Titulación: Grado en Ingeniería Informática y Sistemas de Información
Curso: 2019-2020. Convocatoria Ordinaria de Junio
Asignatura: Bases de Datos Avanzadas – Laboratorio

Practica 1: Arquitectura PostgreSQL y almacenamiento físico

ALUMNO 1:

Nombre y Apellidos: Ana Cortés Cercadillo

DNI:

ALUMNO 2:

Nombre y Apellidos: Carlos Javier Hellín Asensio

DNI:

Fecha: 3 de Marzo_____

Profesor Responsable: José Carlos Holgado

Mediante la entrega de este fichero los alumnos aseguran que cumplen con la normativa de autoría de trabajos de la Universidad de Alcalá, y declaran éste como un trabajo original y propio.

En caso de ser detectada copia, se calificará la asignatura como Suspensa – Cero.

Plazos

Trabajo de Laboratorio: Semana 27 Enero, 3 Febrero, 10 Febrero, 17 Febrero y 24 de Febrero.

Entrega de práctica: Día 3 de Marzo. Aula Virtual

Documento a entregar: Este mismo fichero con las respuestas a las cuestiones planteadas. Si se entrega en formato electrónico el fichero se deberá llamar: **`DNIdelosAlumnos_PECL1.doc`**

AMBOS ALUMNOS DEBEN ENTREGAR EL FICHERO EN LA PLATAFORMA.

Introducción

En esta primera práctica se introduce el sistema gestor de bases de datos PostgreSQL versión 11 o 12. Está compuesto básicamente de un motor servidor y de una serie de clientes que acceden al servidor y de otras herramientas externas. En esta primera práctica se entrará a fondo en la arquitectura de PostgreSQL, sobre todo en el almacenamiento físico de los datos y del acceso a los mismos.

Actividades y Cuestiones

Almacenamiento Físico en PostgreSQL

Cuestión 1. Crear una nueva Base de Datos que se llame **MiBaseDatos**. ¿En qué directorio se crea del disco duro, cuanto ocupa el mismo y qué ficheros se crean? ¿Por qué?

La base de datos se crea en el directorio de datos que se puede saber al consultar con show data_directory(); dentro de PostgreSQL.

```
postgres=# show data_directory;
       data_directory
-----------------------------
 /var/lib/postgresql/12/main
(1 fila)
```

Consultando la documentación de PostgreSQL se sabe que el subdirectorio data es el que contiene los subdirectorios por cada base de datos. Estos subdirectorios son nombrados por su identificador objeto (oid) por lo que para identificar qué subdirectorio contiene la base de datos MiBaseDatos hay que consultar al catálogo pg_database que almacena información sobre las bases de datos disponibles.

```
postgres=# create database MiBaseDatos;
CREATE DATABASE
postgres=# select datname, oid from pg_database where datname = 'mibasedatos';
   datname   |  oid
-------------+-------
 mibasedatos | 16385
(1 fila)
```

En este ejemplo, el oid de MiBaseDatos es 16385 y dicho subdirectorio existe en base.

```
postgres@ubuntu:~$ cd /var/lib/postgresql/12/main/
postgres@ubuntu:~/12/main$ ls -ld base/16385/
drwx------ 2 postgres postgres 12288 feb  3 09:44 base/16385/
```

También se puede saber cuánto ocupa el mismo

```
postgres@ubuntu:~/12/main$ du -sh base/16385/
7,9M    base/16385/
```

que coincide si se consulta en PostgreSQL el tamaño de la base de datos.

```
postgres=# SELECT pg_size_pretty(pg_database_size('mibasedatos'));
 pg_size_pretty
----------------
 7977 kB
(1 fila)
```

Los ficheros que se crean dentro del subdirectorio de 16385 al crear una nueva base de datos son la mayoría de los catálogos del sistema que se copian de las plantillas de la base de datos durante la fase de creación de MiBaseDatos (CREATE DATABASE) y que, después, son específicos para la base de datos creada. Estos ficheros son necesarios porque los catálogos del sistema almacenan metadatos de esquemas, como información de las tablas y columnas, así como información interna necesaria para su funcionamiento.

Por ejemplo, el fichero con el oid 1247 en el subdirectorio 16385 de la base de datos creada.

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/1247*
-rw------- 1 postgres postgres  80K feb  3 09:44 base/16385/1247
-rw------- 1 postgres postgres  24K feb  3 09:44 base/16385/1247_fsm
-rw------- 1 postgres postgres 8,0K feb  3 09:44 base/16385/1247_vm
```

Se puede saber qué catálogo del sistema es al consultar con pg_class el cual almacena todo lo que tenga una columna o es similar a una tabla.

```
postgres=# SELECT relname, oid FROM pg_class WHERE oid = 1247;
 relname | oid
---------+------
 pg_type | 1247
(1 fila)
```

En este caso el fichero 1247 es el catálogo pg_type que almacena información sobre los tipos de datos.

Además, se pueden encontrar dos archivos asociados con sufijo "_fsm" y "_vm". Estos son el mapa de espacio libre (Free Space Map) y mapa de visibilidad (Visibility Map) que, respectivamente, almacenan la información de la capacidad de espacio y la visibilidad en cada página dentro del archivo de tabla.

Otros de los ficheros que se crean son pg_filenode.map que se trata del archivo del mapeo de relaciones, pg_internal.init para la caché de las relaciones y PG_VERSION que simplemente contiene la versión de PostgreSQL.

```
-rw------- 1 postgres postgres  512 feb  3 09:44 pg_filenode.map
-rw------- 1 postgres postgres 145K feb  3 09:44 pg_internal.init
-rw------- 1 postgres postgres    3 feb  3 09:44 PG_VERSION
postgres@ubuntu:~/12/main$ cat base/16385/PG_VERSION
12
```

Cuestión 2. Crear una nueva tabla que se llame **MiTabla** que contenga un campo que se llame id_cliente de tipo integer que sea la Primary Key, otro campo que se llame nombre de tipo text, otro que se llame apellidos de tipo text, otro dirección de tipo text y otro puntos que sea de tipo integer. ¿Qué ficheros se han creado en esta operación? ¿Qué guarda cada uno de ellos? ¿Cuánto ocupan? ¿Por qué?

Al crear la tabla uno de los ficheros que se crean es la que tiene guardada pg_class con el nombre de MiTabla, cuyo oid en este caso es 16394 y sirve para guardar los datos de la tabla que se encuentra ahora mismo vacía.

```
mibasedatos=# CREATE TABLE MiTabla(id_cliente integer PRIMARY KEY, nombre TEXT,
   apellidos TEXT, direccion TEXT, puntos INTEGER);
CREATE TABLE
mibasedatos=# SELECT relname, oid FROM pg_class WHERE relname = 'mitabla';
 relname |  oid
---------+-------
 mitabla | 16394
(1 fila)
```

Otro de los ficheros, incluyendo el de la tabla, que se crean se han identificado con la comparación del directorio antes y después de crear la tabla y debido a que los oid se crean secuencialmente, se sabe que estos ficheros han sido creados durante la operación CREATE TABLE y, además, se muestran cuánto ocupan.

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16*
-rw------- 1 postgres postgres    0 feb  3 11:52 base/16385/16394
-rw------- 1 postgres postgres    0 feb  3 11:52 base/16385/16397
-rw------- 1 postgres postgres 8,0K feb  3 11:52 base/16385/16399
-rw------- 1 postgres postgres 8,0K feb  3 11:52 base/16385/16400
```

Al ser consultados en pg_class, se demuestra que tienen relación con la tabla MiTabla.

```
mibasedatos=# SELECT relname, oid FROM pg_class WHERE oid = 16397;
    relname     |  oid
----------------+-------
 pg_toast_16394 | 16397
(1 fila)

mibasedatos=# SELECT relname, oid FROM pg_class WHERE oid = 16399;
       relname        |  oid
----------------------+-------
 pg_toast_16394_index | 16399
(1 fila)

mibasedatos=# SELECT relname, oid FROM pg_class WHERE oid = 16400;
   relname    |  oid
--------------+-------
 mitabla_pkey | 16400
(1 fila)
```

pg_toast_16394 es usado por TOAST (técnica de almacenamiento de atributos de gran tamaño) que, debido a que el tamaño de un bloque en PostgreSQL por defecto es 8KB cuando necesita guardar datos muy grandes, estos datos se comprimen y se dividen en pequeñas “filas” que se guardan en dicha tabla. Actualmente se encuentra vacía.

```
mibasedatos=# select * from pg_toast.pg_toast_16394;
 chunk_id | chunk_seq | chunk_data
----------+-----------+------------
(0 filas)
```

pg_toast_16394_index es usado para guardar los índices de la tabla anterior y así optimizar la base de datos permitiendo buscar y seleccionar filas mucho más rápido para dicha tabla.

Y, por último, mitabla_pkey que se usa como índice de MiTabla

```
mibasedatos=# \d mitabla
                    Tabla «public.mitabla»
  Columna   |  Tipo   | Ordenamiento | Nulable  | Por omisión
------------+---------+--------------+----------+-------------
 id_cliente | integer |              | not null |
 nombre     | text    |              |          |
 apellidos  | text    |              |          |
 direccion  | text    |              |          |
 puntos     | integer |              |          |
Índices:
    "mitabla_pkey" PRIMARY KEY, btree (id_cliente)
```

```
mibasedatos=# select * from pg_indexes where indexname = 'mitabla_pkey';
-[ RECORD 1 ]----------------------------------------------------------------
---------
schemaname | public
tablename  | mitabla
indexname  | mitabla_pkey
tablespace |
indexdef   | CREATE UNIQUE INDEX mitabla_pkey ON public.mitabla USING btree (id
_cliente)
```

Ha sido creado automáticamente al establecer id_cliente como Primary Key y que, al igual que en pg_toast_16394_index guarda los índices, pero en este caso para la tabla MiTabla.

Cuestión 3. Insertar una tupla en la tabla. ¿Cuánto ocupa la tabla? ¿Se ha producido alguna actualización más? ¿Por qué?

Al insertar una tupla la tabla ocupa 8 KB por su fichero con nombre 16394

```
mibasedatos=# INSERT INTO mitabla VALUES (0, 'nombre0', 'apellidos0', 'direccion0', 0);
INSERT 0 1
postgres@ubuntu:~/12/main$ ls -lh base/16385/16*
-rw------- 1 postgres postgres 8,0K feb  9 14:17 base/16385/16394
-rw------- 1 postgres postgres    0 feb  3 11:52 base/16385/16397
-rw------- 1 postgres postgres 8,0K feb  3 11:52 base/16385/16399
-rw------- 1 postgres postgres  16K feb  9 14:17 base/16385/16400
```

Y al consultar a PostgreSQL con pg_table_size que da el tamaño de la tabla sin los índices, pero incluyendo TOAST (fichero 16399), Free Space Map y Visibility Map.

```
mibasedatos=# SELECT pg_size_pretty(pg_table_size('mitabla'));
 pg_size_pretty
----------------
 16 kB
(1 fila)
```

Y se ha actualizado el fichero 16400 que es mitabla_pkey y se usa como índice de MiTabla al comprobar que ahora ocupa 8 KB más, es decir 16 KB después del insert.

Todo esto es porque PostgreSQL trata los registros en bloques de 8 KB.

