >2000  and id <= 6000;
python=# create index idx03 on padre(id) where id >6000  and id <= 99999;

python=# explain analyze select * from padre where id > 10000 and id < 10500;
                                 QUERY PLAN
-------------------------------------------------------------------------------
Append  (cost=0.28..184.80 rows=516 width=4) (actual time=0.019..1.090 rows=525
                                                 loops=1)
   ->  Index Only Scan using idx03 on padre  (cost=0.28..8.30 rows=1 width=4)
                                    (actual time=0.011..0.011 rows=0 loops=1)
         Index Cond: ((id > 10000) AND (id < 10500))
         Heap Fetches: 0
   ->  Seq Scan on hijo10  (cost=0.00..138.91 rows=514 width=4) (actual
                                           time=0.008..0.830 rows=525 loops=1)
         Filter: ((id > 10000) AND (id < 10500))
         Rows Removed by Filter: 6602
   ->  Seq Scan on hijo20  (cost=0.00..37.59 rows=1 width=4) (actual
                                             time=0.196..0.196 rows=0 loops=1)
         Filter: ((id > 10000) AND (id < 10500))
         Rows Removed by Filter: 1906
 Planning time: 0.282 ms
 Execution time: 1.135 ms
(12 rows)
```

Índices Parciales

```
python=# explain analyze select * from padre where id > 100and id < 500;
                          QUERY PLAN
-----------------------------------------------------------------------------
Append  (cost=43.56..341.27 rows=1883 width=4) (actual time=0.182..1.870
                                                rows=1888 loops=1)
   ->  Bitmap Heap Scan on padre  (cost=43.56..164.78 rows=1881 width=4) (actual
                                   time=0.182..0.671 rows=1888 loops=1)
         Recheck Cond: ((id > 100) AND (id < 500))
         Heap Blocks: exact=93
         ->  Bitmap Index Scan on idx01  (cost=0.00..43.09 rows=1881 width=0)
                                         (actual time=0.166..0.166 rows=1888 loops=1)
               Index Cond: ((id > 100) AND (id < 500))
   ->  Seq Scan on hijo10  (cost=0.00..138.91 rows=1 width=4) (actual
                            time=0.775..0.775 rows=0 loops=1)
         Filter: ((id > 100) AND (id < 500))
         Rows Removed by Filter: 7127
   ->  Seq Scan on hijo20  (cost=0.00..37.59 rows=1 width=4) (actual
                            time=0.214..0.214 rows=0 loops=1)
         Filter: ((id > 100) AND (id < 500))
         Rows Removed by Filter: 1906
 Planning time: 0.154 ms
 Execution time: 1.987 ms
(14 rows)
```

TableSpaces

Los tablespaces nos permite generar áreas de almacenamiento de objetos de una db en directorios ó unidades de almacenamiento diferentes a la determinada por defecto.

La ventaja de esto es que podemos balancear el trabajo de nuestra DB en diversas unidades haciendo que nuestra performanse mejore notablemente.

Un tablespace puede ser:

- **Otro disco duro**
- **Un “disco” de estado solido (SSD)**
- **Una memoria USB**
- **Un “disco” virtual (en RAM)**
- **Otros tipos de unidades de almacenamiento**

TableSpaces

CREATE TABLESPACE tablespacename [OWNER username] LOCATION 'directory'

El owner siempre es por defecto el creador, el directorio donde se creará el tablespace debe estar VACIO y tener los permisos del usuario de Linux que ejecuta el servicio del dbms (en el caso de CentOS es el usuario “postgres”).

Como root (o con sudo):

```
root@depeche:~# mkdir /home/tablespace
root@depeche:~# chown postgres:postgres /home/tablespace/
```

En consola de PgSQL:

```
template1=# create tablespace ts1 location '/home/tablespace';
```

Como usuario postgres:

```
 postgres@ernesto-VirtualBox:/home/tablespace$ ls -la
total 12
drwx------ 3 postgres postgres 4096 dic  1 15:24 .
drwxr-xr-x 4 root     root     4096 dic  1 15:23 ..
drwx------ 2 postgres postgres 4096 dic  1 15:24 PG_9.4_201409291 <-- directorio
                                                                     del tablespace para la data
```

TableSpaces

Ahora creamos una tabla nueva en el nuevo TableSpace e insertamos una gran cantidad de registros.

EN PSQL:

```
python=# create table prueba (id int) tablespace ts1;
CREATE TABLE
python=# insert into prueba values (generate_series(1,1000000));
INSERT 0 1000000
```

En Linux podemos verificar si se están generando los nuevos archivos de datos en el nuevo TableSpace.

```
postgres@ernesto-VirtualBox:/home/tablespace$ cd PG_9.4_201409291/
postgres@ernesto-VirtualBox:/home/tablespace/PG_9.4_201409291$ ls
24685
postgres@ernesto-VirtualBox:/home/tablespace/PG_9.4_201409291$ cd 24685/
postgres@ernesto-VirtualBox:/home/tablespace/PG_9.4_201409291/24685$ ls -la
total 35440
drwx------ 2 postgres postgres     4096 dic  1 15:28 .
drwx------ 3 postgres postgres     4096 dic  1 15:28 ..
-rw------- 1 postgres postgres 36249600 dic  1 15:28 24762
-rw------- 1 postgres postgres    32768 dic  1 15:28 24762_fsm
```

TableSpaces

Podemos usar esta funcionalidad con:

- **Base de datos**
- **Tablas**
- **Índices**

Se recomienda que los Tablespaces creados en discos virtuales ó memorias USB solo almacenen objetos de la db de los cuales se puedo prescindir como tablas temporales, esto debido a que son muy volátiles y fácil de corromper o perder los datos (por ejemplo tener un tablespace en disco virtual desaparece si alguien reinicia el servidor).

Los tablespace de índices tienen una muy buen rendimiento en discos SSD.

Más sobre tablespace en ram:
http://wiki.postgresql.org/wiki/La_verdad_y_la_mentira_de_los_tablespaces_en_memoria

Tablas Particionadas

El particionamiento de tablas nos permite crear diferentes repositorios de los datos (en diversas unidades si es necesario) para optimizar el almacenamiento de volumenes grandes de datos.

Normalmente al crear una tabla se genera un solo repositorio para esta:

```
test5=# create table normal (id int, valor int) tablespace ts1;
CREATE TABLE
test5=# insert into normal values (generate_series(1,10000) * random(),1);


drwx------. 2 postgres postgres     4096 nov 19 20:24 .
drwx------. 3 postgres postgres     4096 nov 19 20:13 ..
-rw-------. 1 postgres postgres 40239104 nov 19 20:19 25189
-rw-------. 1 postgres postgres    32768 nov 19 20:15 25189_fsm
```

Tablas Particionadas

Para crear una tabla particionada primero procedemos a crear la tabla principal:

```
test5=# create table particionada (id int, valor int) partition by range (id) tablespace ts1;
CREATE TABLE
```

Luego procedemos a crear las particiones:

```
test5=# create table particionada1 partition of particionada for values from (0) to (500)
tablespace ts1;
CREATE TABLE

test5=# create table particionada2 partition of particionada for values from (500) to (1000)
tablespace ts1;
CREATE TABLE


[root@localhost 25184]# ls -la
total 39336
drwx------. 2 postgres postgres     4096 nov 19 20:24 .
drwx------. 3 postgres postgres     4096 nov 19 20:13 ..
-rw-------. 1 postgres postgres 40239104 nov 19 20:19 25189
-rw-------. 1 postgres postgres    32768 nov 19 20:15 25189_fsm
-rw-------. 1 postgres postgres        0 nov 19 20:23 25207      ← particiones
-rw-------. 1 postgres postgres        0 nov 19 20:24 25210      ← particiones
```

Tablas Particionadas

La data se dividirá entre las particiones asignadas:

```
test5=# insert into particionada values (generate_series(1,500),10);
INSERT 0 500
-rw-------. 1 postgres postgres    24576 nov 19 20:35 25207
-rw-------. 1 postgres postgres    24576 nov 19 20:35 25207_fsm
-rw-------. 1 postgres postgres        0 nov 19 20:34 25210

test5=# insert into particionada values (generate_series(501,999),10);
INSERT 0 499
test5=# insert into particionada values (generate_series(501,999),10);
INSERT 0 499
-rw-------. 1 postgres postgres    24576 nov 19 20:36 25207
-rw-------. 1 postgres postgres    24576 nov 19 20:35 25207_fsm
-rw-------. 1 postgres postgres    57344 nov 19 20:37 25210
-rw-------. 1 postgres postgres    24576 nov 19 20:36 25210_fsm
```

En este caso se intenta ingresar un valor no considerado entre las particiones y el sistema devuelve un error.

```
test5=# insert into particionada values (generate_series(1000,1999),10);
ERROR:  no partition of relation "particionada" found for row
DETAIL:  Partition key of the failing row contains (id) = (1000).

test5=# insert into particionada values (generate_series(0,999) * random(),10);
INSERT 0 1000        ← repetimos esto 5 veces para generar valores aleatorios
-rw-------. 1 postgres postgres   163840 nov 19 20:39 25207
-rw-------. 1 postgres postgres    24576 nov 19 20:35 25207_fsm
-rw-------. 1 postgres postgres    81920 nov 19 20:39 25210
-rw-------. 1 postgres postgres    24576 nov 19 20:36 25210_fsm
-rw-------. 1 postgres postgres     8192 nov 19 20:37 25210_vm
```

Tablas Particionadas

El generador de rutas del queries es capaz de determinar en lasconsultas cuando debe hacer funcionar una partición y cuando no.

```
test5=# explain analyze select  * from particionada;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
Append  (cost=0.00..95.63 rows=6563 width=8) (actual time=0.007..1.502 rows=6498 loops=1)
   ->  Seq Scan on particionada1  (cost=0.00..63.96 rows=4396 width=8) (actual time=0.007..0.506 rows=4396 loops=1)
   ->  Seq Scan on particionada2  (cost=0.00..31.67 rows=2167 width=8) (actual time=0.006..0.223 rows=2102 loops=1)
```

```
test5=# explain analyze select  * from particionada where id < 400;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
Append  (cost=0.00..74.95 rows=3968 width=8) (actual time=0.017..2.664 rows=3965 loops=1)
   ->  Seq Scan on particionada1  (cost=0.00..74.95 rows=3968 width=8) (actual time=0.017..1.488 rows=3965 loops=1)
         Filter: (id < 400)
         Rows Removed by Filter: 431
```

```
test5=# explain analyze select  * from particionada where id > 400;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
Append  (cost=0.00..112.04 rows=2595 width=8) (actual time=0.072..2.600 rows=2529 loops=1)
   ->  Seq Scan on particionada1  (cost=0.00..74.95 rows=428 width=8) (actual time=0.072..0.741 rows=427 loops=1)
         Filter: (id > 400)
         Rows Removed by Filter: 3969
   ->  Seq Scan on particionada2  (cost=0.00..37.09 rows=2167 width=8) (actual time=0.010..0.975 rows=2102 loops=1)
         Filter: (id > 400)
```

```
test5=# explain analyze select  * from particionada where id > 600;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
Append  (cost=0.00..37.09 rows=1624 width=8) (actual time=0.068..1.970 rows=1563 loops=1)
   ->  Seq Scan on particionada2  (cost=0.00..37.09 rows=1624 width=8) (actual time=0.067..1.158 rows=1563 loops=1)
         Filter: (id > 600)
         Rows Removed by Filter: 539
```

Tablas Particionadas

El indexamiento se debe aplicar a cada partición.

