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: Patricia Cuesta Ruiz

DNI: 03211093V

ALUMNO 2:

Nombre y Apellidos: Álvaro Golbano Durán

DNI: 03202759D

Fecha: 02/03/2020

Profesor Responsable: Óscar Gutiérrez

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 encuentra en la carpeta de PostgresSQL en /data/base, donde se muestran todas las bases de datos que hay creadas.

Para poder saber cuál es MiBaseDatos y así poder acceder a los ficheros de la misma es necesario saber su OID. El OID se obtiene en pgAdmin, accediendo a MiBaseDatos/Catalogs/pg_catalog/Tables/pg_database, realizando click derecho sobre pg_database y pulsar View/All Rows mostrando así los OIDs y nombres de todas las bases de datos creadas. El OID de MiBaseDatos es 17474.

Sabiendo el OID podemos acceder a los ficheros entrando en el directorio 17474. Los ficheros que se crean son los básicos que tiene una base de datos al ser creada. Además, el tamaño en disco de MiBaseDatos es de 7.76MB.

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?

Se han creado cuatro nuevos ficheros en el directorio de nuestra base de datos 16393, uno para la tabla de tamaño 8KB, otro para las Primary Keys de 8KB, y otros dos archivos asociados para los atributos de tipo integer de 0KB. No se crea ningún archivo asociado para los atributos de tipo text.

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

Al insertar la tupla:

```
INSERT INTO "MiTabla" VALUES (132444, 'Patricia', 'Cuesta Ruiz', 'Av Ejercito', 7)
```

el fichero que fue creado para la tabla pasa de ocupar 8KB a ocupar 16KB y uno de los archivos asociados a los atributos de tipo integer pasa de 0KB a 8KB. Esto sucede debido a que se han insertado datos.

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?

En primer lugar se crea la extensión *pg_buffercache*.

```
CREATE EXTENSION pg_buffercache
```

Luego, en la carpeta de PostgresSQL se accede al directorio /share/extension y buscamos el archivo pg_buffercache--1.0--1.1.sql donde obtenemos la siguiente consulta:

```
CREATE OR REPLACE VIEW pg_buffercache AS
    SELECT P.* FROM pg_buffercache_pages() AS P
    (bufferid integer, relfilenode oid, reltablespace oid, reldatabase oid,
     relforknumber int2, relblocknumber int8, isdirty bool, usagecount int2,
     pinning_backends int4);
```

Finalmente, al buscar en la documentación pg_buffercache encontramos la consulta necesaria para que se muestre una tabla con los elementos de los buffers más usados.

```
SELECT c.relname, count(*) AS buffers
             FROM pg_buffercache b INNER JOIN pg_class c
             ON b.relfilenode = pg_relation_filenode(c.oid) AND
                b.reldatabase IN (0, (SELECT oid FROM pg_database
                                      WHERE datname = current_database()))
             GROUP BY c.relname
             ORDER BY 2 DESC
             LIMIT 10;
```

Siendo el resultado de la consulta el siguiente:

	relname name	buffers bigint
1	pg_depend	59
2	pg_attribute	35
3	pg_proc	28
4	pg_operator	14
5	pg_class	14
6	pg_statistic	13
7	pg_type	11
8	pg_proc_pro...	11
9	pg_proc_oid...	10
10	pg_attribute...	9

Es lógico que muestre esa cantidad ya que muestra los 10 procesos con mayor necesidad de memoria, que son los necesarios para que la base de datos esté activa.

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 archivo txt a insertar ocupa casi 1GB. Al insertarlo en MiTabla el archivo físico asociado pasa a ocupar exactamente 1GB, ya que hemos insertado casi 1GB de datos y además es necesaria la indexación de las tuplas.

NÚMERO DE BLOQUES TEÓRICO:

Para saber la longitud de cada campo:

- **Si el campo era un text: hemos tenido que realizar las siguientes consultas:**

```
SELECT MAX(LENGTH("nombre"))
FROM public."MiTabla"

SELECT MAX(LENGTH("apellidos"))
FROM public."MiTabla"

SELECT MAX(LENGTH("direccion"))
FROM public."MiTabla"
```

  Obteniendo L_{nombre} = 14 bytes, $L_{apellidos}$ = 17 bytes y $L_{direccion}$ = 17 bytes.

