

**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:** Laura Mambrilla Moreno

**DNI:**

**ALUMNO 2:**

**Nombre y Apellidos:** Isabel Blanco Martínez

**DNI:**

**Fecha:** 28/01/2020

**Profesor Responsable:** Óscar Gutiérrez Blanco\_\_\_\_\_

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

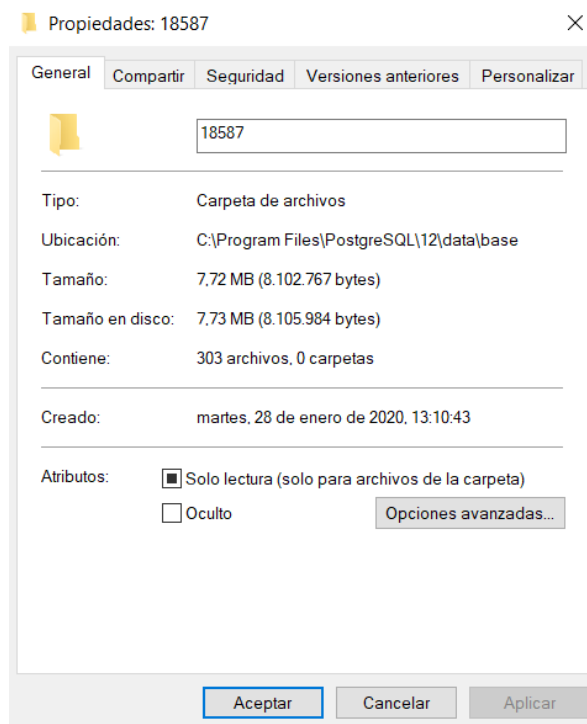
### Almacenamiento Físico en PostgreSQL

**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?

Nuestra base de datos se corresponde con la carpeta 18587 que se ha creado en el siguiente directorio:

Directorio → C:/Program Files/PostgreSQL/12/data/base

Tamaño → 7,72 MB



Archivos → Los archivos se crean en la carpeta de base. Sus nombres provienen del oid de la tabla contenida en pg\_catalog, el cual además puede tener coincidencias con relfilenode.

Nombre	Fecha de modificación	Tipo	Tamaño
112	28/01/2020 13:10	Archivo	8 KB
113	28/01/2020 13:10	Archivo	8 KB
174	28/01/2020 13:10	Archivo	8 KB
175	28/01/2020 13:10	Archivo	8 KB
548	28/01/2020 13:10	Archivo	8 KB
549	28/01/2020 13:10	Archivo	8 KB
826	28/01/2020 13:10	Archivo	0 KB
827	28/01/2020 13:10	Archivo	8 KB
828	28/01/2020 13:10	Archivo	8 KB
1247	28/01/2020 13:10	Archivo	80 KB
1247_fsm	28/01/2020 13:10	Archivo	24 KB
1247_vm	28/01/2020 13:10	Archivo	8 KB
1249	28/01/2020 13:10	Archivo	432 KB
1249_fsm	28/01/2020 13:10	Archivo	24 KB
1249_vm	28/01/2020 13:10	Archivo	8 KB
1255	28/01/2020 13:10	Archivo	632 KB
1255_fsm	28/01/2020 13:10	Archivo	24 KB
1255_vm	28/01/2020 13:10	Archivo	8 KB
1259	28/01/2020 13:10	Archivo	104 KB
1259_fsm	28/01/2020 13:10	Archivo	24 KB
1259_vm	28/01/2020 13:10	Archivo	8 KB
1417	28/01/2020 13:10	Archivo	0 KB
1418	28/01/2020 13:10	Archivo	0 KB
2187	28/01/2020 13:10	Archivo	8 KB
2224	28/01/2020 13:10	Archivo	0 KB
2328	28/01/2020 13:10	Archivo	0 KB
2336	28/01/2020 13:10	Archivo	0 KB
2337	28/01/2020 13:10	Archivo	8 KB

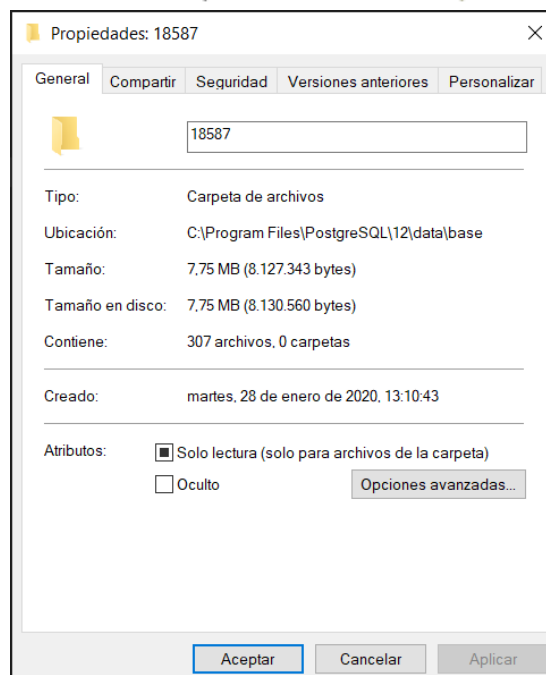
PostgreSQL Catalog (pg_catalog).pg_class/MiBaseDatos/postgres@PostgreSQL 12									
Query Editor   Query History									
1 <b>SELECT</b> * <b>FROM</b> pg_catalog.pg_class									
2									
Data Output   Explain   Messages   Notifications									
	oid oid	relname name	relnamespace oid	reltype oid	reloftype oid	relowner oid	relam oid	relfilenode oid	
1	2619	pg_statistic	11	12016	0	10	2	2619	
2	1247	pg_type	11	71	0	10	2	0	
3	4159	pg_toast_26...	99	12054	0	10	2	4159	
4	4160	pg_toast_26...	99	0	0	10	403	4160	

Al crear una base de datos, aunque no hayamos creado ninguna tabla todavía se crean una serie de ficheros que pertenecen al sistema y son necesarios para el correcto funcionamiento de la base de datos nueva que acabamos de crear (MiBaseDatos).

**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.

```
CREATE TABLE public.MiTabla(
    id_cliente integer NOT NULL,
    nombre text NOT NULL,
    apellidos text NOT NULL,
    direccion text NOT NULL,
    puntos integer NOT NULL,

    CONSTRAINT MiTabla_pk PRIMARY KEY (id_cliente));
```



Se han creado 4 archivos nuevos con la operación.

26779	04/02/2020 13:20	Archivo	0 KB
26782	04/02/2020 13:20	Archivo	0 KB
26784	04/02/2020 13:20	Archivo	8 KB
26785	04/02/2020 13:20	Archivo	8 KB

	oid	relname	relnamespace	reltype	reltoast	reloftype	relowner	relam	relfilenode	reltablespace	relpages	reltuples	relallvisible	reltoastrelid	relhasindex
1	26782	pg_toast_26...	99	26783	0	0	10	2	26782	0	0	0	0	0	true
2	26784	pg_toast_26...	99	0	0	0	10	403	26784	0	1	0	0	0	false
3	26779	mitabla	2200	26781	0	0	10	2	26779	0	0	0	0	26782	true
4	26785	mitabla_pk	2200	0	0	0	10	403	26785	0	1	0	0	0	false

El archivo 26782 ocupa 0 KB porque corresponde al integer creado.

El archivo 26779 ocupa 0 KB, corresponde a los datos de la tabla.

El archivo 26785 ocupa 8 KB y guarda los datos de la primary key.

El archivo 26784 ocupa 8 KB y corresponde los datos de los índices.

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

```
INSERT INTO public.mitabla(
    id_cliente, nombre, apellidos, direccion, puntos)
VALUES ('77777777A', 'Josefa', 'Garcia', 'La Garena', 555);
```

La tabla ahora ocupa lo mismo, 8 KB. Mientras que el archivo de la primary key ha aumentado a 16 KB, porque hemos introducido datos nuevos.

26779	04/02/2020 13:45	Archivo	8 KB
26782	04/02/2020 13:20	Archivo	0 KB
26784	04/02/2020 13:20	Archivo	8 KB
26785	04/02/2020 13:45	Archivo	16 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?