Cuestión 4. Aplicar el módulo pg_buffercache a la base de datos **MiBaseDatos.** ¿Es lógico lo que se muestra referido a la base de datos anterior? ¿Por qué?

```
mibasedatos=# create extension pg_buffercache;
CREATE EXTENSION
```

Después de aplicar el módulo pg_buffercache se comprueba que hay nuevas filas para el relfilenode con números 16394 (mitabla) y 16400 (mitabla_pkey) esto es debido a que PostgreSQL usa una caché compartida para mejorar la optimización de la base de datos sin tener que leer/escribir directamente en el disco duro a hacerlo usando esta memoria compartida.

```
mibasedatos=# select * from pg_buffercache where relfilenode = 16394;
 bufferid | relfilenode | reltablespace | reldatabase | relforknumber | relblocknumber | isdirty | usagecount | pinning_backends
----------+-------------+---------------+-------------+---------------+----------------+---------+------------+------------------
      300 |       16394 |          1663 |       16385 |             0 |              0 | f       |          1 |                0
(1 fila)
```

```
mibasedatos=# select * from pg_buffercache where relfilenode = 16400;
 bufferid | relfilenode | reltablespace | reldatabase | relforknumber | relblocknumber | isdirty | usagecount | pinning_backends
----------+-------------+---------------+-------------+---------------+----------------+---------+------------+------------------
      246 |       16400 |          1663 |       16385 |             0 |              0 | f       |          5 |                0
      301 |       16400 |          1663 |       16385 |             0 |              1 | f       |          1 |                0
(2 filas)
```

Cuestión 5. Borrar la tabla **MiTabla** y volverla a crear. Insertar los datos que se entregan en el fichero de texto denominado datos_mitabla.txt. ¿Cuánto ocupa la información original a insertar? ¿Cuánto ocupa la tabla ahora? ¿Por qué? Calcular teóricamente el tamaño en bloques que ocupa la relación **MiTabla** tal y como se realiza en teoría. ¿Concuerda con el tamaño en bloques que nos proporciona PostgreSQL? ¿Por qué?

El fichero original ocupa unos 900MB

```
postgres@ubuntu:~$ ls -lh datos_mitabla.txt
-rw-rw-r-- 1 postgres postgres 900M ene 25 00:15 datos_mitabla.txt
```

Se insertan los 15.000.000 registros

```
mibasedatos=# \COPY mitabla FROM '/var/lib/postgresql/datos_mitabla.txt' DELIMITER ';';
COPY 15000000
```

Después de insertar los datos, la tabla en ficheros (16416 y 16416.1) ocupa aproximadamente 1,2GB

```
postgres@ubuntu:~/12/main$ ls -l base/16385/16*
-rw------- 1 postgres postgres 1073741824 feb 10 00:12 base/16385/16416
-rw------- 1 postgres postgres  236371968 feb 10 00:14 base/16385/16416.1
-rw------- 1 postgres postgres     344064 feb 10 00:14 base/16385/16416_fsm
-rw------- 1 postgres postgres          0 feb 10 00:03 base/16385/16419
-rw------- 1 postgres postgres       8192 feb 10 00:03 base/16385/16421
-rw------- 1 postgres postgres  447406080 feb 10 00:15 base/16385/16422
```

Y PostgreSQL da el mismo valor si se resta el tamaño del FSM (16416_fsm) y el del TOAST (16421)

```
mibasedatos=# SELECT pg_table_size('mitabla');
 pg_table_size
---------------
    1310466048
(1 fila)
```

Está diferencia de tamaño es debida a que PostgreSQL almacena los registros en bloques de 8KB. Si dividimos el tamaño de la tabla entre 8KB y redondeamos hacia arriba se obtiene que hay aproximadamente casi unos 160.000 bloques de 8 KB cada uno. Y si consultamos a pg_class para MiTabla nos da que los números de bloques son exactamente 159.926 bloques.

```
mibasedatos=# select relpages from pg_class where relname = 'mitabla';
 relpages
----------
   159926
(1 fila)
```

Para calcular teóricamente el tamaño en bloques primero se calcula la media de los campos TEXT de MiTabla al tratarse de campos dinámicos.

```
mibasedatos=# select round(avg(pg_column_size(nombre))) from mitabla;
 round
-------
    14
(1 fila)

mibasedatos=# select round(avg(pg_column_size(apellidos))) from mitabla;
 round
-------
    17
(1 fila)

mibasedatos=# select round(avg(pg_column_size(direccion))) from mitabla;
 round
-------
    17
(1 fila)
```

Y a continuación hacemos los cálculos:

Longitud de id_cliente: 4 bytes

Longitud de nombre: 14 bytes

Longitud de apellidos: 17 bytes

Longitud de dirección: 17 bytes

Longitud de puntos: 4 bytes

La longitud del registro será: LR = 4 + 14 + 17 + 17 + 4 = 56 bytes.

Número de registros / bloque = Factor de bloque = 8192 / 56 = 146,2857143 = 146

Número de bloques = 15.000.000 / 146 = 102.740 bloques.

No concuerda con el mismo número de bloques de PostgreSQL seguramente porque no estamos teniendo en cuenta los 24 bytes de la cabecera de cada bloque que PostgreSQL añade y que los campos TEXT al ser dinámicos, y que aquí se calcula su media, puede provocar que sea más complicado dar con el mismo resultado.

Cuestión 6. Volver a aplicar el módulo pg_buffercache a la base de datos **MiBaseDatos**. ¿Qué se puede deducir de lo que se muestra? ¿Por qué lo hará?

Se puede deducir que, por cada nuevo bloque de 8 KB, se crea un nuevo buffer cache. En este caso el relfilenode es el fichero de mitabla y relblocknumber nos dice qué bloque ha sido cacheado de ese fichero. Lo hará así porque de esta forma se puede acceder más rápido a los registros que están en bloques usando la caché que directamente consultado al disco duro.

bufferid	relfilenode	reltablespace	reldatabase	relforknumber	relblocknumber	isdirty	usagecount	pinning_backends
232	16416	1663	16385	0	159805	f	0	0
328	16416	1663	16385	1	0	f	5	0
555	16416	1663	16385	0	159913	f	1	0
624	16416	1663	16385	1	41	f	5	0
969	16416	1663	16385	0	159842	f	0	0
1105	16416	1663	16385	0	159809	f	0	0
1295	16416	1663	16385	0	159921	f	1	0
1554	16416	1663	16385	0	159873	f	0	0
1585	16416	1663	16385	0	159819	f	0	0
1732	16416	1663	16385	0	159888	f	0	0
1798	16416	1663	16385	0	159811	f	0	0
1921	16416	1663	16385	0	159889	f	1	0

Cuestión 7. Aplicar el módulo pgstattuple a la tabla **MiTabla**. ¿Qué se muestra en las estadísticas? ¿Cuál es el grado de ocupación de los bloques? ¿Cuánto espacio libre queda? ¿Por qué?

```
mibasedatos=# CREATE EXTENSION pgstattuple;
CREATE EXTENSION
```

Las estadísticas muestran en orden lo siguiente: el tamaño físico de la tabla en bytes, números de tuplas, el tamaño total de las tuplas en bytes, porcentaje de tuplas, número de tuplas “muertas”, el tamaño total de tuplas “muertas” en bytes, porcentaje de tuplas “muertas”, tamaño total de espacio libre en bytes y el porcentaje de espacio libre.

```
mibasedatos=# SELECT * FROM pgstattuple('mitabla');
-[ RECORD 1 ]------+-----------
table_len          | 1310113792
tuple_count        | 15000000
tuple_len          | 1219555960
tuple_percent      | 93.09
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 5761792
free_percent       | 0.44
```

El grado de ocupación de los bloques es del 93,09% y de espacio libre queda 5.761.792 bytes porque no hay tuplas "muertas", es decir, tuplas que son eliminadas (DELETE) u obsoletas por actualización (UPDATE) y que no son eliminadas físicamente de su tabla y se mantienen dentro del servidor PostgreSQL.

Cuestión 8 ¿Cuál es el factor de bloque medio real de la tabla? Realizar una consulta SQL que obtenga ese valor y comparar con el factor de bloque teórico siguiendo el procedimiento visto en teoría.

```
mibasedatos=# select relname, relpages, reltuples, reltuples/relpages avgtuple
from pg_class where relname = 'mitabla';
 relname | relpages |  reltuples   |     avgtuple
---------+----------+--------------+------------------
 mitabla |   159926 | 1.500019e+07 | 93.79456748746296
(1 fila)
```

El factor de bloque medio real de la tabla es de 93. Al compararlo con el factor de bloque teórico es diferente y seguramente se deba a que no se tiene en cuenta toda la longitud del registro y la cabecera que añade PostgreSQL a cada bloque.

La longitud del registro será: LR = 4 + 14 + 17 + 17 + 4 = 56 bytes.

Número de registros / bloque = Factor de bloque = 8192 / 56 = 146,2857143 = 146

Cuestión 9 Con el módulo pageinspect, analizar la cabecera y elementos de la página del primer bloque, del bloque situado en la mitad del archivo y el último bloque de la tabla **MiTabla**. ¿Qué diferencias se aprecian entre ellos? ¿Por qué?

```
mibasedatos=# CREATE EXTENSION pageinspect;
CREATE EXTENSION
```

```
mibasedatos=# select * from page_header(get_raw_page('mitabla', 0));
    lsn    | checksum | flags | lower | upper | special | pagesize | version | prune_xid
-----------+----------+-------+-------+-------+---------+----------+---------+-----------
 0/16F4640 |        0 |     0 |   400 |   448 |    8192 |     8192 |       4 |         0
(1 fila)

mibasedatos=# select * from page_header(get_raw_page('mitabla', 79962));
    lsn     | checksum | flags | lower | upper | special | pagesize | version | prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+-----------
 0/4735E3C8 |        0 |     0 |   400 |   448 |    8192 |     8192 |       4 |         0
(1 fila)

mibasedatos=# select * from page_header(get_raw_page('mitabla', 159925));
    lsn     | checksum | flags | lower | upper | special | pagesize | version | prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+-----------
 0/A19FEB28 |        0 |     0 |   128 |  5984 |    8192 |     8192 |       4 |         0
(1 fila)
```

La diferencia más notable es en el número de filas que tienen el primer bloque y mitad al último bloque. Como PostgreSQL utiliza bloques como memoria contigua para evitar fragmentación en la memoria y tener un mejor rendimiento, el último bloque de la tabla todavía puede ser rellenado con más registros.

```
mibasedatos=# select count(*) from heap_page_items(get_raw_page('mitabla', 0));
 count
-------
    94
(1 fila)

mibasedatos=# select count(*) from heap_page_items(get_raw_page('mitabla', 79962));
 count
-------
    94
(1 fila)

mibasedatos=# select count(*) from heap_page_items(get_raw_page('mitabla', 159925));
 count
-------
    26
(1 fila)
```

Cuestión 10. Crear un índice de tipo árbol para el campo puntos. ¿Dónde se almacena físicamente ese índice? ¿Qué tamaño tiene? ¿Cuántos bloques tiene? ¿Cuántos niveles tiene? ¿Cuántos bloques tiene por nivel? ¿Cuántas tuplas tiene un bloque de cada nivel?

```
mibasedatos=# CREATE INDEX puntos_idx ON mitabla USING btree(puntos);
CREATE INDEX
```

Se almacena físicamente en la carpeta con nombre del oid de mibasedatos y el fichero del índice se puede saber por su oid consultando a pg_class. El fichero tiene un tamaño físicamente de 322MB.

```
mibasedatos=# select relname,oid from pg_class where relname = 'puntos_idx';
  relname   |  oid
------------+-------
 puntos_idx | 16462
(1 fila)
```

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16462
-rw------- 1 postgres postgres 322M feb 22 13:35 base/16385/16462
```

Tiene 41.188 bloques en total si se consulta también a pg_class:

```
mibasedatos=# select relpages from pg_class where relname = 'puntos_idx';
 relpages
----------
    41188
(1 fila)
```

Aunque pgstatindex marca que tiene 2 niveles no cuenta la raíz, por lo tanto tiene 3 niveles. Hay 203 bloques de los niveles intermedios más los 40984 bloques en las hojas y más 1 bloque de raíz, hacen los 41.188 bloques totales.

```
mibasedatos=# select * from pgstatindex('puntos_idx');
-[ RECORD 1 ]------+----------
version            | 4
tree_level         | 2
index_size         | 337412096
root_block_no      | 209
internal_pages     | 203
leaf_pages         | 40984
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 90.19
leaf_fragmentation | 0
```

En general hay 367 tuplas en un bloque, excepto para el bloque raíz que son 202 tuplas y el último bloque tiene 222 tuplas.