```
test5=# create index pidx1 on particionada(id);
ERROR:  cannot create index on partitioned table "particionada"

test5=# create index pidx1 on particionada1(id);
CREATE INDEX

test5=# explain analyze select  * from particionada where id = 400;
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
Append  (cost=4.34..18.57 rows=7 width=8) (actual time=0.069..0.079 rows=4 loops=1)
   ->  Bitmap Heap Scan on particionada1  (cost=4.34..18.57 rows=7 width=8) (actual time=0.068..0.077 rows=4
loops=1)
         Recheck Cond: (id = 400)
         Heap Blocks: exact=4
         ->  Bitmap Index Scan on pidx1  (cost=0.00..4.33 rows=7 width=0) (actual time=0.056..0.056 rows=4
loops=1)
               Index Cond: (id = 400)


test5=# explain analyze select  * from particionada where id in (400,800);
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
Append  (cost=8.67..64.71 rows=20 width=8) (actual time=0.051..0.532 rows=7 loops=1)
   ->  Bitmap Heap Scan on particionada1  (cost=8.67..27.62 rows=13 width=8) (actual time=0.050..0.059 rows=4
loops=1)
         Recheck Cond: (id = ANY ('{400,800}'::integer[]))
         Heap Blocks: exact=4
         ->  Bitmap Index Scan on pidx1  (cost=0.00..8.67 rows=13 width=0) (actual time=0.041..0.041 rows=4
loops=1)
               Index Cond: (id = ANY ('{400,800}'::integer[]))
   ->  Seq Scan on particionada2  (cost=0.00..37.09 rows=7 width=8) (actual time=0.073..0.469 rows=3 loops=1)
         Filter: (id = ANY ('{400,800}'::integer[]))
         Rows Removed by Filter: 2099
```

Tablas Particionadas

Las tablas principales no pueden tener Primary Key, pero sus particiones si

```
test5=# alter table particionada add column unico serial;
ALTER TABLE


test5=# alter table particionada add constraint pk1 primary key (unico);
ERROR:  primary key constraints are not supported on partitioned tables
LINE 1: alter table particionada add constraint pk1 primary key (uni...
                                    ^
test5=# alter table particionada1 add constraint pk1 primary key (unico);
ALTER TABLE
```

Algunas consideraciones adicionales:

1)Se puede insertar datos directamente en las particiones
2)ON CONFLICT solo funciona en las particiones no en la tabla principal
3)Un UPDATE genera movimientos de datos entre las particiones
4)Los TRIGGERs deben aplicarse a las particiones
5)Al dropear la tabla principal se eliminan sus particiones

Tablas Particionadas

Particiones basadas en LISTas de datos

```
test5=# create table pa (id serial, tipo int) partition by list (tipo)
tablespace ts1;
CREATE TABLE

test5=# create table pa1 partition of pa for values in(1,2) tablespace ts1;
CREATE TABLE

test5=# create table pa2 partition of pa for values in(3,4) with
(parallel_workers=8) tablespace ts1;    ← parallel_workers ayuda a mejorar los
                                          procesos de escritura en la tabla
CREATE TABLE


test5=# \timing 1
Timing is on.

test5=# insert into pa values (generate_series(1,10000000),3);
INSERT 0 10000000
Time: 27347,502 ms (00:27,348)

test5=# insert into pa values (generate_series(1,10000000),1);
INSERT 0 10000000
Time: 28503,006 ms (00:28,503)
test5=#
```

Tables Unlogged

Las tablas Unlogged son tablas que se escriben directamente en el directorio de DATA y que no tienen sus archivos de respaldo (para casos de reconstrucción en el Pg_XLOG), esto hace que el rendimiento en el caso de gran cantidad de inserciones sea muy bueno, pero corremos el riesgo de en caso de siniestros de quedarnos con una tabla corrupta.