Creamos la extensión desde pgAdmin y después accedemos a la siguiente ruta para abrir el archivo que se ha creado:

C:\Program Files\PostgreSQL\12\share\extension

```
CREATE EXTENSION pg_buffercache;
```

Copiamos el contenido del archivo en pgAdmin y lo ejecutamos.

```

MiBaseDatos/postgres@PostgreSQL 12
Query Editor  Query History  Data Output  Explain  Messages  Notifications
1  /* contrib/pg_buffercache/pg_buffercache--1.0--1.1.sql */
2
3  -- complain if script is sourced in psql, rather than via ALTER EXTENSION echo Use "ALTER EXTENSION pg_buffercache UPDATE TO '1.1'"
4
5  -- Upgrade view to 1.1. format
6  CREATE OR REPLACE VIEW pg_buffercache AS
7      SELECT P.* FROM pg_buffercache_pages() AS P
8      (bufferid integer, relfilenode oid, reltablespace oid, reldatabase oid,
9       relforknumber int2, relblocknumber int8, isdirty bool, usagecount int2,
10      pinning_backends int4);

```

Ejecutamos una consulta para ver lo que ocupa el buffer para cada tabla de la base de datos

```

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;

```

	relname name	buffers bigint
1	pg_depend	59
2	pg_attribute	37
3	pg_proc	37
4	pg_class	17
5	pg_statistic	15
6	pg_operator	14
7	pg_depend_...	12
8	pg_proc_pro...	12
9	pg_proc_oid...	10
10	pg_rewrite	10


Es lógico lo que se muestra ya que se muestran los procesos que necesitan más espacio RAM del ordenador necesarios para que la base de datos se ejecute en segundo plano.

**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?

```
1 CREATE TABLE public.MiTabla(  
2     id_cliente text NOT NULL,  
3     nombre text NOT NULL,  
4     apellidos text NOT NULL,  
5     direccion text NOT NULL,  
6     puntos integer NOT NULL,  
7  
8     CONSTRAINT MiTabla_pk PRIMARY KEY (id_cliente)  
9 );
```



La información original a insertar ocupa 921.093 KB.




Nombre	Fecha de modificación	Tipo	Tamaño
 datos_mitabla	25/01/2020 0:15	Documento de tex...	921.093 KB

Antes de insertar los datos de la tabla:

 26793	11/02/2020 12:53	Archivo	0 KB
 26796	11/02/2020 12:53	Archivo	0 KB
 26798	11/02/2020 12:53	Archivo	8 KB
 26799	11/02/2020 12:53	Archivo	8 KB

Al insertar los datos:

 26796	11/02/2020 12:53	Archivo	0 KB
 26798	11/02/2020 12:53	Archivo	8 KB

 26799	11/02/2020 13:23	Archivo	582.968 KB
 26793_fsm	11/02/2020 13:23	Archivo	360 KB
 26793	11/02/2020 13:19	Archivo	1.048.576 KB

El archivo 26793, referente a mitabla, pasa a ocupar 1.048.576 KB lo cual equivale a 1 GB.

1

SELECT \* FROM public.mitabla

2

Data Output

Explain

Messages

Notifications

	id_cliente [PK] text	nombre text	apellidos text	direccion text	puntos integer
1	337	nombre337	apellidos337	apellidos337	59
2	9020289	nombre902...	apellidos902...	apellidos902...	624
3	14519336	nombre145...	apellidos145...	apellidos145...	132
4	7724637	nombre772...	apellidos772...	apellidos772...	278
5	3943592	nombre394...	apellidos394...	apellidos394...	520
6	1343216	nombre134...	apellidos134...	apellidos134...	392
7	8733445	nombre873...	apellidos873...	apellidos873...	566
8	8878781	nombre887...	apellidos887...	apellidos887...	358
9	13149508	nombre131	apellidos131	apellidos131	696

El archivo 26793 ocupa 1.048.576 KB, es decir, 1GB (1.048.576 / (1024 \*2) = 1GB).

Hemos calculado la longitud de cada campo de la tabla para conseguir el valor de Li (la i va de subíndice):

id\_cliente:

	max integer
1	9

En verdad la longitud de este campo es de 4 KB porque es un int, y lo que se ve en la captura es lo que ocupa el int parseado a text.

nombre:

apellido:

dirección:

	max integer
1	14

	max integer
1	17

	max integer
1	17

puntos:

	max integer
1	3

En verdad la longitud de este campo es de 4 bytes porque es un int, y lo que se ve en la captura es lo que ocupa el int parseado a text.



```
SELECT max(length(cast("puntos" as text)))
FROM public."mitabla"
```

```
SELECT max(length(cast("id_cliente" as text)))
FROM public."mitabla"
```

```
SELECT max(length("nombre"))
FROM public."mitabla"
```

```
SELECT max(length("apellidos"))
FROM public."mitabla"
```

```
SELECT max(length("direccion"))
FROM public."mitabla"
```

Hemos elegido max en vez de avg porque consideramos que sería más correcto, ya que así nos aseguramos que haya espacio suficiente.

Para saber el número de registros totales lo averiguamos con el siguiente query:

```
SELECT count(*)
FROM mitabla;
```

	count bigint
1	15000000

## MÉTODO TEÓRICO

Según las capturas sabemos la longitud en Bytes (L) de cada campo.

Lid\_cliente = 4 Bytes → porque el espacio máximo de un int en sql es 4

Lnombre= 14 Bytes

Lapellido= 17 Bytes

Ldireccion= 17 Bytes

Lpuntos= 4 Bytes → 4 bytes

Lr = Lid\_cliente + Lnombre + Lapellido + Ldireccion + Lpuntos = 56 Bytes

nr = 15000000

B → 8 KB = 8192 Bytes

$$f_R = \lfloor B / L_R \rfloor = 8192 / 56 = 146$$

$$b_R = \lceil n_R / f_R \rceil = 15000000 / 146 = 102740 \text{ bloques}$$

## MÉTODO FÍSICO

Dividimos el tamaño físico del archivo entre 8 KB, que es el tamaño de los bloques.

1048576 KB/ 8 (KB/bloque) = 131072 bloques

### MÉTODO REAL (SQL)

```
SELECT * FROM pg_class WHERE relname='mitabla'
```

	oid oid	relname name	relnamespace oid	reltype oid	relotype oid	relowner oid	relam oid	relfilenode oid	reltablespace oid	relpages integer	reltuples real
1	32777	mitabla	2200	32779	0	10	2	32777	0	159926	1.5e+07

*relpages* = número de bloques

159926 bloques

### ¿Concuerdan los resultados obtenidos de forma teórica o física con los resultados de postgres?

No, los resultados no concuerdan. El resultado obtenido en postgres es mucho más elevado que los bloques obtenidos de forma teórica. [ 159926 bloques > 131072 bloques > 102740 bloques ]. Estos resultados difieren debido a que:

- El método teórico solo tiene en cuenta los registros que introducimos.
- El método físico solo tiene en cuenta el tamaño real del archivo que introducimos (datos\_mitabla.txt).
- El método real nos indica el número exacto de los bloques que tiene la base de datos porque no cuenta únicamente con los registros ni con lo que ocupa solamente la tabla, ya que cuenta también con los archivos necesarios, como los índices de las Primary Keys.

**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?

	relname name	buffers bigint
1	mitabla_pk	15764
2	mitabla	215
3	pg_depend	59
4	pg_proc	22
5	pg_attribute	22
6	pg_class	17
7	pg_rewrite	15
8	pg_statistic	15
9	pg_type	14
10	pg_operator	13

La tabla nos muestra que al insertar todos los datos que contenía el archivo se necesitan: 15764 buffers en la tabla mitabla\_pk. Podemos deducir que obtenemos este resultado ya que para cargar todos los datos que teníamos que insertar se guardan en memoria. Estos datos incluyen sus respectivos registros cuyas PK se acumulan en la memoria también, por lo cuál el tamaño de mitabla\_pk aumenta considerablemente.

**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?

```
CREATE EXTENSION pgstattuple;
```

```
SELECT * FROM pgstattuple('pg_catalog.pg_proc');
```

En las estadísticas se muestra información relacionada con el espacio que ocupan las tuplas, devuelve la longitud física de una relación, el porcentaje de tuplas "muertas" y otra información.