```
mibasedatos=# select * from bt_page_stats('puntos_idx', 1);
-[ RECORD 1 ]-+-----
blkno         | 1
type          | l
live_items    | 367
dead_items    | 0
avg_item_size | 16
page_size     | 8192
free_size     | 800
btpo_prev     | 0
btpo_next     | 2
btpo          | 0
btpo_flags    | 1
```

```
mibasedatos=# select * from bt_page_stats('puntos_idx', 209);
-[ RECORD 1 ]-+-----
blkno         | 209
type          | r
live_items    | 202
dead_items    | 0
avg_item_size | 23
page_size     | 8192
free_size     | 2508
btpo_prev     | 0
btpo_next     | 0
btpo          | 2
btpo_flags    | 2
```

```
mibasedatos=# select * from bt_page_stats('puntos_idx', 41187);
-[ RECORD 1 ]-+------
blkno         | 41187
type          | l
live_items    | 222
dead_items    | 0
avg_item_size | 16
page_size     | 8192
free_size     | 3708
btpo_prev     | 41186
btpo_next     | 0
btpo          | 0
btpo_flags    | 1
```

Cuestión 11. Determinar el tamaño de bloques que teóricamente tendría de acuerdo con lo visto en teoría y el número de niveles. Comparar los resultados obtenidos teóricamente con los resultados obtenidos en la cuestión 10.

Para calcular el nodo intermedio en un árbol B se utiliza la siguiente fórmula:

$m * LPB + (m-1) * (LK + LPB) \leq B$

Longitud del puntero a bloque: 4 bytes

Longitud del registro puntos: 4 bytes

Tamaño del bloque: 8192 bytes

$m * LPB + (m-1) * (LK + LPR) \leq B$

$m * 4 + (m-1) * (4 + 6) \leq 8192$

$4m + 10m - 10 \leq 8192$

$14m - 10 \leq 8192$

14m ≤ 8202

m ≤ 585 punteros a nodo

Raíz: 1 nodo, 584 entradas, 585 punteros a nodos

Nodo intermedio: 585 nodos, 341640 entradas, 342225 punteros a nodos

Hojas: 342225 nodos

1 + 585 + 341640 = 348054 bloques promedio y hay 3 niveles

En los resultados obtenidos de la cuestión 10, en general los bloques tienen 367 entradas que suelen ocupar 16 bytes de media cada uno pero no llegan a llenar el bloque de 8192 bytes, haría falta unas 8192/16 = 512 entradas de media para llenar 1 bloque.

En el apartado teórico hemos calculado una media de cuántos registros por bloque podría haber para llenar los 8192 bytes que son de unas 584 entradas, que se aproxima algo a las 512 entradas de la cuestión 10. A partir de ahí salen más bloques de lo normal porque se tiene en cuenta que todos los bloques se van llenando, cuando en realidad hay bloques con espacios libre para más entradas.

Cuestión 12. Crear un índice de tipo hash para el campo id_cliente y otro para el campo puntos.

```
mibasedatos=# CREATE INDEX id_cliente_idx ON mitabla USING hash(id_cliente);
CREATE INDEX
mibasedatos=# CREATE INDEX puntos_idxh ON mitabla USING hash(puntos);
CREATE INDEX
```

Cuestión 13. A la vista de los resultados obtenidos de aplicar los módulos pgstattuple y pageinspect, ¿Qué conclusiones se puede obtener de los dos índices hash que se han creado? ¿Por qué?

La diferencia más notable es que puntos_idxh tiene más cajones de desbordamiento (overflow pages) que id_cliente_idx.

```
mibasedatos=# select * from pgstathashindex('id_cliente_idx');
-[ RECORD 1 ]--+------------------
version        | 4
bucket_pages   | 49152
overflow_pages | 16237
bitmap_pages   | 1
unused_pages   | 0
live_items     | 15000000
dead_items     | 0
free_percent   | 43.72022039882073

mibasedatos=# select * from pgstathashindex('puntos_idxh');
-[ RECORD 1 ]--+------------------
version        | 4
bucket_pages   | 49152
overflow_pages | 36470
bitmap_pages   | 2
unused_pages   | 0
live_items     | 15000000
dead_items     | 0
free_percent   | 57.019475037472716
```

Por ello también resulta más difícil encontrar cajones llenos en puntos_idxh, sobre todo en los primeros cajones hash.

```
mibasedatos=# select * from hash_page_stats(get_raw_page('id_cliente_idx', 1));
-[ RECORD 1 ]---+-----------
live_items      | 245
dead_items      | 0
page_size       | 8192
free_size       | 3248
hasho_prevblkno | 49151
hasho_nextblkno | 4294967295
hasho_bucket    | 0
hasho_flag      | 2
hasho_page_id   | 65408

mibasedatos=# select * from hash_page_stats(get_raw_page('puntos_idxh', 1));
-[ RECORD 1 ]---+-----------
live_items      | 0
dead_items      | 0
page_size       | 8192
free_size       | 8148
hasho_prevblkno | 49151
hasho_nextblkno | 4294967295
hasho_bucket    | 0
hasho_flag      | 2
hasho_page_id   | 65408
```

```
mibasedatos=# select count(*) from hash_page_items(get_raw_page('id_cliente_idx', 1));
-[ RECORD 1 ]
count | 245

mibasedatos=# select count(*) from hash_page_items(get_raw_page('puntos_idxh', 1));
-[ RECORD 1 ]
count | 0

mibasedatos=# select count(*) from hash_page_items(get_raw_page('puntos_idxh', 5));
-[ RECORD 1 ]
count | 0

mibasedatos=# select count(*) from hash_page_items(get_raw_page('puntos_idxh', 555));
-[ RECORD 1 ]
count | 0

mibasedatos=# select count(*) from hash_page_items(get_raw_page('puntos_idxh', 5333));
-[ RECORD 1 ]
count | 407

mibasedatos=# select count(*) from hash_page_items(get_raw_page('puntos_idxh', 53334));
-[ RECORD 1 ]
count | 407
```

Esto es debido a que la función de asociación está teniendo que meter varios registros en el mismo cajón provocando que estos se llenen y teniendo que crear unos nuevos que son los de desbordamiento.

Cuestión 14. Realice las pruebas que considere de inserción, modificación y borrado para determinar el manejo que realiza PostgreSQL internamente con los registros de datos y las estructuras de los archivos que utiliza. Comentar las conclusiones obtenidas.

Para realizar las siguientes pruebas primero se ha reiniciado el sistema y se ha consultado que pg_buffercache se encuentra vacío para MiTabla (oid 16416)

```
mibasedatos=# select * from pg_buffercache where relfilenode = 16416;
(0 filas)
```

Se inserta una tupla cualquiera.

```
mibasedatos=# INSERT INTO mitabla VALUES (0, 'nombre0', 'apellidos0', 'direccion0', 0);
INSERT 0 1
```

Se vuelve a consultar a pg_buffercache y se sabe que ha creado varios buffers

```
mibasedatos=# select * from pg_buffercache where relfilenode = 16416;
-[ RECORD 1 ]----+-------
bufferid         | 268
relfilenode      | 16416
reltablespace    | 1663
reldatabase      | 16385
relforknumber    | 1
relblocknumber   | 0
isdirty          | f
usagecount       | 1
pinning_backends | 0
-[ RECORD 2 ]----+-------
bufferid         | 269
relfilenode      | 16416
reltablespace    | 1663
reldatabase      | 16385
relforknumber    | 0
relblocknumber   | 159925
isdirty          | t
usagecount       | 2
pinning_backends | 0
-[ RECORD 3 ]----+-------
bufferid         | 283
relfilenode      | 16416
reltablespace    | 1663
reldatabase      | 16385
relforknumber    | 0
relblocknumber   | 13347
isdirty          | f
usagecount       | 1
pinning_backends | 0
```

y que los archivos no han sido modificados al comparar la fecha

```
postgres@ubuntu:~/12/main$ date
sáb feb 29 01:48:19 PST 2020
postgres@ubuntu:~/12/main$ ls -lh base/16385/16416*
-rw------- 1 postgres postgres 1,0G feb 29 01:35 base/16385/16416
-rw------- 1 postgres postgres 226M feb 29 01:40 base/16385/16416.1
-rw------- 1 postgres postgres 336K feb 10 00:14 base/16385/16416_fsm
```

Se actualiza la misma tupla que se ha insertado

```
mibasedatos=# UPDATE mitabla SET nombre = 'nombre1' WHERE id_cliente = 0;
UPDATE 1
```

Y se consulta pg_buffercache de nuevo, el cual mantiene los mismos registros pero en uno de los buffers ha incrementado el usagecount.

```
mibasedatos=# select * from pg_buffercache where relfilenode = 16416;
-[ RECORD 1 ]----+-------
bufferid         | 268
relfilenode      | 16416
reltablespace    | 1663
reldatabase      | 16385
relforknumber    | 1
relblocknumber   | 0
isdirty          | f
usagecount       | 1
pinning_backends | 0
-[ RECORD 2 ]----+-------
bufferid         | 269
relfilenode      | 16416
reltablespace    | 1663
reldatabase      | 16385
relforknumber    | 0
relblocknumber   | 159925
isdirty          | t
usagecount       | 3
pinning_backends | 0
-[ RECORD 3 ]----+-------
bufferid         | 283
relfilenode      | 16416
reltablespace    | 1663
reldatabase      | 16385
relforknumber    | 0
relblocknumber   | 13347
isdirty          | f
usagecount       | 1
pinning_backends | 0
```

Y los archivos se mantienen igual.

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16416*
-rw------- 1 postgres postgres 1,0G feb 29 01:35 base/16385/16416
-rw------- 1 postgres postgres 226M feb 29 01:40 base/16385/16416.1
-rw------- 1 postgres postgres 336K feb 10 00:14 base/16385/16416_fsm
```

Se elimina la tupla anterior

```
mibasedatos=# DELETE FROM mitabla where id_cliente = 0;
DELETE 1
```

Y pg_buffercache hace lo mismo, vuelve a incrementar en 1 el usagecount de uno de los buffers

```
mibasedatos=# select * from pg_buffercache where relfilenode = 16416;
-[ RECORD 1 ]----+-------
bufferid         | 268
relfilenode      | 16416
reltablespace    | 1663
reldatabase      | 16385
relforknumber    | 1
relblocknumber   | 0
isdirty          | f
usagecount       | 1
pinning_backends | 0
-[ RECORD 2 ]----+-------
bufferid         | 269
relfilenode      | 16416
reltablespace    | 1663
reldatabase      | 16385
relforknumber    | 0
relblocknumber   | 159925
isdirty          | t
usagecount       | 4
pinning_backends | 0
-[ RECORD 3 ]----+-------
bufferid         | 283
relfilenode      | 16416
reltablespace    | 1663
reldatabase      | 16385
relforknumber    | 0
relblocknumber   | 13347
isdirty          | f
usagecount       | 1
pinning_backends | 0
```

Y los ficheros se mantienen igual.

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16416*
-rw------- 1 postgres postgres 1,0G feb 29 01:35 base/16385/16416
-rw------- 1 postgres postgres 226M feb 29 01:40 base/16385/16416.1
-rw------- 1 postgres postgres 336K feb 10 00:14 base/16385/16416_fsm
```

En conclusión, lo que realiza PostgreSQL internamente con los registros de datos es guardarlos en el buffercache y no modifica ninguna de las estructuras de los archivos de MiTabla.

Cuestión 15. Borrar 2.000.000 de tuplas de la tabla **MiTabla** de manera aleatoria usando el valor del campo id_cliente. ¿Qué es lo que ocurre físicamente en la base de datos? ¿Se observa algún cambio en el tamaño de la tabla y de los índices? ¿Por qué? Adjuntar el código de borrado.

Antes de hacer el borrado se mira el tamaño físico y de bloques de la tabla y de los índices para luego comparar

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16416*
-rw------- 1 postgres postgres 1,0G feb 29 01:35 base/16385/16416
-rw------- 1 postgres postgres 226M feb 29 01:40 base/16385/16416.1
-rw------- 1 postgres postgres 336K feb 10 00:14 base/16385/16416_fsm
postgres@ubuntu:~/12/main$ ls -lh base/16385/16463*
-rw------- 1 postgres postgres 511M feb 29 01:51 base/16385/16463
postgres@ubuntu:~/12/main$ ls -lh base/16385/16463
-rw------- 1 postgres postgres 511M feb 29 01:51 base/16385/16463
postgres@ubuntu:~/12/main$ ls -lh base/16385/16462
-rw------- 1 postgres postgres 322M feb 29 01:51 base/16385/16462
postgres@ubuntu:~/12/main$ ls -lh base/16385/16464
-rw------- 1 postgres postgres 669M feb 29 01:51 base/16385/16464
```

```
mibasedatos=# select relpages from pg_class where relname = 'puntos_idxh';
-[ RECORD 1 ]---
relpages | 85625

mibasedatos=# select relpages from pg_class where relname = 'mitabla';
-[ RECORD 1 ]----
relpages | 159926

mibasedatos=# select relpages from pg_class where relname = 'puntos_idx';
-[ RECORD 1 ]---
relpages | 41188

mibasedatos=# select relpages from pg_class where relname = 'id_cliente_idx';
-[ RECORD 1 ]---
relpages | 65391
```

Se borran 2.000.000 de tuplas de manera aleatoria

```
mibasedatos=# DELETE FROM mitabla WHERE id_cliente in (select id_cliente from mitabla order by random() limit 2000000);
DELETE 2000000
```

No se observa ningún cambio físicamente ni a las tablas ni a los índices. Tienen el mismo tamaño físico y el mismo número de bloques.