- **Si el campo era un int: el tamaño máximo de un int en SQL es 4 bytes, por lo que L_{id_nombre} = 4 bytes y L_{puntos} = 4 bytes.**

$$L_R = L_{id\_nombre} + L_{nombre} + L_{apellidos} + L_{direccion} + L_{puntos} = 56$$

$$f_R = \left\lfloor \frac{B}{L_R} \right\rfloor = \left\lfloor \frac{8.192}{56} \right\rfloor = 146\ registros/bloque$$

$$b_R = \left\lceil \frac{n_R}{f_R} \right\rceil = \left\lceil \frac{15.000.000}{146} \right\rceil = 102.740\ bloques$$

NÚMERO DE BLOQUES FÍSICO:

Para calcularlo dividimos el tamaño del archivo correspondiente a la tabla entre lo que ocupa un bloque/página:

$$b_R = \left\lceil \frac{1.048.576}{8} \right\rceil = 131.072\ bloques$$

NÚMERO DE BLOQUES REAL:

Realizamos la siguiente consulta:

```
SELECT MAX((ctid::text::point)[0]::bigint) as bloque
FROM "MiTabla"
```

Y obtenemos el siguiente resultado:

	bloque bigint
1	159925

El teórico difiere ya que solo tiene en cuenta el número de registros, el físico solo tiene en cuenta el archivo asociado a MiTabla y el real no tiene en cuenta solo la tabla, sino todos los archivos necesarios para organizar la base de datos, como por ejemplo, el índice de las Primary Keys de la tabla para que no haya ninguna repetida.

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?

Al introducir los mismos comandos que en la cuestión 4 lo que se muestra es lo siguiente:

	relname name	buffers bigint
1	mitabla_pk	15728
2	mitabla	252
3	pg_depend	59
4	pg_attribute	22
5	pg_proc	22
6	pg_class	17
7	pg_rewrite	15
8	pg_type	14
9	pg_statistic	14
10	pg_operator	13

Podemos deducir que al haber insertado todos los datos, ha sido necesario cargar en memoria todas las Primary Keys de todas las tuplas para la indexación y además MiTabla ocupa una parte debido a que al introducir datos ha sido necesario cargarla en memoria.

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?

En primer lugar se crea la extensión *pgstattuple*.

```
CREATE EXTENSION pgstattuple
```

Para después realizar la siguiente consulta:

```
SELECT * FROM pgstattuple('"MiTabla"')
```

Siendo el resultado de la misma:

	table_len bigint	tuple_count bigint	tuple_len bigint	tuple_percent double precision	dead_tuple_count bigint	dead_tuple_len bigint	dead_tuple_percent double precision	free_space bigint	free_percent double precision
1	1310113792	15000000	1219555960	93.09	0	0	0	5761792	0.44

Las estadísticas muestran la longitud física de la relación MiTabla, al igual que el porcentaje de tuplas vivas y muertas, el espacio libre en bytes y su correspondiente porcentaje, al igual que otros datos que no son útiles ahora mismo.

El grado de ocupación de los bloques es la variable *tuple_percent* que es equivalente al 93.09% aproximadamente, ya que este valor indica un porcentaje estimado de ocupación. También se puede calcular al dividir el número de tuplas vivas entre el tamaño máximo de la relación, multiplicado por 100.

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.

Para obtener el factor de bloque medio real de la tabla es necesario realizar las siguientes consultas:

```
SELECT MAX((ctid::text::point)[0]::bigint) as bloque
FROM "MiTabla"
```

	bloque bigint
1	159925

```
SELECT avg(bloque)
  FROM
    (SELECT count (*) AS bloque
     FROM "MiTabla"
     GROUP BY (ctid::text::point)[0]::bigint) as unzincoporfabor
```

Obteniendo así el siguiente resultado:

	avg numeric
1	93.7933794379900704

Como sabemos que un bloque son 8KB y está ocupado un 94% aproximadamente, el bloque contiene 7700 Bytes de registros. Como un registro ocupa 56 Bytes, hay almacenados realmente $\left\lfloor \frac{7700\ B}{56\ B} \right\rfloor = 137$ registros/bloque.

El factor de bloque teórico se calcula de la siguiente forma:

$$L_R = L_{id\_nombre} + L_{nombre} + L_{apellidos} + L_{direccion} + L_{puntos} = 56$$

$$f_R = \left\lfloor \frac{B}{L_R} \right\rfloor = \left\lfloor \frac{8.192}{56} \right\rfloor = 146\ registros/bloque$$

Como se puede observar, el factor teórico difiere del real en aproximadamente 10 registros.

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?