	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
1	655360	2970	602786	91.98	6	2874	0.44	25000

El grado de ocupación de los bloques es:

tuple_percent	
double precision	
	91.98

Espacio libre:

free_space	free_percent
bigint	double precision
25000	3.81

El espacio libre que queda se debe a que no todos los bloques se han completado por completo, entonces ese es el espacio libre que queda.

**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.

Usamos ctid, que aparece en la documentación del system columns para ver las posiciones ocupadas por cada bloque.

```
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 media;
```

Obtenemos el siguiente resultado:

	avg
	numeric
1	93.7933794379900704

Cada bloque ocupa 8KB y como acabamos de conocer, cada uno está ocupado al 94 % aproximadamente. Entonces sabemos que los bloques ocupan 7700 Bytes.

Como cada registro ocupa 56 Bytes, obtenemos:

[7700 Bytes / 56 Bytes] = 137 registros/bloque.

Calculamos el factor de bloque:

$$L_r = L_{id\_cliente} + L_{nombre} + L_{apellido} + L_{direccion} + L_{puntos} = 56 \text{ Bytes}$$

$$f_R = \lfloor B/L_R \rfloor = 8192 / 56 = 146 \text{ registros/bloque}$$

El factor de bloque es diferente al resultado obtenido con 11 registros/bloque de diferencia.

**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?

```
CREATE EXTENSION pageinspect;
```

Tenemos 159926 bloques en total.

El 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	3/B09E06...	0	0	400	448	8192	8192	4	0

Bloque del medio es 79963:

```
SELECT * FROM page_header(get_raw_page('mitabla', 79963));
```

	lsn pg_lsn	checksum smallint	flags smallint	lower smallint	upper smallint	special smallint	pagesize smallint	version smallint	prune_xid xid
1	3/FF083E...	0	0	400	464	8192	8192	4	0

El último bloque:

```
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	6/311873...	0	0	128	5984	8192	8192	4	0

Los bloques difieren en los campos lower y upper. Estos contienen desplazamientos de bytes desde el inicio de la página hasta el inicio del espacio no asignado y hasta el final del espacio no asignado, respectivamente. Es decir, el espacio libre que tiene cada bloque.

Como podemos observar, la diferencia entre el primer bloque y el del medio es muy pequeña, ya que el primer bloque tiene  $448-400=48$  de espacio libre y el del medio tiene  $464-400=64$  de espacio libre porque lo que ocupan las tuplas es distinto y recordamos que los bloques no se ocupan al 100%, sino al 93,79%. Sin embargo, el último bloque tiene  $5984-128=5856$  de espacio libre debido a que es el último bloque y no se llena por completo.

**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?

Usaremos el comando CREATE INDEX ya que crea un árbol de tipo B.

```
CREATE INDEX arbol
ON mitabla (puntos);
```

41013

01/03/2020 11:00

Archivo

329.504 KB

Físicamente se almacena en: C:\Program Files\PostgreSQL\12\data\base\18587

En Postgres un bloque ocupa 8KB, por lo que tiene 41188 bloques, calculado por el método físico.

Por el método real, nos sale el mismo número de bloques:

```
SELECT * FROM pg_relpages('arbol');
```

	pg_relpages bigint
1	41188

Para saber lo referente al árbol:

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

	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

En la captura podemos ver que indica que hay 2 niveles, pero en verdad hay 3 porque no cuenta la raíz en la estadística.

Las hojas ocupan 40984 bloques, el nivel intermedio ocupa 203 bloques y la raíz ocupa un bloque, que corresponde al bloque 209.

Cada nivel tiene el siguiente número de tuplas:

$\text{live\_items} = \text{número de tuplas por nivel} - 1$

$\text{avg\_item\_size} = \text{número de tuplas de media por bloque en el nivel}$

Raíz:

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

	bikno 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

Número de tuplas en el nivel =  $202 + 1 = 203$

Número de tuplas de media por bloque en el nivel = 23

Nivel intermedio:

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

	bikno 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

Número de tuplas en el nivel =  $204 + 1 = 205$

Número de tuplas de media por bloque en el nivel = 23

Hojas:

```
SELECT * FROM bt_page_stats('arbol', 200);
```

	bikno 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	200	l	367	0	16	8192	800	199	201	0	1

Número de tuplas en el nivel =  $367 + 1 = 368$

Número de tuplas de media por bloque en el nivel = 16

**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.

Debemos contar con que posee cajones de punteros debido a que es un índice secundario con campo no clave.

El número de valores diferentes que tenga el campo puntos nos indicará el número de registros a indexar para este índice:

$N_{ri} = V(\text{puntos}) = 700$

Sabiendo que la longitud de un int es de 4 bytes y que, gracias a la siguiente consulta, el valor del tamaño de la tupla (avg\_item\_size) es de 23 bytes...

```
select * from bt_page_stats('arbol',617)
```

	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	617	i	184	0	23	8192	3012	413	821	1	0

... sabemos que la longitud del puntero a bloque es la resta del tamaño de la tupla menos la del tamaño del campo:

$$L_{pb} = 23 - 4 = 19 \text{ bytes}$$

La longitud del registro índice en la hoja del árbol B es de 16 bytes, que corresponden a la suma del campo puntos más la longitud del puntero a registro. Por tanto, para saber cuánto ocupa el tamaño de un puntero a registro, restamos el tamaño del campo.

$$L_{pr} = 16 - 4 = 12 \text{ bytes}$$

Tras esto ya podemos calcular cuántos punteros habrá en un nodo hoja.

Recordamos que posee cajones de punteros ya que es un índice secundario con campo no clave, por lo que los nodos hoja tienen punteros a bloque en vez de punteros a registro. Calculamos:

$$nh \cdot (L_{pr} + L_{puntos}) + L_{pb} \leq B$$

$$nh \cdot 16 + 19 \leq 8192$$

$$16 \cdot nh \leq 8192 - 19$$

$$16 \cdot nh \leq 510,8$$

$$nh = 510 \text{ punteros a bloque o valores de campo}$$

Sabiendo estos valores, calculamos el número de niveles que tiene el árbol B y los bloques que tienen cada uno de ellos:

- Nivel hoja

$$\text{num bloques} = \lceil \text{num registros} / nh \rceil = \lceil 15000000 / 510 \rceil = 29354,21 \text{ bloques}$$

$$29354,21 \text{ bloques} \rightarrow 29355 \text{ bloques}$$

- Nivel intermedio 1

$$\text{num bloques} = \lceil \text{num bloques nivel hoja} / n_{ri} \rceil = \lceil 29355 / 700 \rceil = 41,93 \text{ bloques}$$

$$41,93 \text{ bloques} \rightarrow 42 \text{ bloques}$$

- Raíz

$$\text{num bloques} = \lceil \text{num bloques nivel inter} / n_{ri} \rceil = \lceil 41,93 / 700 \rceil = 1 \text{ bloques}$$

**Cuestión 12.** Crear un índice de tipo hash para el campo id\_cliente y otro para el campo puntos.



Para el campo *id\_cliente*:

```
CREATE INDEX hash_idcliente
ON mitabla USING HASH (id_cliente);
```

Para el campo *puntos*:

```
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?

pageinspect:

```
SELECT * FROM hash_page_stats(get_raw_page('hash_idcliente', 1));
```

	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	203	0	8192	4088	49151	4294967295		0	65408

```
SELECT * FROM hash_page_stats(get_raw_page('hash_puntos', 1));
```

	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	0	0	8192	8148	49151	4294967295		0	65408

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	16240	1	0	15000000	0	43.72280235592257

```
select * from pgstathashindex('hash_puntos');
```

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

	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

Esto es coherente con el campo que muestra 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 racional 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

Al insertar información podemos comprobar que se modifican 7 archivos, los que hacen referencia a la primary key, integers e índices.

```
INSERT INTO public.mitabla(  
    id_cliente, nombre, apellidos, direccion, puntos)  
VALUES (338, 'Josefina', 'Barajas', 'UAH', 100);
```

#### MODIFICACIÓN

```
UPDATE public.mitabla2  
    SET nombre= 'Jose'  
    WHERE id_cliente= 338;
```

Al modificar la información se modifican los mismos 7 archivos.

#### BORRADO







```
DELETE FROM public.mitabla2  
    WHERE id_cliente= 338;
```

Físicamente las tuplas eliminadas están en standby, ya que lo que hemos hecho con el delete ha sido activar el flag que invisibiliza la tupla “eliminada”. Se modifican los mismos 7 archivos.