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16416*
-rw------- 1 postgres postgres 1,0G feb 29 02:23 base/16385/16416
-rw------- 1 postgres postgres 226M feb 29 02:23 base/16385/16416.1
-rw------- 1 postgres postgres 336K feb 10 00:14 base/16385/16416_fsm
postgres@ubuntu:~/12/main$ ls -lh base/16385/16464
-rw------- 1 postgres postgres 669M feb 29 01:51 base/16385/16464
postgres@ubuntu:~/12/main$ ls -lh base/16385/16463
-rw------- 1 postgres postgres 511M feb 29 01:51 base/16385/16463
postgres@ubuntu:~/12/main$ ls -lh base/16385/16462
-rw------- 1 postgres postgres 322M feb 29 01:51 base/16385/16462
```

```
mibasedatos=# select relpages from pg_class where relname = 'mitabla';
-[ RECORD 1 ]----
relpages | 159926

mibasedatos=# select relpages from pg_class where relname = 'puntos_idxh';
-[ RECORD 1 ]---
relpages | 85625

mibasedatos=# select relpages from pg_class where relname = 'puntos_idx';
-[ RECORD 1 ]---
relpages | 41188

mibasedatos=# select relpages from pg_class where relname = 'id_cliente_idx';
-[ RECORD 1 ]---
relpages | 65391
```

Esto es porque en PostgreSQL cuando se hace un DELETE las tuplas no se eliminan al instante mientras que puedan ser visibles a otras transacciones. Aunque cuando esas tuplas eliminadas ya no contemplan un interés hay que reclamar el espacio usado por estas para que lo puedan usar unas nuevas tuplas.

Cuestión 16. En la situación anterior, ¿Qué operaciones se puede aplicar a la base de datos **MiBaseDatos** para optimizar el rendimiento de esta? Aplicarla a la base de datos **MiBaseDatos** y comentar cuál es el resultado final y qué es lo que ocurre físicamente.

Para reclamar el espacio usado por las filas eliminadas que se ha dado en la situación anterior y así optimizar el rendimiento de la base de datos hay que ejecutar VACUUM.

```
mibasedatos=# VACUUM (VERBOSE, ANALYZE) mitabla;
INFO:  haciendo vacuum a «public.mitabla»
INFO:  se recorrió el índice «mitabla_pkey» para eliminar 2000003 versiones de filas
DETALLE:  CPU: usuario: 7.61 s, sistema: 2.85 s, transcurrido: 16.10 s
INFO:  se recorrió el índice «puntos_idx» para eliminar 2000003 versiones de filas
DETALLE:  CPU: usuario: 2.85 s, sistema: 2.82 s, transcurrido: 20.50 s
INFO:  se recorrió el índice «id_cliente_idx» para eliminar 2000003 versiones de filas
DETALLE:  CPU: usuario: 6.26 s, sistema: 10.67 s, transcurrido: 23.51 s
INFO:  se recorrió el índice «puntos_idxh» para eliminar 2000003 versiones de filas
DETALLE:  CPU: usuario: 6.37 s, sistema: 24.08 s, transcurrido: 38.40 s
INFO:  «mitabla»: se eliminaron 2000003 versiones de filas en 159926 páginas
DETALLE:  CPU: usuario: 1.77 s, sistema: 10.14 s, transcurrido: 30.09 s
INFO:  el índice «mitabla_pkey» ahora contiene 12999999 versiones de filas en 54615 páginas
DETALLE:  2000003 versiones de filas del índice fueron eliminadas.
0 páginas de índice han sido eliminadas, 0 son reusables.
CPU: usuario: 0.00 s, sistema: 0.00 s, transcurrido: 0.00 s.
INFO:  el índice «puntos_idx» ahora contiene 12999999 versiones de filas en 41188 páginas
DETALLE:  2000003 versiones de filas del índice fueron eliminadas.
0 páginas de índice han sido eliminadas, 0 son reusables.
CPU: usuario: 0.00 s, sistema: 0.00 s, transcurrido: 0.00 s.
INFO:  el índice «id_cliente_idx» ahora contiene 12999999 versiones de filas en 65391 páginas
DETALLE:  2000002 versiones de filas del índice fueron eliminadas.
0 páginas de índice han sido eliminadas, 0 son reusables.
CPU: usuario: 0.00 s, sistema: 0.00 s, transcurrido: 0.00 s.
INFO:  el índice «puntos_idxh» ahora contiene 12999999 versiones de filas en 85625 páginas
DETALLE:  2000003 versiones de filas del índice fueron eliminadas.
0 páginas de índice han sido eliminadas, 0 son reusables.
CPU: usuario: 0.00 s, sistema: 0.00 s, transcurrido: 0.00 s.
INFO:  «mitabla»: se encontraron 2000004 versiones de filas eliminables y 12999999 no eliminables en 159926 de 159926 páginas
DETALLE:  0 versiones muertas de filas no pueden ser eliminadas aún, xmin máx antiguo: 530
Hubo 2 identificadores de ítem sin usar.
Omitiendo 0 páginas debido a «pins» de página, 0 páginas marcadas «frozen».
0 páginas están completamente vacías.
CPU: usuario: 25.80 s, sistema: 55.14 s, transcurrido: 152.60 s.
INFO:  haciendo vacuum a «pg_toast.pg_toast_16416»
INFO:  el índice «pg_toast_16416_index» ahora contiene 0 versiones de filas en 1 páginas
DETALLE:  0 versiones de filas del índice fueron eliminadas.
0 páginas de índice han sido eliminadas, 0 son reusables.
CPU: usuario: 0.00 s, sistema: 0.00 s, transcurrido: 0.07 s.
INFO:  «pg_toast_16416»: se encontraron 0 versiones de filas eliminables y 0 no eliminables en 0 de 0 páginas
DETALLE:  0 versiones muertas de filas no pueden ser eliminadas aún, xmin máx antiguo: 530
Hubo 0 identificadores de ítem sin usar.
Omitiendo 0 páginas debido a «pins» de página, 0 páginas marcadas «frozen».
0 páginas están completamente vacías.
```

```
CPU: usuario: 0.00 s, sistema: 0.00 s, transcurrido: 0.07 s.
INFO:  analizando «public.mitabla»
INFO:  «mitabla»: se procesaron 30000 de 159926 páginas, que contenían 2439016 filas vigentes y 0 filas no vigentes; 30000 filas en la muestra, 13002069 total de filas estimadas
VACUUM
```

El resultado final es que esas tuplas eliminadas al hacer VACUUM han sido marcadas como espacio disponible para futuros usos, aunque físicamente el espacio es el mismo que antes como se muestra a continuación. Además, se ha creado un mapa de visibilidad para MiTabla.

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16416*
-rw------- 1 postgres postgres 1,0G feb 29 02:50 base/16385/16416
-rw------- 1 postgres postgres 226M feb 29 02:51 base/16385/16416.1
-rw------- 1 postgres postgres 336K feb 29 02:51 base/16385/16416_fsm
-rw------- 1 postgres postgres  40K feb 29 02:51 base/16385/16416_vm
postgres@ubuntu:~/12/main$ ls -lh base/16385/16464
-rw------- 1 postgres postgres 669M feb 29 02:43 base/16385/16464
postgres@ubuntu:~/12/main$ ls -lh base/16385/16463
-rw------- 1 postgres postgres 511M feb 29 02:43 base/16385/16463
postgres@ubuntu:~/12/main$ ls -lh base/16385/16462
-rw------- 1 postgres postgres 322M feb 29 02:43 base/16385/16462
```

En cambio, si hacemos un VACUUM FULL

```
mibasedatos=# VACUUM (FULL, VERBOSE, ANALYZE) mitabla;
INFO:  haciendo vacuum a «public.mitabla»
INFO:  «mitabla»: se encontraron 0 versiones eliminables de filas y 12999999 no eliminables en 159926 páginas
DETALLE:  0 versiones muertas de filas no pueden ser eliminadas aún.
CPU: usuario: 13.05 s, sistema: 79.62 s, transcurrido: 103.90 s.
INFO:  analizando «public.mitabla»
INFO:  «mitabla»: se procesaron 30000 de 138601 páginas, que contenían 2813779 filas vigentes y 0 filas no vigentes; 30000 filas en la muestra, 12999753 total de filas estimadas
VACUUM
```

Mueve mitabla a un nuevo fichero

```
mibasedatos=# select relfilenode from pg_class where relname = 'mitabla';
-[ RECORD 1 ]------
relfilenode | 16537
```

El cual tiene un tamaño menor y deja el anterior a 0 bytes, por lo tanto también se podría recuperar el espacio que usaban las tuplas eliminadas y hacer que la tabla ocupe menos si es necesario.

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16416*
-rw------- 1 postgres postgres 0 feb 29 03:21 base/16385/16416
postgres@ubuntu:~/12/main$ ls -lh base/16385/16537*
-rw------- 1 postgres postgres 1,0G feb 29 03:18 base/16385/16537
-rw------- 1 postgres postgres  59M feb 29 03:18 base/16385/16537.1
```

Y lo mismo ocurre con los índices.

```
mibasedatos=# select relfilenode from pg_class where relname = 'puntos_idx';
-[ RECORD 1 ]------
relfilenode | 16544

mibasedatos=# select relfilenode from pg_class where relname = 'puntos_idxh';
-[ RECORD 1 ]------
relfilenode | 16546

mibasedatos=# select relfilenode from pg_class where relname = 'id_cliente_idx';
-[ RECORD 1 ]------
relfilenode | 16545
```

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16544
-rw------- 1 postgres postgres 279M feb 29 03:19 base/16385/16544
postgres@ubuntu:~/12/main$ ls -lh base/16385/16545
-rw------- 1 postgres postgres 422M feb 29 03:20 base/16385/16545
postgres@ubuntu:~/12/main$ ls -lh base/16385/16546
-rw------- 1 postgres postgres 631M feb 29 03:24 base/16385/16546
```

Cuestión 17. Crear una tabla denominada **MiTabla2** de tal manera que tenga un factor de llenado de tuplas que sea un 40% que el de la tabla **MiTabla** y cargar el archivo de datos anterior Explicar el proceso seguido y qué es lo que ocurre físicamente.

Se crea la tabla MiTabla2 usando filfactor para que tenga un factor de llenado del 40%:

```
mibasedatos=# CREATE TABLE MiTabla2(id_cliente integer PRIMARY KEY, nombre TEXT
, apellidos TEXT, direccion TEXT, puntos INTEGER) WITH (fillfactor=40);
CREATE TABLE
```

Cuyo oid es 16465 y se encuentra el fichero vacío:

```
mibasedatos=# select oid, reloptions from pg_class where relname = 'mitabla2';
-[ RECORD 1 ]---------------
oid        | 16465
reloptions | {fillfactor=40}
```

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16465
-rw------- 1 postgres postgres 0 feb 28 01:31 base/16385/16465
```

Se carga el archivo de datos anterior:

```
mibasedatos=# \COPY mitabla2 FROM '/var/lib/postgresql/datos_mitabla.txt' DELIMITER ';';
COPY 15000000
```

Y ahora se han creado 4 ficheros para MiTabla2, hay 3 ficheros que ocupan 1GB y el último con 94MB

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16465*
-rw------- 1 postgres postgres 1,0G feb 28 01:49 base/16385/16465
-rw------- 1 postgres postgres 1,0G feb 28 01:49 base/16385/16465.1
-rw------- 1 postgres postgres 1,0G feb 28 01:49 base/16385/16465.2
-rw------- 1 postgres postgres  94M feb 28 01:51 base/16385/16465.3
```

Esto es debido que al tener un factor de llenado del 40%, ahora el factor de bloque medio real de la tabla es de 37 registros por bloque y por lo tanto habrá más bloques que en este caso son 405.225 bloques que hacen que físicamente los ficheros ocupan muchos más.

```
                                                                                ^
mibasedatos=# select relname, relpages, reltuples, reltuples/relpages avgtuple from pg_class where relname = 'mitabla2'
;
-[ RECORD 1 ]---------------
relname   | mitabla2
relpages  | 405225
reltuples | 1.4999768e+07
avgtuple  | 37.01589980874822
```

Cuestión 18. Realizar las mismas pruebas que la cuestión 14 en la tabla **MiTabla2**. Comparar los resultados obtenidos con los de la cuestión 14 y explicar las diferencias encontradas.