En primer lugar se crea la extensión *pageinspect*.

```
CREATE EXTENSION pageinspect
```

El primer bloque (0):

```
SELECT * FROM page_header(get_raw_page('"MiTabla"', 0));
```

	lsn pg_lsn	checksum smallint	flags smallint	lower smallint	upper smallint	special smallint	pagesize smallint	version smallint	prune_xid xid
1	0/1A72550	0	0	400	448	8192	8192	4	0

El bloque intermedio (79.962):

```
SELECT * FROM page_header(get_raw_page('"MiTabla"', 79962));
```

	lsn pg_lsn	checksum smallint	flags smallint	lower smallint	upper smallint	special smallint	pagesize smallint	version smallint	prune_xid xid
1	0/4765FE...	0	0	400	448	8192	8192	4	0

El último bloque (159.925):

```
SELECT * FROM page_header(get_raw_page('"MiTabla"', 159925));
```

	lsn pg_lsn	checksum smallint	flags smallint	lower smallint	upper smallint	special smallint	pagesize smallint	version smallint	prune_xid xid
1	0/A1C24C...	0	0	128	5984	8192	8192	4	0

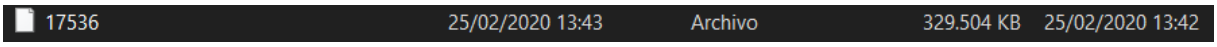
Existe una gran diferencia entre el bloque inicial e intermedio con el bloque final, ya que el bloque final teóricamente es el último en llenarse, por lo que tiene más espacio libre.

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?

Para crear los índices se ejecuta el siguiente comando:

```
CREATE INDEX arbol ON "MiTabla"(puntos);
```

La ruta en la que se almacena es en la carpeta de PostgresSQL, en /data/base, donde se muestran todas las bases de datos que hay creadas.



El tamaño del archivo es de 329.504KB.

El número de bloques teóricos es $\left\lceil \frac{329.504}{8} \right\rceil = 41.188$ bloques.

Ejecutando la siguiente consulta:

```
SELECT * FROM pg_relpages('"arbol"'::regclass);
```

Se obtiene el mismo resultado que el calculado teóricamente:

	pg_relpages bigint
1	41188

Para poder saber las estadísticas que tiene el árbol ejecutamos la siguiente consulta:

```
SELECT * FROM pgstatindex('arbol')
```

Siendo el resultado el siguiente:

	version integer	tree_level integer	index_size bigint	root_block_no bigint	internal_pages bigint	leaf_pages bigint	empty_pages bigint	deleted_pages bigint	avg_leaf_density double precision	leaf_fragmentation double precision
1	4	2	337412096	209	203	40984	0	0	90.19	0

Podemos observar que el número de niveles del árbol es 2 más la raíz, teniendo el árbol así 3 niveles.

Las hojas ocupan 40.984 bloques, el nivel intermedio ocupa 203 bloque y la raíz ocupa 1 bloque, siendo el bloque número 209.

Para saber a cuantas tuplas apunta cada nivel realizamos las siguientes consultas con sus respectivos resultados:

```
SELECT * FROM bt_page_stats('arbol', 209)
```

	blkno integer	type "char" (1)	live_items integer	dead_items integer	avg_item_size integer	page_size integer	free_size integer	btpo_prev integer	btpo_next integer	btpo integer	btpo_flags integer
1	209	r	202	0	23	8192	2508	0	0	2	2

```
SELECT * FROM bt_page_stats('arbol', 208)
```

	blkno integer	type "char" (1)	live_items integer	dead_items integer	avg_item_size integer	page_size integer	free_size integer	btpo_prev integer	btpo_next integer	btpo integer	btpo_flags integer
1	208	i	204	0	23	8192	2452	3	413	1	0

```
SELECT * FROM bt_page_stats('arbol', 320)
```

	blkno integer	type "char" (1)	live_items integer	dead_items integer	avg_item_size integer	page_size integer	free_size integer	btpo_prev integer	btpo_next integer	btpo integer	btpo_flags integer
1	320	l	367	0	16	8192	800	319	321	0	1

La raíz apunta a 23 tuplas, un nodo intermedio apunta a 23 y un nodo hoja apunta a 16 tuplas de media.

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.

Mediante los comandos:

```
SELECT * FROM bt_page_stats('arbol',209)
SELECT * FROM bt_page_items('arbol',209)
```

Siendo 209 el bloque raíz donde lo queremos realizar las estadísticas, podemos calcular la longitud del puntero a bloque. Vemos que el bloque raíz almacena tuplas de longitud 23, por lo que cada tupla almacena un puntero a bloque mas la longitud del campo sobre el que está hecho.

Calculando así que 23 - 4 (L_{campo}) = 19 Bytes de longitud de puntero a bloque y de puntero a registro si buscamos en los comandos de arriba una hoja obtenemos que almacena 16 tuplas. Como sabemos que almacena punteros a registros obtenemos que el tamaño del puntero a registro es de 16 – 4 = 12 Bytes.