**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.

```
DELETE FROM mitabla
WHERE id_cliente IN
(SELECT id_cliente FROM mitabla ORDER BY random() LIMIT 2000000);
```

Se modifica el archivo referente a mitabla, sin embargo el tamaño de dicho archivo y de los índices no se reduce y se mantiene igual. Esto se podría deber a que los datos quedan temporalmente almacenados en un buffer al modificar los datos de la tabla.

 32777	01/03/2020 14:01	Archivo	1.048.576 KB
 2619	01/03/2020 14:01	Archivo	224 KB
 2619_fsm	01/03/2020 14:01	Archivo	24 KB
 2696	01/03/2020 14:01	Archivo	32 KB
 32777.1	01/03/2020 14:01	Archivo 1	230.832 KB
 1259	01/03/2020 14:01	Archivo	104 KB

**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.








La mejor forma en la que podemos optimizar nuestra base de datos usando el comando de VACUUM, que se encarga de eliminar las tuplas muertas que aparecen al eliminar o al tener tuplas obsoletas en tablas.

```
1 VACUUM (VERBOSE, ANALYZE) mitabla;
```

Data Output   Explain   Messages   Notifications

INFO: "mitabla": scanned 30000 of 174676 pages, containing 2237168 live rows and 0 dead rows; 30000 rows in sample, 13025985 estimated total rows  
VACUUM  
Query returned successfully in 3 secs 537 msec.









El número de tuplas muertas pasa a ser 0.

Nombre	Fecha de modificación	Tipo	Tamaño
 26847	28/02/2020 16:24	Archivo	685.000 KB
 26845	28/02/2020 16:24	Archivo	329.504 KB
 26846	28/02/2020 16:24	Archivo	523.152 KB
 26793	28/02/2020 16:21	Archivo	1.048.576 KB
 26793_vm	28/02/2020 16:21	Archivo	48 KB
 26799	28/02/2020 16:21	Archivo	582.968 KB
 26793.1	28/02/2020 16:20	Archivo 1	348.832 KB

**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.

```
CREATE TABLE public."mitabla2"(
    id_cliente integer NOT NULL,
    nombre text NOT NULL,
    apellidos text NOT NULL,
    direccion text NOT NULL,
    puntos integer NOT NULL,
    CONSTRAINT "mitabla2_pkey" PRIMARY KEY (id_cliente))
WITH (
    OIDS = FALSE,
    FILLFACTOR = 40
)
TABLESPACE pg_default;
```

Hemos creado una tabla llamada mitabla2 le hemos asignado el factor de bloque del 40%, y después hemos vuelto a importar los datos del archivo datos\_mitabla.txt.

 2619	01/03/2020 0:29	Archivo	216 KB
 2696	01/03/2020 0:29	Archivo	32 KB
 32854.3	01/03/2020 0:29	Archivo 3	96.072 KB
 1259	01/03/2020 0:29	Archivo	104 KB
 32854	01/03/2020 0:26	Archivo	1.048.576 KB
 32854.1	01/03/2020 0:26	Archivo 1	1.048.576 KB
 32854.2	01/03/2020 0:26	Archivo 2	1.048.576 KB
 32860	01/03/2020 0:22	Archivo	436.920 KB





Como se puede apreciar, en este caso mitabla2 ocupa más que mitabla y se almacena en diferentes archivos. Esto se debe a que al reducir la capacidad de llenado de las tuplas al 40% quepa menos información en cada bloque y necesite más bloques para almacenar todos los registros.

**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 (338, 'Josefina', 'Barajas', 'UAH', 100);
```

En este caso, podemos apreciar que se modifican 4 archivos. Hacen referencia a los integers y los índices.

 2619	01/03/2020 9:58	Archivo	216 KB
 32854.1	01/03/2020 9:58	Archivo 1	1.048.576 KB
 32854.3	01/03/2020 9:58	Archivo 3	96.072 KB
 32860	01/03/2020 9:58	Archivo	436.920 KB


### MODIFICACIÓN

```
UPDATE public.mitabla2
SET nombre= 'Jose'
WHERE id_cliente= 338;
```

El tamaño de los archivos se mantiene igual que en la imagen anterior.

### BORRADO

```
DELETE FROM public.mitabla2
WHERE id_cliente= 338;
```

 2619	01/03/2020 10:03	Archivo	216 KB
 32854.3	01/03/2020 10:03	Archivo 3	96.072 KB

Se modifican dos archivos de nuevo pero el tamaño del archivo se mantiene, ya que aunque eliminemos información siguen guardados en algún buffer temporalmente.

La conclusión que podemos obtener es que el tamaño del archivo aumenta al insertar tuplas pero se mantiene igual al modificarlos o al eliminarlos. Además, tarda más tiempo en realizar las operaciones en mitabla2 que en mitabla.

**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 consiste en dividir la tabla en partes más pequeñas, es decir, subdividir una tabla padre en varias tablas pequeñas hijas y vaciar la tabla padre utilizando la cláusula ONLY.

PostgreSQL ofrece soporte integrado para las siguientes formas de particionamiento:

- Particionamiento de rango

La tabla se divide en "rangos" definidos por una columna clave o un conjunto de columnas, sin superposición entre los rangos de valores asignados a diferentes particiones. Por

ejemplo, uno podría dividir por rangos de fechas o por rangos de identificadores para objetos comerciales particulares.

- Particionamiento de lista

La tabla se divide mediante una lista explícita de los valores clave que aparecen en cada partición.

- Particionamiento de hash

La tabla se divide especificando un módulo y un resto para cada partición. Cada partición contendrá las filas para las cuales el valor hash de la clave de partición dividido por el módulo especificado producirá el resto especificado.

Particionar una tabla es especialmente útil cuando esta es muy grande.















**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.

Creamos la tabla mitabla3 particionándola como HASH y le metemos los datos del archivo de *datos\_mitabla.txt*. Vamos a crear 10 particiones (cajones) con la fórmula  $x \bmod 10$ , ya que necesitamos 10 valores.

```
CREATE TABLE public.mitabla3(
    id_cliente integer NOT NULL,
    nombre text NOT NULL,
    apellidos text NOT NULL,
    direccion text NOT NULL,
    puntos integer NOT NULL)
PARTITION BY HASH (puntos);
```

```
CREATE TABLE public.mitabla3_0 PARTITION OF mitabla3 FOR VALUES WITH (modulus 10, remainder 0);
CREATE TABLE public.mitabla3_1 PARTITION OF mitabla3 FOR VALUES WITH (modulus 10, remainder 1);
CREATE TABLE public.mitabla3_2 PARTITION OF mitabla3 FOR VALUES WITH (modulus 10, remainder 2);
CREATE TABLE public.mitabla3_3 PARTITION OF mitabla3 FOR VALUES WITH (modulus 10, remainder 3);
CREATE TABLE public.mitabla3_4 PARTITION OF mitabla3 FOR VALUES WITH (modulus 10, remainder 4);
CREATE TABLE public.mitabla3_5 PARTITION OF mitabla3 FOR VALUES WITH (modulus 10, remainder 5);
CREATE TABLE public.mitabla3_6 PARTITION OF mitabla3 FOR VALUES WITH (modulus 10, remainder 6);
CREATE TABLE public.mitabla3_7 PARTITION OF mitabla3 FOR VALUES WITH (modulus 10, remainder 7);
CREATE TABLE public.mitabla3_8 PARTITION OF mitabla3 FOR VALUES WITH (modulus 10, remainder 8);
CREATE TABLE public.mitabla3_9 PARTITION OF mitabla3 FOR VALUES WITH (modulus 10, remainder 9);
```

Físicamente ocurre lo siguiente:

Nombre	Fecha de modifica...	Tipo	Tamaño
 32794	29/02/2020 21:09	Archivo	129.760 KB
 32806	29/02/2020 21:09	Archivo	133.456 KB
 32812	29/02/2020 21:09	Archivo	109.720 KB
 32818	29/02/2020 21:09	Archivo	153.600 KB
 32824	29/02/2020 21:09	Archivo	120.752 KB
 32830	29/02/2020 21:09	Archivo	140.752 KB
 32836	29/02/2020 21:09	Archivo	117.008 KB
 32842	29/02/2020 21:09	Archivo	124.160 KB
 32848	29/02/2020 21:09	Archivo	125.976 KB
 1259	29/02/2020 21:09	Archivo	104 KB
 2619	29/02/2020 21:09	Archivo	216 KB
 2619_fsm	29/02/2020 21:09	Archivo	24 KB
 2696	29/02/2020 21:09	Archivo	32 KB
 32800	29/02/2020 21:09	Archivo	124.248 KB

Como se puede observar se han creado 10 diferentes particiones (cajones) de tamaños similares y los archivos asociados a cada una.

**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.

- **PostgreSQL:**

En la partición 1 hay 16220 bloques:

```
SELECT * FROM pg_relpages('mitabla3_0');
```



	pg_relpages	
	bigint	
1	16220	

En la partición 2 hay 15531 bloques:

```
SELECT * FROM pg_relpages('mitabla3_1');
```

	pg_relpages	
	bigint	
1	15531	

En la partición 3 hay 16682 bloques:

```
SELECT * FROM pg_relpages('mitabla3_2');
```

	pg_relpages	
	bigint	
1	16682	

En la partición 4 hay 13715 bloques:

```
SELECT * FROM pg_relpages('mitabla3_3');
```

	pg_relpages	
	bigint	
1	13715	

En la partición 5 hay 19200 bloques:

```
SELECT * FROM pg_relpages('mitabla3_4');
```

	pg_relpages	
	bigint	
1	19200	

En la partición 6 hay 15094 bloques:

```
SELECT * FROM pg_relpages('mitabla3_5');
```

	pg_relpages	
	bigint	
1	15094	

En la partición 7 hay 15594 bloques:

```
SELECT * FROM pg_relpages('mitabla3_6');
```

	pg_relpages	
	bigint	
1		17594

En la partición 8 hay 14626 bloques:

```
SELECT * FROM pg_relpages('mitabla3_7');
```

	pg_relpages	
	bigint	
1		14626

En la partición 9 hay 15520 bloques:

```
SELECT * FROM pg_relpages('mitabla3_8');
```

	pg_relpages	
	bigint	
1		15520

En la partición 9 hay 15747 bloques:

```
SELECT * FROM pg_relpages('mitabla3_9');
```

	pg_relpages	
	bigint	
1		15747

- **Teóricamente:**

$$n_{RC} = n_R / N_C = 15000000 / 10 = 1500000 \text{ registros por cajón}$$

$$B_C = \lceil n_{rc} / f_{rc} \rceil = 1500000 / 146 = 10273,97 \rightarrow 10274 \text{ bloques}$$

### **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.

Podemos monitorizar la actividad de la base de datos de tres formas diferentes: usando el módulo *oid2name*, haciendo una inspección manual de system catalogs o usando las siguientes funciones SQL:

- **pg\_column\_size(any)**: Número de bytes utilizados para almacenar un valor.
- **pg\_database\_size(oid)**: Espacio en disco utilizado por la base de datos con el OID especificado.
- **pg\_database\_size(name)**: Espacio en disco utilizado por la base de datos con el nombre especificado.
- **pg\_indexes\_size(regclass)**: Espacio total en disco utilizado por los índices adjuntos a la tabla especificada.
- **pg\_relation\_size(relation regclass, fork text)**: Espacio en disco utilizado por el fork especificado ('main', 'fsm', 'vm' o 'init') de la tabla o índice especificado.
- **pg\_relation\_size(relation regclass)**: Forms corta de `pg_relation_size(..., 'main')`
- **pg\_size\_bytes(text)**: Convierte un tamaño en formato legible para personas en bytes.
- **pg\_size pretty(bigint)**: Convierte integers de 64-bits en en formato legible para personas.
- **pg\_size pretty(numeric)**: Convierte bytes expresados como un valor numérico en un formato legible por humanos.
- **pg\_table\_size(regclass)**: Espacio en disco utilizado por la tabla especificada, excluyendo índices (pero incluyendo TOAST, mapa de espacio libre y mapa de visibilidad).
- **pg\_tablespace\_size(oid)**: Espacio en disco utilizado por el tablespace con el OID especificado.
- **pg\_tablespace\_size(name)**: Espacio en disco utilizado por el tablespace con el nombre especificado.
- **pg\_total\_relation\_size(regclass)**: Espacio total en disco utilizado por la tabla especificada, incluidos todos los índices y datos TOAST.

Podemos recopilar información siempre y cuando haya sido previamente aplicado VACUUM en nuestra base de datos, o analizada.

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

```
CREATE INDEX btree_primario_puntos
ON mitabla (puntos ASC);
```

PostgreSQL, al crear un índice, por defecto hace que sea un árbol B primario. Por lo cual, primero debemos ordenar los datos de mitabla según el campo puntos y después haremos el índice.

**Cuestión 24.** Crear un índice hash sobre el campo puntos y otro sobre id\_cliente.

Para especificar que el índice sea un HASH pondremos using HASH.

Hash sobre el campo puntos:

```
CREATE INDEX hash_puntos  
ON mitabla using HASH (puntos);
```

Hash sobre el campo id\_cliente:

```
CREATE INDEX hash_idcliente  
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?


Índice primario btree (puntos):

Podemos ver que el tamaño del índice primario Árbol B sobre el campo puntos ocupa 329.504KB.

 32788	29/02/2020 20:31	Archivo	329.504 KB
--	------------------	---------	------------


Hash (puntos):

Podemos ver que el tamaño del índice hash sobre el campo puntos ocupa 685.000KB.

 32789	29/02/2020 20:41	Archivo	685.000 KB
---	------------------	---------	------------

Hash (id\_cliente):

Podemos ver que el tamaño del índice índice hash sobre el campo id\_cliente ocupa 393.232 KB.

 32790	29/02/2020 20:45	Archivo	393.232 KB
---	------------------	---------	------------

Podemos, por tanto, observar que por tamaño el índice óptimo para el campo puntos es el árbol B, ya que es el que menos ocupa. Además sabemos que el hash de puntos es el que más ocupa ya que es secundario no clave.

**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:

La información que podemos obtener de la base de datos son los siguientes:

- Estadísticas referentes a los índices de una base de datos específica (P.ej pg\_statio\_all\_indexes).
- Estadísticas referentes a cada secuencia específica que se produce en una base de datos específica (P.ej pg\_statio\_all\_sequences).

- Estadísticas referentes a las tablas y a los accesos a ellas mismas.(P.ej pg\_stat\_all\_tables).
- Finalmente, tenemos las estadísticas referentes a la base de datos en general. Usando pg\_stat\_database podemos acceder a la siguiente información útil:
  - **xact\_commit**: Transacciones realizadas por la base de datos.
  - **blks\_read**: Número de bloques leídos por la base de datos.
  - **blks\_hit**: Número de bloques que se encontraban en el buffer cache de postgresQL, los cuales no han sido necesarios leer de nuevo.
  - **tup\_returned**: Número de tuplas devueltas por las query.
  - **tup\_fetched**: Número de tuplas devueltas por las query.
  - **tup\_inserted**: Número de tuplas insertadas por las query.
  - **tup\_updated**: Número de tuplas actualizadas por las query.
  - **tup\_deleted**: Número de tuplas eliminadas por las query.
  - **temp\_files**: Número de archivos temporales creados por las query.
  - **temp\_bytes**: Tamaño de los archivos temporales creados por las query.
  - **blk\_read\_time**: Tiempo en milisegundos que tarda en leer los bloques.
  - **blk\_write\_time**: Tiempo en milisegundos que tarda en escribir los bloques.
  - **stats\_reset**: Última vez que han sido restablecidas las estadísticas.

El colector de estadísticas transmite la información recogida a través de archivos temporales. Estos archivos se almacenan en el directorio llamado por el parámetro stats\_temp\_directory parameter, pg\_stat\_tmp por defecto.

Para restaurar las estadísticas antes de realizar cada query usaremos pg\_stat\_reset().

```
SELECT pg_stat_reset() FROM pg_stat_database WHERE datname= 'MiBaseDatos';
```

Para ver obtener información acerca de las consultas utilizaremos blks\_read, blks\_hit, tup\_returned y blk\_read\_time.

```
SELECT blks_read, blks_hit, tup_returned, blk_read_time
FROM pg_stat_database WHERE datname= 'MiBaseDatos';
```

El resultado de ejecutar dicho código después de haber restablecido las estadísticas es el siguiente:




	blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
1	0	3	1	0

Como podemos ver, tendremos que tener en cuenta estos resultados para descontarlo de los resultados de cada query realizada a continuación.