```
db=> create table con_log(id int);
db=> create unlogged table sin_log(id int);

db=> \timing
Timing is on.
db=> insert into con_log values (generate_series(1,10000000));
INSERT 0 10000000
Time: 28005,327 ms
db=> insert into sin_log values (generate_series(1,10000000));
INSERT 0 10000000
Time: 5529,174 ms
```

Automatizando tareas con CRON

PostgreSQL utiliza el sistema de automatización de tareas de Gnu/Linux para automatizar tareas rutinarias como por ejemplo las de mantenimiento.

CRON es el demonio más utilizado, cualquier usuario puede eventualmente crear una tarea automatizada, en este caso utilizaremos el usuario "postgres" del sistema operativo para nuestras tareas que afectan directamente a las bases de datos.

En CentOS el editor por defecto es VI pero como es algo complicado de usar utilizaremos NANO, para ello exportamos la variable "EDITOR" de Gnu/Linux

```
   [root@host] su – postgres
   [postgres@host] export EDITOR=nano
   [postgres@host] crontab -e                    <-- entramos al editor del cron

        0-59 * * * * echo "hola" >> /var/lib/pgsql/file.txt   <-- grabar la palabra "hola"
y la hora
        0-59 * * * * date >> /var/lib/pgsql/file.txt              cada minuto en el
archivo file.txt

   [root@host] service cron restart           <-- es necesario reiniciar el demonio del
"cron" cada
                                                  Vez que se cambie una configuración
```

Automatizando tareas con CRON

```
Los rangos de tiempo se presentan así:

* * * *  *   COMANDO
M H D MO DS

M  = minuto
H  = hora en formato de 0-23
D  = día en formato de 1-31 (deben especificarse días válidos)
MO = mes en formato 1-12
DS = día de la semana en formato 0-7 donde 0 ó 7 es domingo
COMANDO = comando del sistema operativo que debe ejecutarse

1 * * * *      COMANDO <-- ejecutará el comando al minuto 1 de cada hora
0-59 * * * *   COMANDO <-- ejecutará el comando cada minuto
0-59/2 * * * * COMANDO <-- ejecutará el comando cada 2 minutos

5 2 * * *      COMANDO <-- ejecutará el comando al minuto 5 de la hora 2 de
cada día
5 2 20 * *     COMANDO <-- ejecutará el comando al minuto 5 de la hora 2 de
cada día
                           20 del mes

1 * * 1 *      COMANDO <-- ejecutará el comando al minuto 1 de todos los
Enero
1 * * 1-12/4 * COMANDO <-- ejecutará el comando al minuto 1 de cada 3er mes

1 * * * 1      COMANDO <-- ejecutará el comando al minuto 1 de todos los
Lunes
```

Automatizando tareas con CRON

El COMANDO puede ser un archivo aparte que pueda ser interpretado por el PSQL, debe verificar que se cuente con el acceso pertinente para ejecutar los querys que este archivo aparte va a ejecutar, por ello es lo mejor ejecutar el CRON como usuario POSTGRES.

```
[postgres@host] crontab -e

  * * * * * /var/lib/pgsql/cron.sh
```

Creamos un archivo cron.sh (o cualquier nombre) y le asignamos permisos de ejecución.

```
[postgres@host]touch cron.sh
[postgres@host]chmod +x cron.sh
[postgres@host]mcedit cron.sh

  #!/bin/sh
  # Programa en bash script de ejemplo de un CRON para PostgreSQL
  echo "Se inicia el log" >> file.txt
  date >> file.txt
  echo "Ejecuta un query con salida al log"
  export PGPASSWORD=dbadmin
  psql prueba -c "select * from tablita limit 5"  -U dbadmin  >> file.txt
  echo "Inicia un proceso de backup"
  FECHA=$(date)
  pg_dump prueba > $FECHA.dump -U dbadmin
  echo "Backup finalizado, proceso concluido $FECHA" >> file.txt
```