Al realizar los cálculos con las fórmulas usadas en teoría, obtenemos que:

$$n * L_{PB} + (n-1) * L_K \leq B \rightarrow n = 356 \frac{registros}{blk}$$

$$n_h * (P_{registro} + L_{campo}) + P_{bloque} \leq B \rightarrow n_h = 430 \frac{registros}{blk}$$

$$Nivel\ hoja \rightarrow \left\lceil \frac{15000000}{430} \right\rceil = 34884\ bloques$$

$$Nivel\ intermedio \rightarrow \left\lceil \frac{34884}{356} \right\rceil = 98\ bloques$$

$$Nivel\ raiz \rightarrow \left\lceil \frac{98}{356} \right\rceil = 1\ bloques$$

Por lo tanto los bloques totales teóricos son : 34983 bloques

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

Los comandos necesarios para crear los hash son:

```
CREATE INDEX hash_idcliente ON "MiTabla" USING HASH (id_cliente);

CREATE INDEX hash_puntos ON "MiTabla" USING HASH (puntos);
```

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?

Aplicamos el módulo pgstattuple:

```
SELECT * FROM pgstathashindex('hash_idcliente');
```

	version integer	bucket_pages bigint	overflow_pages bigint	bitmap_pages bigint	unused_pages bigint	live_items bigint	dead_items bigint	free_percent double precision
1	4	49152	16237	1	0	15000000	0	43.72022039882073

```
SELECT * FROM pgstathashindex('hash_puntos');
```

	version integer	bucket_pages bigint	overflow_pages bigint	bitmap_pages bigint	unused_pages bigint	live_items bigint	dead_items bigint	free_percent double precision
1	4	49152	36470	2	0	15000000	0	57.019475037472716

Ahora aplicamos el módulo pageinspect:

```
SELECT * FROM hash_page_stats(get_raw_page('hash_idcliente', 65390));
```

	live_items integer	dead_items integer	page_size integer	free_size integer	hasho_prevblkno bigint	hasho_nextblkno bigint	hasho_bucket bigint	hasho_flag integer	hasho_page_id integer
1	64	0	8192	6868	32768	4294967295	32767	1	65408

```
SELECT * FROM hash_page_stats(get_raw_page('hash_puntos', 85624));
```

	live_items integer	dead_items integer	page_size integer	free_size integer	hasho_prevblkno bigint	hasho_nextblkno bigint	hasho_bucket bigint	hasho_flag integer	hasho_page_id integer
1	66	0	8192	6828	85623	4294967295	48959	1	65408

Observando la ruta de la base de datos, vemos que el índice sobre id_cliente ocupa 523.128KB y el índice hash sobre puntos ocupa 685.000KB. Esto se refleja también en el número de páginas que encontramos en los índices hash de id_cliente y puntos con el módulo pageinspect (65.380 vs. 85.624).

Esto es congruente con el campo que denota el espacio libre, donde vemos que en el hash de puntos queda menos espacio libre que en el del índice hash de id_cliente.

Al inspeccionar la última página en ambos, vemos que realmente no es la última, sino que es la última sin overflow, ya que ambas comparten el mismo número de ID en sus respectivas tablas. Sin embargo, el hash sobre puntos contiene muchísimas más páginas con overflow respecto al hash de id_cliente.

Todo ello es congruente porque en número de puntos en la tabla escala más rápido que el número de id_cliente. Podemos ver esto consultando las primeras 100 tuplas de la tabla.

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.

INSERCIÓN:

```
INSERT INTO public."MiTabla"(
    id_cliente, nombre, apellidos, direccion, puntos)
    VALUES (69420, 'Alvaro', 'Golbano', 'calle Horcajo', 0)
```

La inserción se realiza en el primer bloque disponible que haya. Al llamar al módulo pageinspect se puede verificar que esa inserción se ha producido al final de la tabla.

MODIFICACIÓN:

```
UPDATE public."MiTabla"
    SET puntos = 69
    WHERE id_cliente = 69420;
```

Esto quiere decir que la tupla se ha movido dentro del archivo, ya que se encuentra en un bloque totalmente diferente, que resulta ser el último bloque del archivo. Esto es debido a que al hacer Update la tupla con el nuevo valor se inserta en la primera posición libre que encuentre, en este caso, el bloque final. Mientras que los valores de la posición que ocupada en el bloque antiguo son puestos a null, generando una tupla muerta.

BORRADO:

```
DELETE FROM public."MiTabla"
    WHERE id_cliente = 69420;
```

Al realizar el borrado, la tupla borrada se convertirá en una tupla muerta que ocupará espacio en memoria.

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.

