

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: David Ramos Fernández

ALUMNO 2:

Nombre y Apellidos: Sergio Sánchez Campo

Fecha: 02/03/2020

Profesor Responsable: Santiago Hermira Anchuelo

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

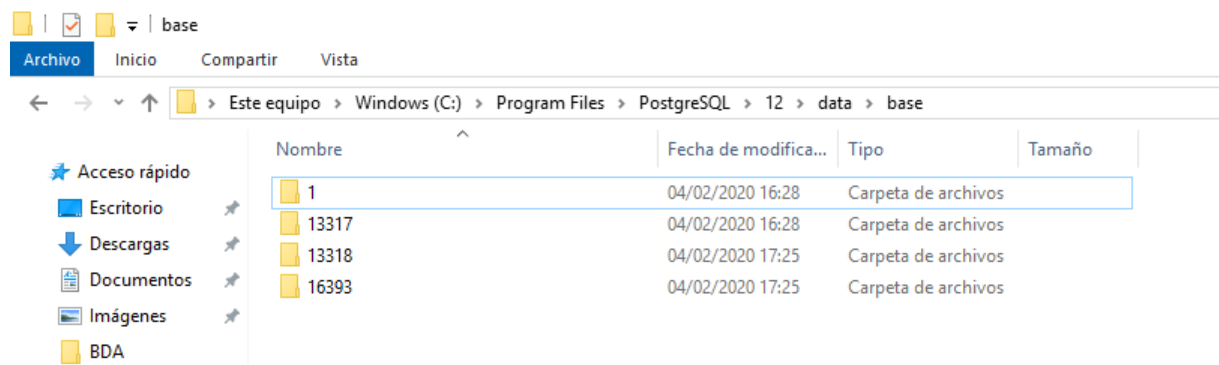
En esta primera práctica se introduce el sistema gestor de bases de datos **PostgreSQL versión 11 o 12**. Está compuesto básicamente de un motor servidor y de una serie de clientes que acceden al servidor y de otras herramientas externas. En esta primera práctica se entrará a fondo en la arquitectura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

Actividades y Cuestiones

Almacenamiento Físico en PostgreSQL

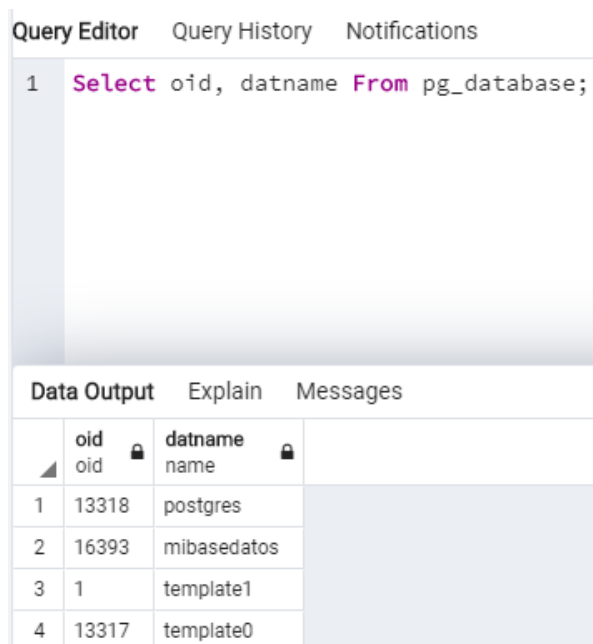
Cuestión 1. Crear una nueva Base de Datos que se llame **MiBaseDatos**. ¿En qué directorio se crea del disco duro, cuanto ocupa el mismo y qué ficheros se crean? ¿Por qué?

En la instalación de PostgreSQL se define una dirección donde se guardarán todos los archivos. En nuestro caso se ha instalado en la dirección por defecto que viene en el programa de instalación. Dentro de esta dirección encontramos subdirecciones con el nombre de las versiones de PostgreSQL instaladas, en nuestro caso solo tenemos una versión instalada la 12 por lo que solo tendremos el archivo **12**. Dentro de este archivo se encuentran todos los datos del servidor de los cuales en la información de la base de datos se encuentra en la dirección **data/base**, que es la dirección donde se almacenan todas los **oid** bases de datos del servidor.