Como en la cuestión 14 se ha reiniciado el sistema y se ha consultado que pg_buffercache se encuentra vacío para MiTabla2 (oid 16465)

```
mibasedatos=# select oid from pg_class where relname = 'mitabla2';
  oid
-------
 16465
(1 fila)
```

```
mibasedatos=# select * from pg_buffercache where relfilenode = 16465;
(0 filas)
```

Y los ficheros antes de realizar las pruebas para poder comparar.

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16465*
-rw------- 1 postgres postgres 1,0G feb 29 13:02 base/16385/16465
-rw------- 1 postgres postgres 1,0G feb 29 13:02 base/16385/16465.1
-rw------- 1 postgres postgres 1,0G feb 28 01:49 base/16385/16465.2
-rw------- 1 postgres postgres  94M feb 29 13:14 base/16385/16465.3
-rw------- 1 postgres postgres 816K feb 28 01:51 base/16385/16465_fsm
```

Se inserta una tupla cualquiera y se vuelve a consultar a pg_buffercache para saber que ha creado varios buffers, en este caso 2 buffers a diferencia de la cuestión 14 que fueron 3 buffers.

```
mibasedatos=# INSERT INTO mitabla2 VALUES (15000000, 'nombre0', 'apellidos0', 'direccion0', 0);
INSERT 0 1
mibasedatos=# select * from pg_buffercache where relfilenode = 16465;
-[ RECORD 1 ]----+-------
bufferid         | 326
relfilenode      | 16465
reltablespace    | 1663
reldatabase      | 16385
relforknumber    | 1
relblocknumber   | 0
isdirty          | f
usagecount       | 1
pinning_backends | 0
-[ RECORD 2 ]----+-------
bufferid         | 327
relfilenode      | 16465
reltablespace    | 1663
reldatabase      | 16385
relforknumber    | 0
relblocknumber   | 405224
isdirty          | t
usagecount       | 2
pinning_backends | 0
```

Los ficheros se mantienen igual que antes y como pasaba en la cuestión 14.

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16465*
-rw------- 1 postgres postgres 1,0G feb 29 13:02 base/16385/16465
-rw------- 1 postgres postgres 1,0G feb 29 13:02 base/16385/16465.1
-rw------- 1 postgres postgres 1,0G feb 28 01:49 base/16385/16465.2
-rw------- 1 postgres postgres  94M feb 29 13:14 base/16385/16465.3
-rw------- 1 postgres postgres 816K feb 28 01:51 base/16385/16465_fsm
```

Se actualiza la misma tupla que se ha insertado y se consulta pg_buffercache de nuevo, el cual mantiene los mismos registros pero en uno de los buffers ha incrementado el usagecount como ocurría en la cuestión 14.

```
mibasedatos=# UPDATE mitabla2 SET nombre = 'nombre1' WHERE id_cliente = 15000000;
UPDATE 1
mibasedatos=# select * from pg_buffercache where relfilenode = 16465;
-[ RECORD 1 ]----+-------
bufferid         | 326
relfilenode      | 16465
reltablespace    | 1663
reldatabase      | 16385
relforknumber    | 1
relblocknumber   | 0
isdirty          | f
usagecount       | 1
pinning_backends | 0
-[ RECORD 2 ]----+-------
bufferid         | 327
relfilenode      | 16465
reltablespace    | 1663
reldatabase      | 16385
relforknumber    | 0
relblocknumber   | 405224
isdirty          | t
usagecount       | 3
pinning_backends | 0
```

Los ficheros siguen igual.

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16465*
-rw------- 1 postgres postgres 1,0G feb 29 13:02 base/16385/16465
-rw------- 1 postgres postgres 1,0G feb 29 13:02 base/16385/16465.1
-rw------- 1 postgres postgres 1,0G feb 28 01:49 base/16385/16465.2
-rw------- 1 postgres postgres  94M feb 29 13:14 base/16385/16465.3
-rw------- 1 postgres postgres 816K feb 28 01:51 base/16385/16465_fsm
```

Se elimina la misma tupla y se mira en pg_buffercache. Incrementa en 1 el usagecount.

```
mibasedatos=# DELETE FROM mitabla2 where id_cliente = 15000000;
DELETE 1
mibasedatos=# select * from pg_buffercache where relfilenode = 16465;
-[ RECORD 1 ]----+-------
bufferid         | 326
relfilenode      | 16465
reltablespace    | 1663
reldatabase      | 16385
relforknumber    | 1
relblocknumber   | 0
isdirty          | f
usagecount       | 1
pinning_backends | 0
-[ RECORD 2 ]----+-------
bufferid         | 327
relfilenode      | 16465
reltablespace    | 1663
reldatabase      | 16385
relforknumber    | 0
relblocknumber   | 405224
isdirty          | t
usagecount       | 4
pinning_backends | 0
```

Los ficheros se mantienen.

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16465*
-rw------- 1 postgres postgres 1,0G feb 29 13:02 base/16385/16465
-rw------- 1 postgres postgres 1,0G feb 29 13:02 base/16385/16465.1
-rw------- 1 postgres postgres 1,0G feb 28 01:49 base/16385/16465.2
-rw------- 1 postgres postgres  94M feb 29 13:14 base/16385/16465.3
-rw------- 1 postgres postgres 816K feb 28 01:51 base/16385/16465_fsm
```

Comparando con la cuestión 14, tenemos que el buffercache en este caso trabaja con dos buffers, el primero para el bloque 0 y el otro para el bloque 405224, este último es mayor que a los bloques que accede la cuestión 14 que son el bloque 0, el 159925 y el 13347. Esto es debido a que como el fillfactor es del 40% ahora hay muchos más bloques, por lo tanto tendrá que acceder al último que es 405224.

Cuestión 19. Las versiones 11 y 12 de PostgreSQL permite trabajar con particionamiento de tablas. ¿Para qué sirve? ¿Qué tipos de particionamientos se pueden utilizar? ¿Cuándo será útil el particionamiento?

Las particiones de tabla permiten dividir una tabla grande en varias tablas pequeñas que sirven para reducir la cantidad de datos a recorrer y aumentar el rendimiento de la base de datos.

Tipos de particionamientos que se pueden utilizar:

- Range: La tabla se divide en rangos (como por ejemplo, en rangos de fechas) que suelen estar definidas por una columna clave o un conjunto de columnas.
- List: Se divide con el uso de una lista que contiene los valores clave que aparecen en cada partición.

- Hash: Se especifica una función Hash con un módulo y un resto para cada partición que tendrá las filas para las cuales el valor hash de la clave de partición dividido por el módulo producirá el resto.

El particionamiento será útil para:
- Realizar consultas puede mejorar el rendimiento aunque depende de qué tipo de consulta. Si las tuplas a recuperar se encuentran en una única partición o en algunas de ellas pero no todas, entonces el particionamiento puede ser útil ya que aprovecha en hacer una lectura secuencial de esa partición en lugar de usar un índice.
- El particionamiento puede sustituir las columnas a indexar provocando que el tamaño de índice sea reducido y que cuando haya que cargarlo a memoria quepan todas las partes más utilizadas.
- Borrar muchas tuplas a la vez como se realizó en la cuestión 15 (aunque no de forma aleatoria) con DELETE y que luego requería VACUUM se puede evitar si el particionamiento ha sido bien diseñado y, en cambio, usar un DROP TABLE para una partición es mucho más rápido que eliminar varias tuplas a la vez.

Cuestión 20. Crear una nueva tabla denominada **MiTabla3** con los mismos campos que la cuestión 2, pero sin PRIMARY KEY, que esté particionada por medio de una función HASH que devuelva 10 valores sobre el campo puntos. Explicar el proceso seguido y comentar qué es lo que ha ocurrido físicamente en la base de datos.

Se crea la tabla MiTabla3 usando PARTITION BY HASH (puntos) para realizar la partición mediante la función HASH.

```
mibasedatos=# CREATE TABLE MiTabla3(id_cliente integer, nombre TEXT, apellidos TEXT, direccion TEXT, puntos INTEGER) PA
RTITION BY HASH (puntos);
CREATE TABLE
```

Se consulta el oid de MiTabla3 que es 16473

```
mibasedatos=# select oid from pg_class where relname = 'mitabla3';
-[ RECORD 1 ]
oid | 16473
```

Y se consulta el fichero en la carpeta de mibasedatos pero no existe físicamente el archivo

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16473
ls: no se puede acceder a 'base/16385/16473': No existe el archivo o el directorio
```

Se especifica el módulo 10 y los distintos restos para que devuelva 10 valores sobre el campo puntos.

```
mibasedatos=# CREATE TABLE MiTabla3_p0 PARTITION OF MiTabla3 (puntos) FOR VALUES WITH (MODULUS 10, REMAINDER 0);
CREATE TABLE
mibasedatos=# CREATE TABLE MiTabla3_p1 PARTITION OF MiTabla3 (puntos) FOR VALUES WITH (MODULUS 10, REMAINDER 1);
CREATE TABLE
mibasedatos=# CREATE TABLE MiTabla3_p2 PARTITION OF MiTabla3 (puntos) FOR VALUES WITH (MODULUS 10, REMAINDER 2);
CREATE TABLE
mibasedatos=# CREATE TABLE MiTabla3_p3 PARTITION OF MiTabla3 (puntos) FOR VALUES WITH (MODULUS 10, REMAINDER 3);
CREATE TABLE
mibasedatos=# CREATE TABLE MiTabla3_p4 PARTITION OF MiTabla3 (puntos) FOR VALUES WITH (MODULUS 10, REMAINDER 4);
CREATE TABLE
mibasedatos=# CREATE TABLE MiTabla3_p5 PARTITION OF MiTabla3 (puntos) FOR VALUES WITH (MODULUS 10, REMAINDER 5);
CREATE TABLE
mibasedatos=# CREATE TABLE MiTabla3_p6 PARTITION OF MiTabla3 (puntos) FOR VALUES WITH (MODULUS 10, REMAINDER 6);
CREATE TABLE
mibasedatos=# CREATE TABLE MiTabla3_p7 PARTITION OF MiTabla3 (puntos) FOR VALUES WITH (MODULUS 10, REMAINDER 7);
CREATE TABLE
mibasedatos=# CREATE TABLE MiTabla3_p8 PARTITION OF MiTabla3 (puntos) FOR VALUES WITH (MODULUS 10, REMAINDER 8);
CREATE TABLE
mibasedatos=# CREATE TABLE MiTabla3_p9 PARTITION OF MiTabla3 (puntos) FOR VALUES WITH (MODULUS 10, REMAINDER 9);
CREATE TABLE
```

Se consultan los oid de cada partición

```
mibasedatos=# select oid from pg_class where relname = 'mitabla3_p0' or relname = 'mitabla3_p1' or relname = 'mitabla3_
p2' or relname = 'mitabla3_p3' or relname = 'mitabla3_p4' or relname = 'mitabla3_p5' or relname = 'mitabla3_p6' or reln
ame = 'mitabla3_p7' or relname = 'mitabla3_p8' or relname = 'mitabla3_p9';
  oid
-------
 16482
 16476
 16488
 16500
 16494
 16506
 16512
 16524
 16518
 16530
(10 filas)
```

Y se comprueba que existen cada uno de los ficheros para cada partición