Al ejecutar la siguiente consulta se eliminan 2.000.000 de tuplas aleatoriamente:

```
DELETE FROM "MiTabla"
WHERE id_cliente IN (SELECT id_cliente
                     FROM "MiTabla"
                     ORDER BY random()
                     LIMIT 2000000)
```

A pesar de que el borrado ha sido correcto, el tamaño de los archivos visto desde el explorador de Windows no ha variado después de ejecutar la consulta. Esto se debe a que POSTGRESQL no elimina el contenido de los bloques directamente, es necesario compactar la base de datos para que estas posiciones dentro de los bloques sean liberados.

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 optimizar la base de datos hemos ejecutado el siguiente comando:

```
VACUUM FULL "MiTabla"
```

La función Vacuum libera el espacio ocupado por las tuplas muertas.

Desde el explorador de archivos se puede ver que el tamaño de los ficheros de la base de datos se ha visto reducido.

Usando el módulo pageinspect podemos comprobar que los espacios dentro de los bloques que antes estaban puestos a null, es decir, las tuplas muertas, ahora tienen información válida.

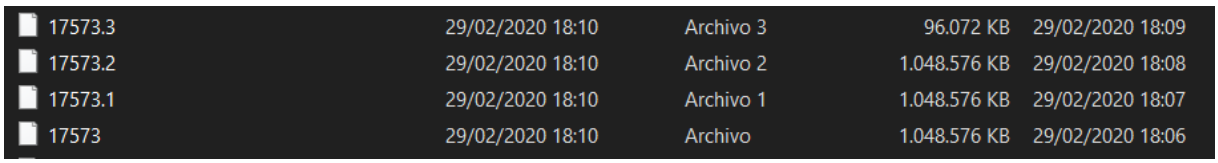
Cuestión 17. Crear una tabla denominada **MiTabla2** de tal manera que tenga un factor de llenado de tuplas que sea un 40% que el de la tabla **MiTabla** y cargar el archivo de datos anterior. Explicar el proceso seguido y qué es lo que ocurre físicamente.

Para crear la tabla MiTabla2 ejecutamos el siguiente comando:

```
CREATE TABLE public."MiTabla2"(
    id_cliente integer,
    nombre text,
    apellidos text,
    direccion text,
    puntos integer,
    PRIMARY KEY (id_cliente))
WITH (
    OIDS = FALSE,
    FILLFACTOR = 40
)
TABLESPACE pg_default;
```

Para que el factor de bloque sea el 40% que el de mi tabla el parámetro FILLFACTOR se iguala a 40.

Al mirar los archivos físicos de la base de datos se ve que se han creado los siguientes cuatro archivos:



Estos archivos asociados a MiTabla2 se han creado ya que el factor de bloque es menor por lo que se necesitan más archivos asociados a la tabla.

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.

INSERCIÓN:

```
INSERT INTO public."MiTabla2"(
    id_cliente, nombre, apellidos, direccion, puntos)
    VALUES (2423532, 'Alvaro', 'Golbano', 'Calle Horcajo', 0);
```

Al igual que hemos explicado en la cuestión 14, la inserción se realiza en el primer bloque disponible que haya. Esto es así ya que, a la hora de insertar, debido a que hemos declarado un FILLFACTOR del 40%, todos los bloques anteriores al último cuentan como llenos desde el punto de vista de una inserción nueva, por lo que, según estos parámetros, el primer bloque libre que encuentra para la inserción es el último.

MODIFICACIÓN:

```
UPDATE public."MiTabla2"
    SET puntos = 69
    WHERE id_cliente = 2423532;
```

La razón por la que la tupla se ha vuelto a insertar en el mismo bloque es que estos bloques no están llenos debido a su FILLFACTOR. Este FILLFACTOR se tiene en cuenta a la hora de insertar, y es por eso que, en la cuestión anterior, la nueva inserción se va al último bloque. Sin embargo, este factor no se tiene en cuenta a la hora de hacer updates, por lo que el bloque no cuenta como lleno y por lo tanto se puede seguir insertando en el mismo bloque. Esto mejora mucho el rendimiento a la hora de usar índices, ya que no necesitan ser modificados al hacer updates.

BORRADO:

```
DELETE FROM public."MiTabla2"
    WHERE id_cliente = 2423532;
```

Al realizar el borrado, la tupla borrada se convertirá en una tupla muerta que ocupará espacio en memoria.

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?

El particionamiento de tablas reduce la cantidad de datos a recorrer en cada consulta y además aumente el rendimiento, ya que cuantos menos datos haya que recorrer, más rápida será la ejecución.