Dirección: C:\Program Files\PostgreSQL\12\data\base

Para saber cuál es el subdirectorio de nuestra base de datos creada, **MiBaseDatos**, realizamos la siguiente consulta:



De esta manera podemos saber que **oid** tiene nuestra base de datos y por tanto la direccion donde se almacenarán los datos de esta. En nuestro caso tiene el **oid: 16393**.

En el directorio /base está formado por cuatro subdirectorios: 1, 13317, 13318, 16393 que son los **oid** de **template1**, **templateo**, **postgres** y **MiBaseDatos** correspondientemente. **template1** y **templateo** son unas bases de datos que utiliza el sistema como apoyo para crear nuevas bases de datos.

- **template1**: permite agregar nuevos objetos como tablas, tipos de datos a las bases de datos existentes.
- **templateo**: esta base de datos contiene los mismos datos iniciales que **template1**, pero no pueden ser modificados ya que esta actuaría como una copia de seguridad por si existiera un error grave poder volver al punto de inicio.
- **postgres**: es una base de datos que crea el sistema por defecto.

Para consultar el tamaño de los archivos realizamos la siguiente consulta:

Query Editor Query History Notifications

1 **Select** pg_database.datname **As** nombre,

2 pg_database_size(pg_database.datname) **As** bytes

3 **From** pg_database

Data Output Explain Messages

	<div>nombre</div> <div>name</div>	<div>bytes</div> <div>bigint</div>	
1	postgres	8110959	
2	mibasedatos	8102767	
3	template1	7954947	
4	template0	7954947	

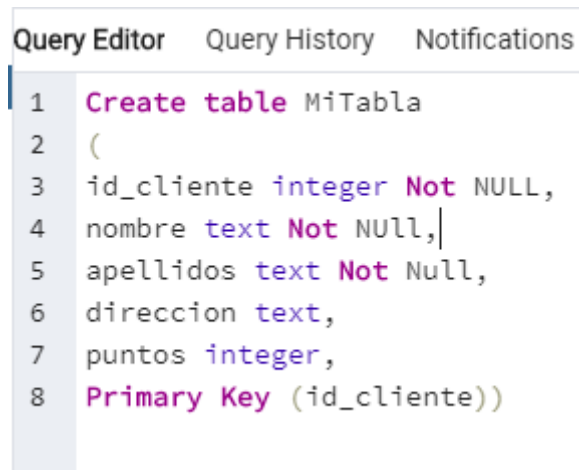
Los tamaños mostrados en la segunda columna estan representados en bytes. Como podemos observar **template1** y **templateo** tienen el mismo tamaño por lo explicado anteriormente. Sin embargo **postgres** y **mibasedatos** a pesar de ser tablas que no se han modificado, **postgres** ocupa mas espacio ya que contiene una extension adicional

que **mibasedatos**. Esta extension se llama **adminpack** la cual provee de funciones adicionales a la base de datos como control remoto de archivos del servidor. El tamaño total del directorio sería 32.123.620 bytes que equivale aproximadamente a 32 MB.

Si entramos en el subdirectorio de **mibasedatos** observamos que ya hay muchos archivos creados. Estos provienen de una plantilla donde se recrean todos los objetos necesarios y elementos para definir una nueva base de datos.

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, otra dirección de tipo text y otros puntos que sea de tipo integer. ¿Qué ficheros se han creado en esta operación? ¿Qué guarda cada uno de ellos? ¿Cuánto ocupan? ¿Por qué?