```
postgres@ubuntu:~/12/main$ ls -lh base/16385/16482
-rw------- 1 postgres postgres 0 feb 28 02:47 base/16385/16482
postgres@ubuntu:~/12/main$ ls -lh base/16385/16476
-rw------- 1 postgres postgres 0 feb 28 02:43 base/16385/16476
postgres@ubuntu:~/12/main$ ls -lh base/16385/16488
-rw------- 1 postgres postgres 0 feb 28 02:47 base/16385/16488
postgres@ubuntu:~/12/main$ ls -lh base/16385/16500
-rw------- 1 postgres postgres 0 feb 28 02:47 base/16385/16500
postgres@ubuntu:~/12/main$ ls -lh base/16385/16494
-rw------- 1 postgres postgres 0 feb 28 02:47 base/16385/16494
postgres@ubuntu:~/12/main$ ls -lh base/16385/16506
-rw------- 1 postgres postgres 0 feb 28 02:47 base/16385/16506
postgres@ubuntu:~/12/main$ ls -lh base/16385/16512
-rw------- 1 postgres postgres 0 feb 28 02:47 base/16385/16512
postgres@ubuntu:~/12/main$ ls -lh base/16385/16524
-rw------- 1 postgres postgres 0 feb 28 02:48 base/16385/16524
postgres@ubuntu:~/12/main$ ls -lh base/16385/16518
-rw------- 1 postgres postgres 0 feb 28 02:47 base/16385/16518
postgres@ubuntu:~/12/main$ ls -lh base/16385/16530
-rw------- 1 postgres postgres 0 feb 28 02:48 base/16385/16530
```

Por lo tanto, físicamente la base de datos no crea un fichero para MiTabla3 como se hacía anteriormente, sino que ahora como se usan particiones entonces se divide en varias tablas más pequeñas y eso se refleja en los distintos ficheros que se crean.

Cuestión 21. ¿Cuántos bloques ocupa cada una de las particiones? ¿Por qué? Comparar con el número bloques que se obtendría teóricamente utilizando el procedimiento visto en teoría.

```
mibasedatos=# select relpages from pg_class where relname = 'mitabla3_p0' or relname = 'mitabla3_p1'
or relname = 'mitabla3_p2' or relname = 'mitabla3_p3' or relname = 'mitabla3_p4' or relname = 'mitabl
a3_p5' or relname = 'mitabla3_p6' or relname = 'mitabla3_p7' or relname = 'mitabla3_p8' or relname =
'mitabla3_p9';
-[ RECORD 1 ]---
relpages | 15531
-[ RECORD 2 ]---
relpages | 16220
-[ RECORD 3 ]---
relpages | 16682
-[ RECORD 4 ]---
relpages | 19200
-[ RECORD 5 ]---
relpages | 13715
-[ RECORD 6 ]---
relpages | 15094
-[ RECORD 7 ]---
relpages | 17594
-[ RECORD 8 ]---
relpages | 15520
-[ RECORD 9 ]---
relpages | 14626
-[ RECORD 10 ]--
relpages | 15747
```

Cada partición tiene un número distinto de bloques, esto es debido a que se utiliza la función hash sobre el campo puntos. Dependiendo del resto a la hora de hacer el módulo 10 de puntos, se insertará en una partición u otra.

Función de asociación: puntos mod 10

Números de cajones: 10

Longitud de id_cliente: 4 bytes

Longitud de nombre: 14 bytes

Longitud de apellidos: 17 bytes

Longitud de dirección: 17 bytes

Longitud de puntos: 4 bytes

Tamaño del bloque: 8192 bytes

La longitud del registro será: LR = 4 + 14 + 17 + 17 + 4 = 56 bytes.

Nr = 15.000.000 registros

Cuantos registros hay por cajón: 15.000.000 / 10 = 1.500.000 registros / cajón hash

Número de registros / bloque = Factor de bloque = 8192 / 56 = 146,2857143 = 146

Número de bloques = 1.500.000 / 146 = 10274 bloques / cajón hash

En total hay 10274 * 10 = 102740 bloques

Como ocurre en la cuestión 3, no concuerda con el mismo número de bloques de PostgreSQL seguramente porque no estamos teniendo en cuenta los 24 bytes de la cabecera de cada bloque que PostgreSQL añade y que los campos TEXT al ser

dinámicos, y que aquí se calcula su media, puede provocar que sea más complicado dar con el mismo resultado.

Monitorización de la actividad de la base de datos

En este último apartado se mostrará el acceso a los datos con una serie de consultas sobre la tabla original. Para ello, borrar todas las tablas creadas y volver a crear la tabla MiTabla como en la cuestión 2. Cargar los datos que se encuentran originalmente en el fichero datos_mitabla.txt

Cuestión 22. ¿Qué herramientas tiene PostgreSQL para monitorizar la actividad de la base de datos sobre el disco? ¿Qué información de puede mostrar con esas herramientas? ¿Sobre qué tipo de estructuras se puede recopilar información de la actividad? Describirlo brevemente.

PostgreSQL usa consultas para monitorizar el uso que hace la base de datos sobre el disco. Para ver el uso del disco que hace cada tabla medida en bloques se usa la siguiente consulta.

```
mibasedatos=# SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE relname = 'mitabla';
 pg_relation_filepath | relpages
----------------------+----------
 base/16433/16464     |   159926
(1 fila)
```

Cuando el tamaño de los datos almacenados en una tabla son anchos, se crea un archivo TOAST que ayuda a su almacenamiento. Este archivo también contiene índices y tablas asociadas con la tabla de la base de datos. El espacio ocupado por estos archivos también se pueden saber a través de una consulta.

```
mibasedatos=# SELECT relname, relpages FROM pg_class, (SELECT reltoastrelid FROM pg_class WHERE relname = 'mitabla') AS ss
WHERE oid = ss.reltoastrelid OR oid = (SELECT indexrelid FROM pg_index WHERE indrelid = ss.reltoastrelid) ORDER BY relname;

      relname       | relpages
--------------------+----------
 pg_toast_16464     |        0
 pg_toast_16464_index |        1
(2 filas)
```

Además, se puede consultar el tamaño de los índices.

```
mibasedatos=# SELECT c2.relname, c2.relpages FROM pg_class c, pg_class c2, pg_index i WHERE
c.relname = 'mitabla' AND c.oid = i.indrelid AND c2.oid = i.indexrelid ORDER BY c2.relname;
   relname    | relpages
--------------+----------
 mitabla_pkey |    54615
(1 fila)
```

Finalmente, se puede saber la tabla y el índice que más espacio ocupa con la siguiente consulta, ya que muestra el tamaño de todos ellos de mayor a menor.

```
mibasedatos=# SELECT relname, relpages FROM pg_class ORDER BY relpages DESC;
                  relname                  | relpages
-------------------------------------------+----------
 mitabla                                   |   159926
 mitabla_pkey                              |    54615
 pg_proc                                   |       79
 pg_toast_2618                             |       57
 pg_depend                                 |       56
```

Cuestión 23. Crear un índice primario btree sobre el campo puntos. ¿Cuál ha sido el proceso seguido?

Para crear índices, se usa el comando create index. Este comando, por defecto, crea un índice de tipo btree.

```
mibasedatos=# create index indice_puntos on mitabla(puntos);
CREATE INDEX
```

Cuestión 24. Crear un índice hash sobre el campo puntos y otro sobre id_cliente

```
mibasedatos=# create index indice_hash_puntos on mitabla using hash (puntos);
CREATE INDEX
```

```
mibasedatos=# create index indice_hash_id_cliente on mitabla using hash (id_cliente);
CREATE INDEX
```

Cuestión 25. Analizar el tamaño de todos los índices creados y compararlos entre sí. ¿Qué conclusiones se pueden extraer de dicho análisis?

```
mibasedatos=# SELECT c2.relname, c2.relpages FROM pg_class c, pg_class c2, pg_index i WHERE
c.relname = 'mitabla' AND c.oid = i.indrelid AND c2.oid = i.indexrelid ORDER BY c2.relname;
        relname         | relpages
------------------------+----------
 indice_hash_id_cliente |    65391
 indice_hash_puntos     |    85625
 indice_puntos          |    41188
 mitabla_pkey           |    54615
(4 filas)
```

El indice_hash_id_cliente ocupa 65391 bloques, el indice_hash_puntos ocupa 85625 bloques y el indice_puntos (btree) ocupa 41188 bloques. Se puede observar que los índices hash ocupan más espacio que los btree.

Cuestión 26. Para cada una de las consultas que se muestran a continuación, ¿Qué información se puede obtener de los datos monitorizados por la base de datos al realizar la consulta? ¿Comentar cómo se ha realizado la resolución de la consulta? ¿Cuántos bloques se han leído? ¿Por qué? Importante, reinicializar los datos recolectados de la actividad de la base de datos antes de lanzar cada consulta:

```
mibasedatos=# SELECT * from pg_statio_user_indexes ;
 relid | indexrelid | schemaname | relname |      indexrelname      | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey           |       4952456 |     40099098
 16464 |      16476 | public     | mitabla | indice_puntos          |             0 |            0
 16464 |      16477 | public     | mitabla | indice_hash_puntos     |             0 |            0
 16464 |      16478 | public     | mitabla | indice_hash_id_cliente |             0 |            0
(4 filas)
```

Con la consulta anterior, podemos saber los bloques que han sido leídos en cada índice en el disco (idx_blks_read) y en el buffer (idx_blks_hit). A partir de esta información,

vemos como los bloques leídos aumentan. Para empezar, se reinician esos datos estadísticos con la siguiente consulta. Se reiniciarán antes de realizar cada consulta.

```
mibasedatos=# select pg_stat_reset();
 pg_stat_reset
---------------

(1 fila)


mibasedatos=# SELECT * from pg_statio_user_indexes ;
 relid | indexrelid | schemaname | relname |      indexrelname      | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey           |             0 |            0
 16464 |      16476 | public     | mitabla | indice_puntos          |             0 |            0
 16464 |      16477 | public     | mitabla | indice_hash_puntos     |             0 |            0
 16464 |      16478 | public     | mitabla | indice_hash_id_cliente |             0 |            0
(4 filas)
```

1. Mostar la información de las tuplas con id_cliente=8.101.000.

```
mibasedatos=# select * from mitabla where id_cliente=8101000;
 id_cliente |    nombre     |    apellidos     |    direccion     | puntos
------------+---------------+------------------+------------------+--------
    8101000 | nombre8101000 | apellidos8101000 | apellidos8101000 |    391
(1 fila)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |      indexrelname      | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey           |             0 |            0
 16464 |      16476 | public     | mitabla | indice_puntos          |             0 |            0
 16464 |      16477 | public     | mitabla | indice_hash_puntos     |             0 |            0
 16464 |      16478 | public     | mitabla | indice_hash_id_cliente |             2 |            0
(4 filas)
```

La consulta se ha realizado a través del índice hash de id_cliente en mitabla. Como se puede observar, se ha realizado la lectura de 2 bloques en indice_hash_id_cliente.

2. Mostrar la información de las tuplas con id_cliente <30000.

```
mibasedatos=# select * from mitabla where id_cliente<30000;
 id_cliente |   nombre    |   apellidos    |   direccion    | puntos
------------+-------------+----------------+----------------+--------
        337 | nombre337   | apellidos337   | apellidos337   |     59
      24164 | nombre24164 | apellidos24164 | apellidos24164 |    534
      20407 | nombre20407 | apellidos20407 | apellidos20407 |    470
       1623 | nombre1623  | apellidos1623  | apellidos1623  |    157
        450 | nombre450   | apellidos450   | apellidos450   |    315
      11906 | nombre11906 | apellidos11906 | apellidos11906 |    220
      15805 | nombre15805 | apellidos15805 | apellidos15805 |    603
       4640 | nombre4640  | apellidos4640  | apellidos4640  |    653
       2998 | nombre2998  | apellidos2998  | apellidos2998  |    454
```

```
 13661 | nombre13661 | apellidos13661 | apellidos13661 |     674
  3743 | nombre3743  | apellidos3743  | apellidos3743  |      33
  5700 | nombre5700  | apellidos5700  | apellidos5700  |     241
 15015 | nombre15015 | apellidos15015 | apellidos15015 |     223
  7956 | nombre7956  | apellidos7956  | apellidos7956  |     423
(30000 filas)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |      indexrelname      | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey           |           100 |            3
 16464 |      16476 | public     | mitabla | indice_puntos          |             0 |            0
 16464 |      16477 | public     | mitabla | indice_hash_puntos     |             0 |            0
 16464 |      16478 | public     | mitabla | indice_hash_id_cliente |             0 |            0
(4 filas)
```

Se han leído 100 bloques del índice mitabla_pkey. Se han tenido que leer 3 bloques en el buffer para realizar correctamente la búsqueda.

3. Mostrar el número de tuplas cuyo id_cliente >8000 y id_cliente <100000.