Hay dos tipos de particionamientos:

- **Horizontal: consiste en tener varias tablas con las mismas columnas en cada una de ellas y distribuir equitativamente la cantidad de registros en estas tablas.**
- **Vertical: este particionamiento se basa en guardar en otras tablas los valores de las tuplas poco usados referenciándolas como Foreign Keys.**

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.

Para crear la tabla MiTabla3 con 10 particiones sobre el campo puntos hemos ejecutado los siguientes comandos:

```
CREATE TABLE "MiTabla3" (
    id_cliente int,
    nombre text,
    apellidos text,
    direccion text,
    puntos int) PARTITION BY HASH (puntos);
CREATE TABLE "MiTabla3_0" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 0);
CREATE TABLE "MiTabla3_1" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 1);
CREATE TABLE "MiTabla3_2" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 2);
CREATE TABLE "MiTabla3_3" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 3);
CREATE TABLE "MiTabla3_4" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 4);
CREATE TABLE "MiTabla3_5" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 5);
CREATE TABLE "MiTabla3_6" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 6);
CREATE TABLE "MiTabla3_7" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 7);
CREATE TABLE "MiTabla3_8" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 8);
CREATE TABLE "MiTabla3_9" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 9);
```

Físicamente se han creado 30 nuevos archivos, 3 para cada tabla, uno asociado a la propia tabla y otros dos asociados a los valores tipo integer.

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.

Cada partición tiene estos bloques:

Partición 0: 16219

Partición 1: 15530

Partición 2: 16681

Partición 3: 13714

Partición 4: 19199

Partición 5: 15093

Partición 6: 17593

Parición 7: 14625

Partición 8: 15519

Partición 9: 15746

Ocupando un total de 159.919 bloques totales, con un reparto relativamente equitativo entre particiones del hash, como cabe esperar en una distribución así.

$$n_{RC} = \frac{n_R}{n_C} = \frac{13.000.000}{10} = 1.300.000\ reg/cajón$$

$$b_C = \left\lceil \frac{n_{RC}}{f_{RC}} \right\rceil = \left\lceil \frac{1.300.000}{146} \right\rceil = 8.905\ bloques$$

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 tiene un recolector de estadísticas que funciona como un subsistema que recopila información sobre la actividad del servidor de la base de datos. Puede recopilar información sobre el uso de los índices, así como el acceso a las tablas (y los bloques asociados a dichas estructuras). También recopila información sobre los procesos de vacuum y de analyze para cada tabla. Se guardan también datos sobre las llamadas a las funciones definidas por el usuario en la base de datos.

Este recolector de estadísticas también permite generar reportes dinámicos sobre la actividad de la base de datos en un determinado momento, esto es, por ejemplo, las conexiones activas que tiene la base de datos, o la consulta que se esté ejecutando en un determinado momento sobre la base de datos.

Este subsistema de recolección de estadísticas puede ser configurado libremente e incluso desactivado, ya que obviamente afecta al rendimiento de la base de datos.

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

Para crear el índice primario ejecutamos la siguiente orden SQL.

```
CREATE INDEX indice_primario_puntos ON "MiTabla" (puntos ASC);
```

(ARREGLADO)

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

Índice Hash sobre el campo puntos:

```
CREATE INDEX hash_puntos
ON "MiTabla" USING HASH (puntos);
```

Índice Hash sobre el campo id_cliente:

```
CREATE INDEX hash_id_cliente
ON "MiTabla" USING HASH (id_cliente);
```

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

Nombre Indice:	Tamaño en KB:
Btree sobre campo puntos	329.048 **(ARREGLADO)**
Hash sobre campo puntos	645.656
Hash sobre campo id_cliente	431.456

Podemos observar que los hash ocupan más espacio que los árboles, también es apreciable la diferencia entre los dos hashes de puntos e id_cliente ya que uno esta ordenado por la PRIMARY KEY y el otro no, siendo más grande obviamente el del campo puntos. Viendo los tamaños de los índices podemos llevar a cabo la conclusión de que si se busca organizar las tablas de una forma en la que se ocupe menos espacio en disco, la mejor forma es organizar las tablas en Btree sobre el campo clave y ordenado.

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:

Usando las sentencias EXPLAIN y ANALYZE junto con diferentes parámetros, podemos obtener información sobre el tipo de búsqueda que se va a hacer en la base de datos (secuencial, usando índices...), sobre los costes de dicha consulta, el uso de los buffers y el número de lecturas correctas y/o fallos de lectura escritura, el tiempo que pasa en cada nodo con la consulta, etc.