En primer lugar, creamos la tabla **MiTabla** con el siguiente código SQL:



```
Query Editor  Query History  Notifications
1  Create table MiTabla
2  (
3  id_cliente integer Not NULL,
4  nombre text Not NULL,
5  apellidos text Not Null,
6  direccion text,
7  puntos integer,
8  Primary Key (id_cliente))
```

Una vez creada la tabla nos dirigimos a la dirección **/data/base/16393** , es decir , el directorio de nuestra base de datos **mibasedatos**. Dentro del directorio podemos observar que se han creado 4 ficheros:

Nombre	Fecha de modifica...	Tipo	Tamaño
1259	14/02/2020 11:24	Archivo	104 KB
1259_vm	14/02/2020 11:24	Archivo	8 KB
2579	14/02/2020 11:24	Archivo	16 KB
2606	14/02/2020 11:24	Archivo	8 KB
2606_vm	14/02/2020 11:24	Archivo	8 KB
1247	14/02/2020 11:24	Archivo	80 KB
16394	14/02/2020 11:22	Archivo	0 KB
16397	14/02/2020 11:22	Archivo	0 KB
16399	14/02/2020 11:22	Archivo	8 KB
16400	14/02/2020 11:22	Archivo	8 KB
pg_internal.init	13/02/2020 14:41	Archivo INIT	145 KB
112	04/02/2020 16:41	Archivo	8 KB
113	04/02/2020 16:41	Archivo	8 KB
174	04/02/2020 16:41	Archivo	8 KB
175	04/02/2020 16:41	Archivo	8 KB

Los tamaños de los archivos **16394** y **16397** son de 0 bytes por lo que estos archivos están vacíos y los archivos **16399** y **16400** son de 8 KB. Para obtener más información sobre estos archivos realizamos la siguiente consulta:

Query Editor			Query History	Notifications
1	Select oid ,relname			
2	From pg_class			
Data Output			Explain	Messages
	oid	relname		
	oid	name		
1	16397	pg_toast_16...		
2	16399	pg_toast_16...		
3	16394	mitabla		
4	16400	mitabla_pkey		
5	2619	pg_statistic		
6	1247	pg_type		

El archivo **16394** se encarga de almacenar los datos de la tabla, actualmente al ser una tabla vacía tiene un tamaño de 0 como vimos arriba. El archivo **16400** contiene los

valores de las PK de la tabla para asegurarse que no se repitan el cual tiene un tamaño de 8KB ya que funciona como un índice que ocupa inicialmente 8 KB que es el tamaño por defecto de un bloque en postgre. El archivo **16397** contiene una estructura **TOAST**. Una estructura **TOAST** sirve para almacenar datos de gran tamaño, se crean debido a que existen atributos de longitud variable, en nuestro caso el tipo de dato **text** en nombre, apellidos y direccion. Este archivo actualmente se encuentra vacío debido a que no se ha introducido ningún dato en estos campos. Finalmente el archivo **16399** se corresponde con un índice de la estructura **TOAST** y por tanto al igual que el otro índice ocupa 8 KB.















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

Para insertar una tupla realizamos la siguiente operación:

```
Query Editor  Query History  Notifications

1  INSERT INTO public.mitabla(
2      id_cliente, nombre, apellidos, direccion, puntos)
3      VALUES (1, 'Aguiles', 'García Castro', '5A/5/calle de la plata/ Madrid', 70);
4
5
6
```

Una vez realizada la inserción de la tupla en la tabla nos vamos a la dirección **/data/base/16393** donde se encuentra nuestra base de datos:

	13178_fsm	04/02/2020 16:41	Archivo	24 KB
	13178_vm	04/02/2020 16:41	Archivo	8 KB
	13180	04/02/2020 16:41	Archivo	0 KB
	13182	04/02/2020 16:41	Archivo	8 KB
	13183	04/02/2020 16:41	Archivo	0 KB
	13185	04/02/2020 16:41	Archivo	0 KB
	13187	04/02/2020 16:41	Archivo	8 KB
	16394	14/02/2020 12:19	Archivo	8 KB
	16397	14/02/2020 11:22	Archivo	0 KB
	16399	14/02/2020 11:22	Archivo	8 KB
	16400	14/02/2020 12:19	Archivo	16 KB
	pg_filenode.map	04/02/2020 16:41	Archivo MAP	1 KB
	pg_internal.init	13/02/2020 14:41	Archivo INIT	145 KB
	PG_VERSION	04/02/2020 16:41	Archivo	1 KB

Los archivos correspondientes a los TOAST, **16397** y **16399**, no se han modificado debido a que los campos tipo **text** introducidos no eran lo suficientemente grandes para requerir de un archivo **TOAST**. El archivo **16400** ha aumentado su tamaño en un bloque (8 KB) debido a que se ha introducido una nueva **PK**. Por último, el archivo **16394** correspondiente a los datos de **MiTabla** han aumentado en un bloque (8 KB) debido a la inserción de nuevos 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?