```
mibasedatos=# select count(*) from mitabla where id_cliente>8000 and id_cliente<100000;
 count
-------
 91999
(1 fila)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |      indexrelname      | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey           |           315 |            5
 16464 |      16476 | public     | mitabla | indice_puntos          |             0 |            0
 16464 |      16477 | public     | mitabla | indice_hash_puntos     |             0 |            0
 16464 |      16478 | public     | mitabla | indice_hash_id_cliente |             0 |            0
(4 filas)
```

Para realizar esta consulta, se usa el índice mitabla_pkey. Se leen 5 bloques en el buffer del índice y se recuperan los datos después de la lectura de 315 bloques en el disco.

4. Mostar la información de las tuplas con id_cliente=34500 o id_cliente=30.204.000.

```
mibasedatos=# select * from mitabla where id_cliente=34500 or id_cliente=30204000;
 id_cliente |   nombre    |   apellidos    |   direccion    | puntos
------------+-------------+----------------+----------------+--------
      34500 | nombre34500 | apellidos34500 | apellidos34500 |     17
(1 fila)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |      indexrelname      | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey           |             0 |            0
 16464 |      16476 | public     | mitabla | indice_puntos          |             0 |            0
 16464 |      16477 | public     | mitabla | indice_hash_puntos     |             0 |            0
 16464 |      16478 | public     | mitabla | indice_hash_id_cliente |             2 |            0
(4 filas)
```

En esta consulta, se leen 2 bloques del índice indice_hash_id_cliente. Solamente se muestra la tupla con id_cliente=24500 ya que es la única de las dos que existe en el índice, por lo que solo se busca esa.

5. Mostrar las tuplas cuyo id_cliente es distinto de 3450000.

```
mibasedatos=# select * from mitabla where id_cliente<>3450000;
 id_cliente |     nombre      |     apellidos      |     direccion      | puntos
------------+-----------------+--------------------+--------------------+--------
        337 | nombre337       | apellidos337       | apellidos337       |     59
    9020289 | nombre9020289   | apellidos9020289   | apellidos9020289   |    624
   14519336 | nombre14519336  | apellidos14519336  | apellidos14519336  |    132
    7724637 | nombre7724637   | apellidos7724637   | apellidos7724637   |    278
    3943592 | nombre3943592   | apellidos3943592   | apellidos3943592   |    520
    1343216 | nombre1343216   | apellidos1343216   | apellidos1343216   |    392
    8733445 | nombre8733445   | apellidos8733445   | apellidos8733445   |    566
    8878781 | nombre8878781   | apellidos8878781   | apellidos8878781   |    358
```

```
mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |      indexrelname      | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey           |             0 |            1
 16464 |      16476 | public     | mitabla | indice_puntos          |             0 |            1
 16464 |      16477 | public     | mitabla | indice_hash_puntos     |             0 |            0
 16464 |      16478 | public     | mitabla | indice_hash_id_cliente |             0 |            0
(4 filas)
```

Se consulta en los índices mitabla_pkey e indice_puntos leyendo un bloque en cada uno, pero no se lee ningún bloque de estos índices para devolver todos los registros que se muestran.

6. Mostrar las tuplas que tiene un nombre igual a ‘nombre3456789’.

```
mibasedatos=# select * from mitabla where nombre='nombre3456789';
 id_cliente |    nombre     |    apellidos     |    direccion     | puntos
------------+---------------+------------------+------------------+--------
    3456789 | nombre3456789 | apellidos3456789 | apellidos3456789 |    126
(1 fila)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |      indexrelname      | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey           |             0 |            1
 16464 |      16476 | public     | mitabla | indice_puntos          |             0 |            1
 16464 |      16477 | public     | mitabla | indice_hash_puntos     |             0 |            0
 16464 |      16478 | public     | mitabla | indice_hash_id_cliente |             0 |            0
(4 filas)
```

Ocurre lo mismo que en la consulta anterior.

7. Mostar la información de las tuplas con puntos=650.

```
mibasedatos=# select * from mitabla where puntos=650;
 id_cliente |     nombre      |     apellidos      |     direccion      | puntos
------------+-----------------+--------------------+--------------------+--------
   11893144 | nombre11893144  | apellidos11893144  | apellidos11893144  |    650
   10294047 | nombre10294047  | apellidos10294047  | apellidos10294047  |    650
    4078824 | nombre4078824   | apellidos4078824   | apellidos4078824   |    650
    3792006 | nombre3792006   | apellidos3792006   | apellidos3792006   |    650
   12203889 | nombre12203889  | apellidos12203889  | apellidos12203889  |    650
    3463237 | nombre3463237   | apellidos3463237   | apellidos3463237   |    650
     582936 | nombre582936    | apellidos582936    | apellidos582936    |    650
```

```
   1949285 | nombre1949285 | apellidos1949285 | apellidos1949285 |    650
   9286921 | nombre9286921 | apellidos9286921 | apellidos9286921 |    650
(21362 filas)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |      indexrelname      | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey           |             0 |            0
 16464 |      16476 | public     | mitabla | indice_puntos          |            61 |            0
 16464 |      16477 | public     | mitabla | indice_hash_puntos     |             0 |            0
 16464 |      16478 | public     | mitabla | indice_hash_id_cliente |             0 |            0
(4 filas)
```

En esta consulta se usa el índice btree indice_puntos. Se leen 61 bloques.

8. Mostrar la información de las tuplas con puntos<200.

```
mibasedatos=# select * from mitabla where puntos<200;
 id_cliente |     nombre      |     apellidos      |     direccion      | puntos
------------+-----------------+--------------------+--------------------+--------
        337 | nombre337       | apellidos337       | apellidos337       |     59
   14519336 | nombre14519336  | apellidos14519336  | apellidos14519336  |    132
    8886110 | nombre8886110   | apellidos8886110   | apellidos8886110   |     65
    1074928 | nombre1074928   | apellidos1074928   | apellidos1074928   |    134
    1259508 | nombre1259508   | apellidos1259508   | apellidos1259508   |    126
```

```
mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |      indexrelname      | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey           |             0 |            1
 16464 |      16476 | public     | mitabla | indice_puntos          |             0 |            1
 16464 |      16477 | public     | mitabla | indice_hash_puntos     |             0 |            0
 16464 |      16478 | public     | mitabla | indice_hash_id_cliente |             0 |            0
(4 filas)
```

La consulta se realiza de la misma forma que la consulta 5.

9. Mostrar la información de las tuplas con puntos>30000.

```
mibasedatos=# select * from mitabla where puntos>30000;
 id_cliente | nombre | apellidos | direccion | puntos
------------+--------+-----------+-----------+--------
(0 filas)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |      indexrelname      | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey           |             0 |            0
 16464 |      16476 | public     | mitabla | indice_puntos          |             0 |            6
 16464 |      16477 | public     | mitabla | indice_hash_puntos     |             0 |            0
 16464 |      16478 | public     | mitabla | indice_hash_id_cliente |             0 |            0
(4 filas)
```

No se ha leído ningún bloque porque la consulta no devuelve ningún registro. Se consulta en 6 bloques del índice indice_puntos para comprobar que no hay ningún registro con puntos>30000.

10. Mostrar la información de las tuplas con id_cliente=90000 o puntos=230

```
mibasedatos=# select * from mitabla where  id_cliente=90000 or puntos=230;
 id_cliente |     nombre      |     apellidos      |     direccion      | puntos
------------+-----------------+--------------------+--------------------+--------
   12528122 | nombre12528122  | apellidos12528122  | apellidos12528122  |    230
   13620081 | nombre13620081  | apellidos13620081  | apellidos13620081  |    230
   14966231 | nombre14966231  | apellidos14966231  | apellidos14966231  |    230
    5858278 | nombre5858278   | apellidos5858278   | apellidos5858278   |    230
    9630730 | nombre9630730   | apellidos9630730   | apellidos9630730   |    230
```

```
   13035258 | nombre13035258 | apellidos13035258 | apellidos13035258 |    230
   10140974 | nombre10140974 | apellidos10140974 | apellidos10140974 |    230
(21414 filas)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |      indexrelname      | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey           |             0 |            0
 16464 |      16476 | public     | mitabla | indice_puntos          |            61 |            1
 16464 |      16477 | public     | mitabla | indice_hash_puntos     |             0 |            0
 16464 |      16478 | public     | mitabla | indice_hash_id_cliente |             2 |            0
(4 filas)
```

Se han leído 2 bloques a través del índice indice_hash_id_cliente y 61 bloques a través del índice indice_puntos. Para este último, se ha tenido que consultar en 1 bloque antes de recuperar las tuplas del índice.

11. Mostrar la información de las tuplas con id_cliente=90000 y puntos=230

```
mibasedatos=# select * from mitabla where  id_cliente=90000 and puntos=230;
 id_cliente | nombre | apellidos | direccion | puntos
------------+--------+-----------+-----------+--------
(0 filas)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |      indexrelname      | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey           |             0 |            0
 16464 |      16476 | public     | mitabla | indice_puntos          |             0 |            0
 16464 |      16477 | public     | mitabla | indice_hash_puntos     |             0 |            0
 16464 |      16478 | public     | mitabla | indice_hash_id_cliente |             1 |            0
(4 filas)
```

Para realizar esta consulta se lee un bloque del índice indice_hash_id_cliente.

Cuestión 27. Borrar los índices creados y crear un índice multiclave btree sobre los campos puntos y nombre.

```
mibasedatos=# drop index indice_puntos;
DROP INDEX
mibasedatos=# drop index indice_hash_puntos;
DROP INDEX
mibasedatos=# drop index indice_hash_id_cliente;
DROP INDEX
```

```
mibasedatos=# create index idx_mitabla_puntos_nombre on mitabla (puntos, nombre);
CREATE INDEX
```

Cuestión 28. Para cada una de las consultas que se muestran a continuación, ¿Qué información se puede obtener de los datos monitorizados por la base de datos al realizar la consulta? ¿Comentar cómo se ha realizado la resolución de la consulta? ¿Cuántos bloques se han leído? ¿Por qué? Importante, reinicializar los datos recolectados de la actividad de la base de datos antes de lanzar cada consulta:

1. Mostrar las tuplas cuyos puntos valen 200 y su nombre es nombre3456789.

```
mibasedatos=# select * from mitabla where puntos=200 and nombre='nombre3456789';
 id_cliente | nombre | apellidos | direccion | puntos
------------+--------+-----------+-----------+--------
(0 filas)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |       indexrelname        | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+---------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey              |             1 |            0
 16464 |      16479 | public     | mitabla | idx_mitabla_puntos_nombre |             5 |            0
(2 filas)
```

Se ha leído un bloque del índice mitabla_pkey y 5 bloques en el índice idx_mitabla_puntos_nombre.

2. Mostrar las tuplas cuyos puntos valen 200 o su nombre es nombre3456789.

```
mibasedatos=# select * from mitabla where puntos=200 or nombre='nombre3456789';
 id_cliente |     nombre      |     apellidos     |     direccion     | puntos
------------+-----------------+-------------------+-------------------+--------
   14599538 | nombre14599538 | apellidos14599538 | apellidos14599538 |    200
    1393782 | nombre1393782  | apellidos1393782  | apellidos1393782  |    200
    1425168 | nombre1425168  | apellidos1425168  | apellidos1425168  |    200
    4420134 | nombre4420134  | apellidos4420134  | apellidos4420134  |    200
   11770199 | nombre11770199 | apellidos11770199 | apellidos11770199 |    200
    3440353 | nombre3440353  | apellidos3440353  | apellidos3440353  |    200
    6721516 | nombre6721516  | apellidos6721516  | apellidos6721516  |    200
(21386 filas)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |       indexrelname        | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+---------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey              |             0 |            1
 16464 |      16479 | public     | mitabla | idx_mitabla_puntos_nombre |             0 |            1
(2 filas)
```

No se ha leído ningún bloque en los índices pero se ha comprobado la condición de la consulta con un bloque en cada unos de los índices.

3. Mostrar las tuplas cuyo id_cliente vale 6000 o su nombre es nombre3456789.