Se ejecuta el siguiente comando antes de cada consulta para evitar que las estadísticas recolectadas con las mismas influyan en las siguientes consultas:

```
SELECT pg_stat_reset()
```

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
    SELECT *
    FROM "MiTabla"
    WHERE id_cliente = 8101000
```

   Tras este comando se muestra que se han leído 2 bloques sobre el índice hash_id_cliente. Se han leído solo esos bloques ya que al ser un índice hash sobre la PK tiene que leer esos bloques como mucho para encontrar un valor exacto.

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
    SELECT *
    FROM "MiTabla"
    WHERE id_cliente < 30000
```

   Tras la ejecución de este comando se muestra que se han leído 71 bloques sobre la tabla de PK asociada a MiTabla. Como es razonable se han leído muchos bloques ya que estamos buscando todos los id_cliente menores que 30000 en una tabla de 13000000 de elementos.

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
    SELECT COUNT(id_cliente)
    FROM "MiTabla"
    WHERE id_cliente > 8000 AND id_cliente < 100000
```

Tras la ejecución de este comando se muestra que se han leído 219 bloques sobre la tabla de PK asociada a MiTabla. Al igual que la query anterior es razonable que se hayan tenido que leer tantos bloques para encontrar un rango de valores en la tabla.

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla"
WHERE id_cliente = 34500 OR id_cliente = 30204000
```

Tras la ejecución de este comando se muestra en la salida que se han leído 3 bloques en total sobre el hash hash_id_cliente. Al igual que en el primer ejercicio para encontrar valores sueltos en una tabla es razonable que se use un indexado distinto al secuencial de la tabla asociada a la PK. En este caso ha pasado lo mismo, resultando en la lectura de unos pocos bloques en el hash para obtener los dos valores.

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla"
WHERE id_cliente != 3450000
```

Con la ejecución de este comando se muestra que se han leído 122.398 bloques sobre una lectura secuencial de la tabla. Como es lógico si hay que buscar todos los valores distintos a una PK, lo normal es que se lean todos los bloques de la tabla.

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla"
WHERE nombre = 'nombre3456789'
```

Esta ejecución muestra que se han leído 138.601 sobre una lectura paralela secuencial en el archivo asociado a MiTabla. Como tiene que encontrar una sola tupla en toda la tabla, ha de leer muchos bloques.

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla"
WHERE puntos = 650
```

En esta ejecución se muestra que se han leído 17.480 bloques sobre el índice índice_primario_puntos. Es razonable que lea sobre el indice_primario_puntos ya que está buscando tuplas sobre el campo puntos.

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla"
WHERE puntos < 200
```

Esta ejecución muestra que se han leído 122.647 bloques en una lectura secuencial de "MiTabla". Como tiene que buscar en un rango de valores, parece más óptimo hacer una lectura secuencial e ir apuntando los valores que cuadran con esa condición.

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla"
WHERE puntos > 30000
```

Tras la ejecución muestra que no se han leído ningún bloque en el escaneo del índice_primario_puntos ya que no se encuentra la condición que busca. No hay ninguna tupla con puntos mayores de 30000.

(En este punto he encontrado un error sobre el índice primario de puntos y lo he arreglado, ahora a partir de aquí se ha arreglado y arriba también está arreglado en el ejercicio correspondiente.)

---

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla"
WHERE id_cliente = 90000 OR puntos = 230
```

Esta ejecución muestra que se han leído 15.822 bloques. Es razonable ya que como solo tiene que buscar dos valores inicialmente busca en los índices de puntos y de id_cliente y posteriormente cuando sabe dónde se encuentran los valores y a continuación los busca en el archivo.

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla"
WHERE id_cliente = 90000 AND puntos = 230
```

Con esta ejecución no se ha leído ningún bloque ya que no hay una tupla que contenga esos valores, por lo que al buscar en el índice hash_id_cliente, el valor con id_cliente = 90.000 y el filtro puntos = 230 no encuentra ninguna tupla con dichos valores.

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

Para crear el índice se ejecuta el siguiente comando:

```
CREATE INDEX arbol_multiclave
ON "MiTabla"(puntos, nombre);
```

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:

Se ejecuta el siguiente comando antes de cada consulta para evitar que las estadísticas recolectadas con las mismas influyan en las siguientes consultas:

```
SELECT pg_stat_reset()
```

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla"
WHERE puntos = 200 AND nombre = 'nombre3456789'
```

No se ha leído ningún bloque ya que ha buscado en el índice árbol_multiclave y no ha encontrado ninguna tupla que contenga los valores buscados.

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla"
WHERE puntos = 200 OR nombre = 'nombre3456789'
```

Al ejecutar esta consulta se han leído 122.623 bloques. Se ha realizado una búsqueda secuencial pararela sobre MiTabla con los filtros de la cláusula WHERE, por ende y como indican los resultados, al leer secuencialmente un archivo se leen muchos bloques.

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla"
WHERE puntos = 6000 AND nombre = 'nombre3456789'
```

No se ha leído ningún bloque ya que ha buscado en el índice árbol_multiclave y no ha encontrado ninguna tupla que contenga los valores buscados.