## 1. Mostrar la información de las tuplas con id\_cliente=8.101.000.

```
SELECT *
FROM public."mitabla"
WHERE id_cliente=8101000;
```

Ejecutamos de nuevo la consulta para ver las estadísticas anteriormente mencionadas:


	 blks_read bigint	 blks_hit bigint	 tup_returned bigint	 blk_read_time double precision
1	78	910	3566	0

Como podemos ver, se han leído un total de 78 bloques. 910 bloques almacenados en buffer.

Para responder a cómo se ha realizado el código de la consulta usaremos el comando EXPLAIN:

```
EXPLAIN SELECT *
FROM public."mitabla"
WHERE id_cliente=8101000;
```

Nos da la siguiente información:

	QUERY PLAN	
	 text	
1	Index Scan using hash_idcliente on mitabla (cost=0.00..8.02 rows=1 width=56)	
2	Index Cond: (id_cliente = 8101000)	

Usa el índice hash\_idcliente para buscar el registro en el cual id\_cliente = 8101000;

## 2. Mostrar la información de las tuplas con id\_cliente <30000.

```
SELECT *
FROM public."mitabla"
WHERE id_cliente<30000;
```

	<div>blks_read</div> <div>bigint</div>	<div>blks_hit</div> <div>bigint</div>	<div>tup_returned</div> <div>bigint</div>	<div>blk_read_time</div> <div>double precision</div>
1	27409	115	31896	0

Como podemos ver, se han leído un total de 27409 bloques. 115 bloques almacenados en buffer.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *
FROM public."mitabla"
WHERE id_cliente<30000;
```

	QUERY PLAN	
	text	
1	Bitmap Heap Scan on mitabla (cost=654.15..75078.72 rows=29125 width=56)	
2	Recheck Cond: (id_cliente < 30000)	
3	-> Bitmap Index Scan on mitabla_pk (cost=0.00..646.87 rows=29125 width=0)	
4	Index Cond: (id_cliente < 30000)	

Lee mitabla y busca la concordancia con la condición, que es que el id\_cliente sea menor que 30000.

### 3. Mostrar el número de tuplas cuyo id\_cliente >8000 y id\_cliente <100000.

```
SELECT *
FROM public."mitabla"
WHERE (id_cliente>8000 AND id_cliente<100000);
```

	blks_read	blks_hit	tup_returned	blk_read_time
	bigint	bigint	bigint	double precision
1	70508	497	94150	0

Como podemos ver, se han leído un total de 70508 bloques. 497 bloques almacenados en buffer.

Usando EXPLAIN vemos:





```
EXPLAIN SELECT *
FROM public."mitabla"
WHERE (id_cliente>8000 AND id_cliente<100000);
```

	QUERY PLAN	
	text	
1	Gather (cost=3219.92..203045.80 rows=89316 width=56)	
2	Workers Planned: 2	
3	-> Parallel Bitmap Heap Scan on mitabla (cost=2219.92..193114.20 rows=37215 width=56)	
4	Recheck Cond: ((id_cliente > 8000) AND (id_cliente < 100000))	
5	-> Bitmap Index Scan on mitabla_pk (cost=0.00..2197.59 rows=89316 width=0)	
6	Index Cond: ((id_cliente > 8000) AND (id_cliente < 100000))	
7	JIT:	
8	Functions: 2	
9	Options: Inlining false, Optimization false, Expressions true, Deforming true	

Lee mitabla y comprueba que se cumplan las dos condiciones: id\_cliente>8000 e id\_cliente<100000.

#### 4. Mostrar la información de las tuplas con id\_cliente=34500 o id\_cliente=30.204.000.



```
SELECT *  
FROM public."mitabla"  
WHERE (id_cliente=34500 OR id_cliente=30204000);
```

	 blks_read bigint	 blks_hit bigint	 tup_returned bigint	 blk_read_time double precision
1	5	102	1894	0

Como podemos ver, se han leído un total de 5 bloques. 102 bloques almacenados en buffer.

Usando EXPLAIN vemos:





```
EXPLAIN SELECT *  
FROM public."mitabla"  
WHERE (id_cliente=34500 OR id_cliente=30204000);
```

	QUERY PLAN	
	 text	
1	Bitmap Heap Scan on mitabla (cost=8.02..16.02 rows=2 width=56)	
2	Recheck Cond: ((id_cliente = 34500) OR (id_cliente = 30204000))	
3	-> BitmapOr (cost=8.02..8.02 rows=2 width=0)	
4	-> Bitmap Index Scan on hash_idcliente (cost=0.00..4.01 rows=1 width=0)	
5	Index Cond: (id_cliente = 34500)	
6	-> Bitmap Index Scan on hash_idcliente (cost=0.00..4.01 rows=1 width=0)	
7	Index Cond: (id_cliente = 30204000)	

Se realizan dos búsquedas paralelas en el índice hash\_idcliente para buscar id\_cliente=34500 y id\_cliente=30204000. Después recorre el archivo de datos hasta encontrar alguno de los dos registros.

#### 5. Mostrar las tuplas cuyo id\_cliente es distinto de 3450000.

```
SELECT *FROM public."mitabla"  
WHERE NOT id_cliente= 3450000;
```

	 blks_read bigint	 blks_hit bigint	 tup_returned bigint	 blk_read_time double precision
1	143727	16380	15003157	0

Como podemos ver, se han leído un total de 143727 bloques. 16380 bloques almacenados en buffer.

Usando EXPLAIN vemos:



```
EXPLAIN SELECT *
FROM public."mitabla"
WHERE NOT id_cliente= 3450000;
```

QUERY PLAN		
	text	
1	Seq Scan on mitabla (cost=0.00..347426.00 rows=14999999 width=56)	
2	Filter: (id_cliente <> 3450000)	
3	JIT:	
4	Functions: 2	
5	Options: Inlining false, Optimization false, Expressions true, Deforming true	

En este caso, escanea directamente el archivo de mitabla hasta que encuentra el id\_cliente 3450000.

## 6. Mostrar las tuplas que tiene un nombre igual a 'nombre3456789'.

```
SELECT *FROM public."mitabla"
WHERE nombre = 'nombre3456789';
```

	blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
1	143791	16485	15002035	0

Como podemos ver, se han leído un total de 143791 bloques. 16485 bloques almacenados en buffer.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *
FROM public."mitabla"
WHERE nombre = 'nombre3456789';
```

QUERY PLAN		
	text	
1	Gather (cost=1000.00..239051.10 rows=1 width=56)	
2	Workers Planned: 2	
3	-> Parallel Seq Scan on mitabla (cost=0.00..238051.00 rows=1 width=56)	
4	Filter: (nombre = 'nombre3456789':text)	
5	JIT:	
6	Functions: 2	
7	Options: Inlining false, Optimization false, Expressions true, Deforming true	

Lee la tabla y filtra los registros cuyo nombre es "nombre3456789", mediante una búsqueda secuencial paralela en mitabla.

## 7. Mostrar la información de las tuplas con puntos=650.

```
SELECT *  
FROM public."mitabla"  
WHERE puntos = 650;
```

blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
20108	107	23256	0

Como podemos ver, se han leído un total de 20108 bloques. 107 bloques almacenados en buffer.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *  
FROM public."mitabla"  
WHERE puntos = 650;
```

QUERY PLAN		
	text	
1	Bitmap Heap Scan on mitabla (cost=401.46..59362.77 rows=21293 width=56)	
2	Recheck Cond: (puntos = 650)	
3	-> Bitmap Index Scan on btree_primario_puntos (cost=0.00..396.13 rows=21293 width=0)	
4	Index Cond: (puntos = 650)	

En este caso utiliza el índice btree para localizar puntos=650 y después de haberlo localizado accede al archivo de datos para obtener dicho registro.

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

```
SELECT *  
FROM public."mitabla"  
WHERE puntos < 200;
```

blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
143727	16303	15001893	0

Como podemos ver, se han leído un total de 143727 bloques. 16303 bloques se encuentran en buffers.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *
FROM public."mitabla"
WHERE puntos < 200;
```

	QUERY PLAN	
	text	🔒
1	Seq Scan on mitabla (cost=0.00..347426.00 rows=4246074 width=56)	
2	Filter: (puntos < 200)	
3	JIT:	
4	Functions: 2	
5	Options: Inlining false, Optimization false, Expressions true, Deforming true	

Busca los registros que coinciden con la condición de que puntos sea menor que 200.