```
mibasedatos=# select * from mitabla where id_cliente=6000 or nombre='nombre3456789';
 id_cliente |    nombre     |    apellidos     |    direccion     | puntos
------------+---------------+------------------+------------------+--------
    3456789 | nombre3456789 | apellidos3456789 | apellidos3456789 |    126
       6000 | nombre6000    | apellidos6000    | apellidos6000    |    218
(2 filas)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |       indexrelname        | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+---------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey              |             0 |            0
 16464 |      16479 | public     | mitabla | idx_mitabla_puntos_nombre |             0 |            0
(2 filas)
```

Para esta consulta no se ha usado ningún índice, ya que no hay ningún bloque consultado ni leído tanto del buffer como del disco.

```
mibasedatos=# explain select * from mitabla where id_cliente=6000 or nombre='nombre3456789';
                                QUERY PLAN
---------------------------------------------------------------------------
 Gather  (cost=1000.00..254676.20 rows=2 width=56)
   Workers Planned: 2
   ->  Parallel Seq Scan on mitabla  (cost=0.00..253676.00 rows=1 width=56)
         Filter: ((id_cliente = 6000) OR (nombre = 'nombre3456789'::text))
(4 filas)
```

Se ha realizado una búsqueda secuencial en la propia tabla mitabla.

4. Mostrar las tuplas cuyo id_cliente vale 6000 y su nombre es nombre3456789.

```
mibasedatos=# select * from mitabla where id_cliente=6000 and nombre='nombre3456789';
 id_cliente | nombre | apellidos | direccion | puntos
------------+--------+-----------+-----------+--------
(0 filas)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |       indexrelname        | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+---------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey              |             0 |            3
 16464 |      16479 | public     | mitabla | idx_mitabla_puntos_nombre |             0 |            0
(2 filas)
```

Se ha consultado en el índice mitabla_pkey leyendo 3 bloques pero no se ha recuperado ninguna tupla.

Cuestión 29. Crear la tabla **MiTabla3** como en la cuestión 20. Para cada una de las consultas que se muestran a continuación, ¿Qué información se puede obtener de los datos monitorizados por la base de datos al realizar la consulta? ¿Comentar cómo se ha realizado la resolución de la consulta? ¿Cuántos bloques se han leído? ¿Por qué? Importante, reinicializar los datos recolectados de la actividad de la base de datos antes de lanzar cada consulta:

```
mibasedatos=# create table mitabla3(id_cliente integer, nombre text, apellidos text, direccion text, puntos integer) partition by hash (puntos);
CREATE TABLE
mibasedatos=# create table mitabla3_p0 partition of mitabla3 (puntos) for values with (modulus 10, remainder 0);
CREATE TABLE
mibasedatos=# create table mitabla3_p1 partition of mitabla3 (puntos) for values with (modulus 10, remainder 1);
CREATE TABLE
mibasedatos=# create table mitabla3_p2 partition of mitabla3 (puntos) for values with (modulus 10, remainder 2);
CREATE TABLE
mibasedatos=# create table mitabla3_p3 partition of mitabla3 (puntos) for values with (modulus 10, remainder 3);
CREATE TABLE
mibasedatos=# create table mitabla3_p4 partition of mitabla3 (puntos) for values with (modulus 10, remainder 4);
CREATE TABLE
mibasedatos=# create table mitabla3_p5 partition of mitabla3 (puntos) for values with (modulus 10, remainder 5);
CREATE TABLE
mibasedatos=# create table mitabla3_p6 partition of mitabla3 (puntos) for values with (modulus 10, remainder 6);
CREATE TABLE
mibasedatos=# create table mitabla3_p7 partition of mitabla3 (puntos) for values with (modulus 10, remainder 7);
CREATE TABLE
mibasedatos=# create table mitabla3_p8 partition of mitabla3 (puntos) for values with (modulus 10, remainder 8);
CREATE TABLE
mibasedatos=# create table mitabla3_p9 partition of mitabla3 (puntos) for values with (modulus 10, remainder 9);
CREATE TABLE
```

1. Mostrar las tuplas cuyos puntos valen 200.

```
mibasedatos=# select * from mitabla where puntos=200;
 id_cliente |     nombre     |     apellidos     |     direccion     | puntos
------------+----------------+-------------------+-------------------+--------
   14599538 | nombre14599538 | apellidos14599538 | apellidos14599538 |    200
    1393782 | nombre1393782  | apellidos1393782  | apellidos1393782  |    200
    1425168 | nombre1425168  | apellidos1425168  | apellidos1425168  |    200
    4420134 | nombre4420134  | apellidos4420134  | apellidos4420134  |    200
    3440333 | nombre3440333  | apellidos3440333  | apellidos3440333  |    200
    7949893 | nombre7949893  | apellidos7949893  | apellidos7949893  |    200
    6721516 | nombre6721516  | apellidos6721516  | apellidos6721516  |    200
(21385 filas)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |       indexrelname        | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+---------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey              |             0 |            0
 16464 |      16479 | public     | mitabla | idx_mitabla_puntos_nombre |           110 |            0
(2 filas)
```

Para esta consulta se usa el índice idx_mitabla_puntos_nombre. De este índice se leen 110 bloques.

2. Mostrar las tuplas cuyos puntos valen 200 y 300.

```
mibasedatos=# select * from mitabla where puntos=200 and puntos=300;
 id_cliente | nombre | apellidos | direccion | puntos
------------+--------+-----------+-----------+--------
(0 filas)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |       indexrelname        | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+---------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey              |             0 |            1
 16464 |      16479 | public     | mitabla | idx_mitabla_puntos_nombre |             0 |            1
(2 filas)
```

Se hace una búsqueda en cada uno de los índices con un bloque en cada uno, pero no se ha leído ningún bloque porque la condición no se cumple en ningún caso.

3. Mostrar las tuplas cuyos puntos valen 200 o 202

```
mibasedatos=# select * from mitabla where puntos=200 or puntos=300;
 id_cliente |     nombre     |     apellidos     |     direccion     | puntos
------------+----------------+-------------------+-------------------+--------
   14905667 | nombre14905667 | apellidos14905667 | apellidos14905667 |    300
    6077384 | nombre6077384  | apellidos6077384  | apellidos6077384  |    300
   11237936 | nombre11237936 | apellidos11237936 | apellidos11237936 |    300
   14599538 | nombre14599538 | apellidos14599538 | apellidos14599538 |    200
   [illegible]
    6721516 | nombre6721516  | apellidos6721516  | apellidos6721516  |    200
    4716630 | nombre4716630  | apellidos4716630  | apellidos4716630  |    300
(42860 filas)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |       indexrelname        | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+---------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey              |             0 |            0
 16464 |      16479 | public     | mitabla | idx_mitabla_puntos_nombre |           218 |            1
(2 filas)
```

Se leen 218 bloques para realizar esta consulta y recuperar las tuplas. También se lee un bloque más, pero no es de recuperación de los datos. Todo esto se realiza sobre el índice idx_mitabla_puntos_nombre.

4. Mostrar las tuplas cuyos puntos son > 500.

```
mibasedatos=# select * from mitabla where puntos>500;
 id_cliente |     nombre     |     apellidos     |     direccion     | puntos
------------+----------------+-------------------+-------------------+--------
    9020289 | nombre9020289  | apellidos9020289  | apellidos9020289  |    624
    3943592 | nombre3943592  | apellidos3943592  | apellidos3943592  |    520
    8733445 | nombre8733445  | apellidos8733445  | apellidos8733445  |    566
   13149508 | nombre13149508 | apellidos13149508 | apellidos13149508 |    696
```

```
mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |       indexrelname        | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+---------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey              |             0 |            1
 16464 |      16479 | public     | mitabla | idx_mitabla_puntos_nombre |             0 |            1
(2 filas)


mibasedatos=# explain select * from mitabla where puntos>500;
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on mitabla  (cost=0.00..347426.00 rows=4291180 width=56)
   Filter: (puntos > 500)
(2 filas)
```

Esta consulta se realiza a través de una búsqueda secuencial en la tabla mitabla. Los bloques que se leen en los índices, un bloque en el índice mitabla_pkey y un bloque en el índice idx_mitabla_puntos_nombre, no se usan para recuperar los datos.

5. Mostrar las tuplas cuyos puntos son > 500 y < 550.

```
mibasedatos=# select * from mitabla where puntos>500 and puntos<550;
 id_cliente |     nombre     |     apellidos     |     direccion     | puntos
------------+----------------+-------------------+-------------------+--------
    3943592 | nombre3943592  | apellidos3943592  | apellidos3943592  |    520
    7385730 | nombre7385730  | apellidos7385730  | apellidos7385730  |    540
    3354119 | nombre3354119  | apellidos3354119  | apellidos3354119  |    546
    8249591 | nombre8249591  | apellidos8249591  | apellidos8249591  |    514
```

```
mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |       indexrelname        | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+---------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey              |             0 |            1
 16464 |      16479 | public     | mitabla | idx_mitabla_puntos_nombre |             0 |            1
(2 filas)


mibasedatos=# explain select * from mitabla where puntos>500 and puntos<550;
                                QUERY PLAN
---------------------------------------------------------------------------
 Gather  (cost=1000.00..361687.30 rows=1070113 width=56)
   Workers Planned: 2
   ->  Parallel Seq Scan on mitabla  (cost=0.00..253676.00 rows=445880 width=56)
         Filter: ((puntos > 500) AND (puntos < 550))
(4 filas)
```

En esta consulta ocurre lo mismo que en la anterior.

6. Mostrar las tuplas cuyos puntos son 800

```
mibasedatos=# select * from mitabla where puntos=800;
 id_cliente | nombre | apellidos | direccion | puntos
------------+--------+-----------+-----------+--------
(0 filas)


mibasedatos=# select * from pg_statio_user_indexes;
 relid | indexrelid | schemaname | relname |       indexrelname        | idx_blks_read | idx_blks_hit
-------+------------+------------+---------+---------------------------+---------------+--------------
 16464 |      16470 | public     | mitabla | mitabla_pkey              |             0 |            0
 16464 |      16479 | public     | mitabla | idx_mitabla_puntos_nombre |             0 |            4
(2 filas)
```

Para esta consulta se leen 4 bloques en el índice idx_mitabla_puntos_nombre pero no se lee ningún bloque que recupere datos. Esto se debe a que no hay ninguna tupla cumple la condición de la consulta.

Cuestión 30. A la vista de los resultados obtenidos de este apartado, comentar las conclusiones que se pueden obtener del acceso de PostgreSQL a los datos almacenados en disco.

PostgreSQL ofrece tablas con datos estadísticos que ayudan a la gestión de las consultas. Para recuperar los datos que se piden al realizar una consulta, se tiene que acceder a los datos del disco. El acceso a los datos almacenados en disco es una tarea costosa, por lo que los datos estadísticos ayudan a que los recursos no sean malgastados.

Al acceder a estos datos estadísticos podemos saber el espacio ocupado en el disco por las diferentes tablas o índices así como los bloques que se leen en una consulta. Los comandos usados en los ejercicios anteriores nos ayuda a saber todos esos datos para poder conocer nuestra base de datos de una forma más profunda.

Bibliografía (PostgreSQL 12)

- Capítulo 1: Getting Started.
- Capítulo 5: 5.5 System Columns.
- Capítulo 5: 5.11 Table Partitioning.
- Capítulo 11: Indexes.
- Capítulo 19: Server Configuration.
- Capítulo 24: Routine Database Maintenance Tasks.
- Capítulo 28: Monitoring Database Activity.
- Capítulo 29: Monitoring Disk Usage.
- Capítulo VI.II: PostgresSQL Client Applications.
- Capítulo VI.III: PostgresSQL Server Applications.
- Capítulo 50: System Catalogs.
- Capítulo 68: Database Physical Storage.
- Apéndice F: Additional Supplied Modules.
- Apéndice G: Additional Supplied Programs.