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla"
WHERE puntos = 6000 OR nombre = 'nombre3456789'
```

Al ejecutar esta consulta se han leído 122.616 bloques. Se ha realizado una búsqueda secuencial pararela sobre MiTabla con los filtros de la cláusula WHERE, por ende y como indican los resultados, al leer secuencialmente un archivo se leen muchos bloques.

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:

Para crear la tabla MiTabla3 con 10 particiones sobre el campo puntos hemos ejecutado los siguientes comandos:

```
CREATE TABLE "MiTabla3" (
    id_cliente int,
    nombre text,
    apellidos text,
    direccion text,
    puntos int) PARTITION BY HASH (puntos);
CREATE TABLE "MiTabla3_0" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 0);
CREATE TABLE "MiTabla3_1" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 1);
CREATE TABLE "MiTabla3_2" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 2);
CREATE TABLE "MiTabla3_3" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 3);
CREATE TABLE "MiTabla3_4" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 4);
CREATE TABLE "MiTabla3_5" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 5);
CREATE TABLE "MiTabla3_6" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 6);
CREATE TABLE "MiTabla3_7" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 7);
CREATE TABLE "MiTabla3_8" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 8);
CREATE TABLE "MiTabla3_9" PARTITION OF "MiTabla3" FOR VALUES WITH (MODULUS 10, REMAINDER 9);
```

Se ejecuta el siguiente comando antes de cada consulta para evitar que las estadísticas recolectadas con las mismas influyan en las siguientes consultas:

```
SELECT pg_stat_reset()
```

1. Mostrar las tuplas cuyos puntos valen 200.

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla3"
WHERE puntos = 200
```

Tras la ejecución de esta secuencia, nos muestra que se ha leído secuencialmente la partición 1 de la tabla nueva leyendo así 15339 bloques en total, que son más o menos los totales de esa partición, como cabe esperar en una lectura secuencial.

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla3"
WHERE puntos = 200 OR puntos = 300
```

Se puede ver en los resultados de estos comandos que al estar organizadas las particiones por un hash de puntos busca secuencialmente en las dos primeras particiones de la tabla de forma secuencial. Leyendo así 31431 bloques que es aproximadamente la suma de bloques de las dos particiones.

3. Mostrar las tuplas cuyos puntos valen 200 o 202

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla3"
WHERE puntos = 200 OR puntos = 202
```

En este caso pasa lo mismo que en el anterior, pero simplemente cambian un poco los bloques leídos (30603) ya que los busca en las particiones 1 y 8 de forma secuencial.

4. Mostrar las tuplas cuyos puntos son > 500.

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla3"
WHERE puntos > 500
```

Tras la sucesión de esta consulta, nos muestra que ha leído en todas las particiones de la tabla de forma secuencial ya que necesita buscar en todos los cajones para encontrar la condición. Es por eso que ha leído todos los bloques totales 158937 para devolver la respuesta correspondiente.

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

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla3"
WHERE puntos > 500 AND puntos < 550
```

En este caso ocurre algo similar ya que si para los mayores que 500 ha mirado en toda la tabla, es normal o cabe esperar al menos, que para esta condición también

lo mire en todas las particiones, volviendo a leer, prácticamente los mismos bloques 158617.

6. Mostrar las tuplas cuyos puntos son 800

```
EXPLAIN (ANALYZE TRUE,BUFFERS TRUE,FORMAT text)
SELECT *
FROM "MiTabla3"
WHERE puntos = 800
```

Por último en esta query simplemente se miran los datos al estar organizados en un hash en la partición 6 de la tabla, leyendo el total de los bloques de la misma aproximadamente 17498.

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.

Como hemos visto a lo largo de toda esta práctica es que PostgresSQL es una gran aplicación para mantener una base de datos ya que procura en la mejor medida ocupar el menor tamaño posible y que haya cargado en memoria la menos cantidad de datos posible.

Esto es posible gracias a la gran optimización del mismo y del buen uso de las estadísticas del mismo, ya que si sabes que hay que leer antes de leerlo, no tienes que mirar toda la tabla o tablas para saber lo que busca.

También encontramos necesario la optimización de nuestra base de datos por parte del administrador, ya que dependiendo de la forma de almacenamiento o de la forma de indexación que hagas por tu parte, seguramente aunque ocupe algo más la base de datos en disco irá mejor ya que sabrá buscar de forma mas eficiente y óptima.

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.