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

```
SELECT *
FROM public."mitabla"
WHERE puntos > 30000;
```

	blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
1	3	108	1894	0

Como podemos ver, se han leído un total de 3 bloques. 108 bloques acumulados en buffers.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *
FROM public."mitabla"
WHERE puntos > 30000;
```

	QUERY PLAN	
	text	🔒
1	Index Scan using btree_primario_puntos on mitabla (cost=0.43..8.45 rows=1 width=56)	
2	Index Cond: (puntos > 30000)	

Buscamos en el índice primario árbol b con la condición de puntos > 30000 en mitabla.

## 10. Mostrar la información de las tuplas con id\_cliente=90000 o puntos=230

```
SELECT *
FROM public."mitabla"
WHERE (puntos = 30000 OR id_cliente=90000);
```

	blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
1	2	184	3158	0

Como podemos ver, se han leído un total de 2 bloques. 184 bloques acumulados en buffer.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *
FROM public."mitabla"
WHERE (puntos = 30000 OR id_cliente=90000);
```

	QUERY PLAN
1	Bitmap Heap Scan on mitabla (cost=410.79..59427.76 rows=21294 width=56)
2	Recheck Cond: ((puntos = 30000) OR (id_cliente = 90000))
3	-> BitmapOr (cost=410.79..410.79 rows=21294 width=0)
4	-> Bitmap Index Scan on btree_primario_puntos (cost=0.00..396.13 rows=21293 width=0)
5	Index Cond: (puntos = 30000)
6	-> Bitmap Index Scan on hash_idcliente (cost=0.00..4.01 rows=1 width=0)
7	Index Cond: (id_cliente = 90000)

En este caso utiliza el índice btree para localizar puntos=30000 y el índice hash para encontrar el id\_cliente=90000. Después de haberlo localizado accede al archivo de datos para obtener dicho registro.

## 11. Mostrar la información de las tuplas con id\_cliente=90000 y puntos=230

```
SELECT *
FROM public."mitabla"
WHERE (puntos = 30000 AND id_cliente=90000);
```

	blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
1	0	106	1894	0

Como podemos ver, se han leído un total de 0 bloques. Esto se debe a que no encuentra datos que cumplan ambas condiciones. 106 bloques almacenados en buffer.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *
FROM public."mitabla"
WHERE (puntos = 30000 AND id_cliente=90000);
```

QUERY PLAN		
	text	
1	Index Scan using hash_idcliente on mitabla (cost=0.00..8.02 rows=1 width=56)	
2	Index Cond: (id_cliente = 90000)	
3	Filter: (puntos = 30000)	

Lee mitabla y busca lo que cunpla con las dos condiciones: id\_cliente=90000 y puntos=30000.

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

```
CREATE INDEX btree_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:

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

```
SELECT *
FROM mitabla
WHERE id_cliente=8101000;
```

	blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
1	5	32	427	Read-0

Como podemos ver, se han leído un total de 5 bloques. 32 bloques almacenados en buffer.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *
FROM public."mitabla"
WHERE (puntos = 200 AND nombre = 'nombre3456789');
```

	QUERY PLAN	
	text	
1	Index Scan using btree_multiclave on mitabla (cost=0.56..8.58 rows=1 width=56) (actual time=0.013..0.013 rows=0 loops=1)	
2	Index Cond: ((puntos = 200) AND (nombre = 'nombre3456789'::text))	
3	Planning Time: 0.056 ms	
4	Execution Time: 0.030 ms	

En este caso utiliza el índice árbol B primario hasta encontrar el registro que cumple ambas condiciones.

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

```
SELECT *
FROM public."mitabla"
WHERE (puntos = 200 OR nombre = 'nombre3456789');
```

	blks_read	blks_hit	tup_returned	blk_read_time	
	bigint	bigint	bigint	double precision	
1	143927	16355	15002033	0	

Como podemos ver, se han leído un total de 143927 bloques. 16355 bloques almacenados en buffer.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *
FROM public."mitabla"
WHERE (puntos = 200 OR nombre = 'nombre3456789');
```

	QUERY PLAN	
	text	
1	Gather (cost=1000.00..256805.40 rows=21294 width=56)	
2	Workers Planned: 2	
3	-> Parallel Seq Scan on mitabla (cost=0.00..253676.00 rows=8872 width=56)	
4	Filter: ((puntos = 200) OR (nombre = 'nombre3456789'::text))	
5	JIT:	
6	Functions: 2	
7	Options: Inlining false, Optimization false, Expressions true, Deforming true	

En esta query se realiza una secuencia paralela hasta que encuentra una de las dos condiciones, y una vez la encuentra accede al archivo de datos directamente.

- Mostrar las tuplas cuyo id\_cliente vale 6000 o su nombre es nombre3456789.

```
SELECT *
FROM public."mitabla"
WHERE (id_cliente = 6000 OR nombre = 'nombre3456789');
```

	blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
1	143916	16366	15002033	0

Como podemos ver, se han leído un total de 143916 bloques. 16366 bloques almacenados en buffer.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *
FROM public."mitabla"
WHERE (id_cliente = 6000 OR nombre = 'nombre3456789');
```

	QUERY PLAN
1	Gather (cost=1000.00..254676.20 rows=2 width=56)
2	Workers Planned: 2
3	-> Parallel Seq Scan on mitabla (cost=0.00..253676.00 rows=1 width=56)
4	Filter: ((id_cliente = 6000) OR (nombre = 'nombre3456789'::text))
5	JIT:
6	Functions: 2
7	Options: Inlining false, Optimization false, Expressions true, Deforming true

Lee mitabla y comprueba que cumpla una de las dos condiciones: o que id\_cliente=6000 o que nombre='nombre3456789'.

- **Mostrar las tuplas cuyo id\_cliente vale 6000 y su nombre es nombre3456789.**

```
SELECT *
FROM public."mitabla"
WHERE (nombre = 'nombre3456789' AND id_cliente=90000);
```

	blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
1	3	105	1890	0

Como podemos ver, se han leído un total de 3 bloques. 105 bloques almacenados en buffer.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *
FROM public."mitabla"
WHERE (nombre = 'nombre3456789' AND id_cliente=90000);
```

	QUERY PLAN	
	text	
1	Index Scan using mitabla_pk on mitabla (cost=0.43..8.46 rows=1 width=56)	
2	Index Cond: (id_cliente = 90000)	
3	Filter: (nombre = 'nombre3456789'::text)	

Comprueba que se cumpla nombre='nombre3456789' e id\_cliente=90000.

**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:

1. Mostrar las tuplas cuyos puntos valen 200.

```
SELECT *
FROM public."mitabla3"
WHERE puntos = 200;
```

	blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
1	14000	1880	1458920	0

Como podemos ver, se han leído un total de 14000 bloques. 1880 bloques almacenados en buffer.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *
FROM mitabla3
WHERE puntos=200;
```

	QUERY PLAN	
	text	
1	Gather (cost=1000.00..26308.24 rows=21900 width=56)	
2	Workers Planned: 2	
3	-> Parallel Seq Scan on mitabla3_1 (cost=0.00..23118.24 rows=9125 width=56)	
4	Filter: (puntos = 200)	

En este query, se realiza una búsqueda paralela has



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

En este caso hay una contradicción ya que es imposible que tenga dos valores al mismo tiempo. Por lo tanto, ejecutamos la query como OR en vez de AND.

```
SELECT *  
FROM public."mitabla3"  
WHERE (puntos = 200 OR puntos = 300);
```

	blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
1	28529	3589	2980233	0

Como podemos ver, se han leído un total de 28529 bloques. 3589 bloques almacenados en buffer.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *  
FROM mitabla3  
WHERE (puntos = 200 OR puntos = 300);
```

	QUERY PLAN
	text
1	Gather (cost=1000.00..55846.43 rows=43909 width=56)
2	Workers Planned: 2
3	-> Parallel Append (cost=0.00..50455.53 rows=18295 width=56)
4	-> Parallel Seq Scan on mitabla3_0 (cost=0.00..25728.37 rows=9170 width=56)
5	Filter: ((puntos = 200) OR (puntos = 300))
6	-> Parallel Seq Scan on mitabla3_1 (cost=0.00..24635.69 rows=9125 width=56)
7	Filter: ((puntos = 200) OR (puntos = 300))

Lee la tupla y busca registros que coincidan con las condiciones de puntos=200 o puntos=300, cada uno de ellos está en particiones diferentes, en este caso el primer y segundo cajón.

## 3. Mostrar las tuplas cuyos puntos valen 200 o 202.

```
SELECT *  
FROM public."mitabla3"  
WHERE (puntos = 200 OR puntos = 202);
```

	blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
1	27894	3510	2914482	0

Como podemos ver, se han leído un total de 27894 bloques. 3510 bloques almacenados en buffer.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *  
FROM mitabla3  
WHERE (puntos = 200 OR puntos = 202);
```

QUERY PLAN	
	text
1	Gather (cost=1000.00..50272.39 rows=31344 width=71)
2	Workers Planned: 2
3	-> Parallel Append (cost=0.00..46137.99 rows=13060 width=71)
4	-> Parallel Seq Scan on mitabla3_1 (cost=0.00..24635.69 rows=9125 width=56)
5	Filter: ((puntos = 200) OR (puntos = 202))
6	-> Parallel Seq Scan on mitabla3_8 (cost=0.00..21437.00 rows=3935 width=104)
7	Filter: ((puntos = 200) OR (puntos = 202))

Lee la tupla y busca registros que coincidan con las condiciones de puntos=200 o puntos=202, cada uno de ellos está en particiones diferentes, en este caso segundo y noveno cajón.

#### 4. Mostrar las tuplas cuyos puntos son > 500.

```
SELECT *  
FROM public."mitabla3"  
WHERE puntos>500;
```

	blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
1	144168	16095	15003447	0

Como podemos ver, se han leído un total de 144168 bloques. 16095 bloques almacenados en buffer.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *  
FROM mitabla3  
WHERE puntos>500
```

	QUERY PLAN
	text
1	Append (cost=0.00..335033.13 rows=3962228 width=75)
2	-> Seq Scan on mitabla3_0 (cost=0.00..35236.74 rows=368265 width=56)
3	Filter: (puntos > 500)
4	-> Seq Scan on mitabla3_1 (cost=0.00..33740.38 rows=521031 width=56)
5	Filter: (puntos > 500)
6	-> Seq Scan on mitabla3_2 (cost=0.00..36240.80 rows=448966 width=56)
7	Filter: (puntos > 500)
8	-> Seq Scan on mitabla3_3 (cost=0.00..29795.00 rows=534413 width=56)
9	Filter: (puntos > 500)
10	-> Seq Scan on mitabla3_4 (cost=0.00..41710.06 rows=491740 width=56)
11	Filter: (puntos > 500)
12	-> Seq Scan on mitabla3_5 (cost=0.00..26603.18 rows=306911 width=104)
13	Filter: (puntos > 500)
14	-> Seq Scan on mitabla3_6 (cost=0.00..31009.43 rows=357745 width=104)
15	Filter: (puntos > 500)
16	-> Seq Scan on mitabla3_7 (cost=0.00..25778.33 rows=297395 width=104)
17	Filter: (puntos > 500)
18	-> Seq Scan on mitabla3_8 (cost=0.00..27354.00 rows=315573 width=104)
19	Filter: (puntos > 500)
20	-> Seq Scan on mitabla3_9 (cost=0.00..27754.09 rows=320189 width=104)
21	Filter: (puntos > 500)
22	JIT:
23	Functions: 20
24	Options: Inlining false, Optimization false, Expressions true, Deforming true

Esta búsqueda es bastante más amplia, por lo cual al considerarse tantos registros, se realiza una búsqueda secuencial de muchos de los datos. Empieza por el cajón 10, después la 8, 7, 6 y así progresivamente hasta el primer cajón.

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

```
SELECT *
FROM public."mitabla3"
WHERE (puntos > 500 AND puntos < 550);
```

	blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
1	144182	16236	15002286	0

Como podemos ver, se han leído un total de 144182 bloques. 16236 bloques almacenados en buffer.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *
FROM mitabla3
WHERE (puntos > 500 AND puntos < 550);
```

	QUERY PLAN
	text
1	Gather (cost=1000.00..292993.97 rows=533079 width=59)
2	Workers Planned: 2
3	-> Parallel Append (cost=0.00..238686.07 rows=222116 width=59)
4	-> Parallel Seq Scan on mitabla3_4 (cost=0.00..30455.03 rows=53074 width=56)
5	Filter: ((puntos > 500) AND (puntos < 550))
6	-> Parallel Seq Scan on mitabla3_2 (cost=0.00..26461.40 rows=43138 width=56)
7	Filter: ((puntos > 500) AND (puntos < 550))
8	-> Parallel Seq Scan on mitabla3_0 (cost=0.00..25728.37 rows=17432 width=56)
9	Filter: ((puntos > 500) AND (puntos < 550))
10	-> Parallel Seq Scan on mitabla3_1 (cost=0.00..24635.69 rows=62681 width=56)
11	Filter: ((puntos > 500) AND (puntos < 550))
12	-> Parallel Seq Scan on mitabla3_6 (cost=0.00..24301.71 rows=2236 width=104)
13	Filter: ((puntos > 500) AND (puntos < 550))
14	-> Parallel Seq Scan on mitabla3_3 (cost=0.00..21755.00 rows=35805 width=56)
15	Filter: ((puntos > 500) AND (puntos < 550))
16	-> Parallel Seq Scan on mitabla3_9 (cost=0.00..21750.54 rows=2001 width=104)
17	Filter: ((puntos > 500) AND (puntos < 550))
18	-> Parallel Seq Scan on mitabla3_8 (cost=0.00..21437.00 rows=1972 width=104)
19	Filter: ((puntos > 500) AND (puntos < 550))
20	-> Parallel Seq Scan on mitabla3_5 (cost=0.00..20848.59 rows=1918 width=104)
21	Filter: ((puntos > 500) AND (puntos < 550))
22	-> Parallel Seq Scan on mitabla3_7 (cost=0.00..20202.16 rows=1859 width=104)
23	Filter: ((puntos > 500) AND (puntos < 550))
24	JIT:
25	Functions: 20
26	Options: Inlining false, Optimization false, Expressions true, Deforming true

En este caso también tiene que considerar muchos datos pero la diferencia con la query anterior es que accede a los cajones de forma paralela, buscando con puntos entre 500 y 550.

## 6. Mostrar las tuplas cuyos puntos son 800


```
SELECT *
FROM public."mitabla3"
WHERE puntos = 800;
```

	blks_read bigint	blks_hit bigint	tup_returned bigint	blk_read_time double precision
1	16011	1906	1652328	0

Como podemos ver, se han leído un total de 16011 bloques. 1906 bloques almacenados en buffer.

Usando EXPLAIN vemos:

```
EXPLAIN SELECT *
FROM mitabla3
WHERE puntos = 800;
```

	QUERY PLAN	
	text	
1	Gather (cost=1000.00..24720.36 rows=5366 width=104)	
2	Workers Planned: 2	
3	-> Parallel Seq Scan on mitabla3_6 (cost=0.00..23183.76 rows=2236 width=104)	
4	Filter: (puntos = 800)	

En esta query, se accede a la partición (cajón) número 7 y de forma paralela en dicha partición hasta que encuentra puntos=800.

**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.

Las consultas de PostgreSQL nos facilitan el saber cosas relacionadas con la base de datos con la que estamos trabajando, desde ordenar los registros de una manera u otra hasta encontrar una tupla en concreto leyendo el mínimo número de bloques.

Lo que dimos en teoría lo hemos podido ver aplicado a una base de datos “real” de una manera más sencilla y dinámica gracias a las consultas, que vienen muy bien explicadas en la documentación de PostgreSQL 12 , por lo que su uso es muy sencillo.

## 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: PostgreSQL Client Applications.
- Capítulo VI.III: PostgreSQL Server Applications.
- Capítulo 50: System Catalogs.
- Capítulo 68: Database Physical Storage.
- Apéndice F: Additional Supplied Modules.
- Apéndice G: Additional Supplied Programs.