Para poder aplicar el módulo **pg_buffercache** es necesario añadir la extensión del **pg_buffercache** previamente a la lista de extensiones:

```
Query Editor  Query History  Notifications
1 Create Extension pg_buffercache;
```

El módulo `pg_buffercache` nos permite saber lo que sucede en tiempo real en la cache del buffer. Si queremos que solo aparezca la información relativa a nuestra base de datos deberemos especificar el atributo **reldatabase** con el **oid** de nuestra base de datos. Para ello realizamos la siguiente consulta cogiendo solo los archivos referentes a nuestra tabla:

Query Editor
Query History
Notifications

```

1 Select *
2 From pg_buffercache
3 Where reldatabase = 16393 and
4 relfilenode = 16400 or relfilenode = 16399 or relfilenode = 16397 or relfilenode = 16394
5

```

Data Output
Explain
Messages

	bufferid integer	relfilenode oid	reltablespace oid	reldatabase oid	relforknumber smallint	relblocknumber bigint	isdirty boolean	usagecount smallint	pinning_backends integer
1	944	16394	1663	16393	0	0	false	1	0
2	945	16400	1663	16393	0	0	false	2	0
3	946	16400	1663	16393	0	1	false	1	0

Observando la imagen podemos ver que los archivos **TOAST** no aparecen en la caché y esto se debe a que ambos archivos no presentan almacenados datos referentes a la base de datos en cambio los archivos **16394** (datos de **MiTabla**) y **16400** (índice de las **PK**) si que estan cargados en la caché. El archivo **16400** tiene dos buffers ya que ocupa dos bloques como se muestra en la columna **relblocknumber** donde vemos que este archivo posee dos bloques diferentes (0 y 1) a diferencia del archivo **16394** el cual solo tiene un bloque (0).

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?

Una vez borrada la tabla **MiTabla**, la volvemos a crear tal y como dice el enunciado. El archivo .txt de datos de la tabla ocupa 921093 KB y los insertaremos en **MiTabla**. Ahora veremos el tamaño que ocupa nuestra tabla.

13180	04/02/2020 16:41	Archivo	0 KB
13182	04/02/2020 16:41	Archivo	8 KB
13183	04/02/2020 16:41	Archivo	0 KB
13185	04/02/2020 16:41	Archivo	0 KB
13187	04/02/2020 16:41	Archivo	8 KB
16411	21/02/2020 11:33	Archivo	1.048.576 KB
16411.1	21/02/2020 11:37	Archivo 1	230.832 KB
16411_fsm	21/02/2020 11:37	Archivo	336 KB
16414	21/02/2020 10:26	Archivo	0 KB
16416	21/02/2020 10:26	Archivo	8 KB
16417	21/02/2020 11:39	Archivo	436.920 KB
pg_filenode.map	04/02/2020 16:41	Archivo MAP	1 KB
pg_internal.init	13/02/2020 14:41	Archivo INIT	145 KB
PG_VERSION	04/02/2020 16:41	Archivo	1 KB

1.

El archivo correspondiente de almacenar los datos de **MiTabla** es el archivo con el oid **16411** el cual ocupa 1048576 KB. Como el archivo ha superado 1 GB que es el tamaño máximo que postgres permite almacenar en un archivo y por tanto crea un nuevo archivo llamado **16411.1** el cual ocupa 230832 KB. Por otro lado, tenemos el archivo con extensión _fsm con el oid correspondiente al archivo de almacenamiento de **MiTabla**, **16411_fsm** que ocupa 336 KB. Este archivo se encarga de llevar a cabo un seguimiento del espacio disponible en la relación de la tabla. Los archivos **16414** y **16416** son archivos **TOAST** y no se han modificado con la inserción de los datos por lo que no hay ningún dato que supere el tamaño requerido. El archivo **16417** es el archivo que almacena los índices de la tabla y ocupa 436920 KB.

El archivo original ocupaba 921093 KB mientras que el archivo de PostgreSQL donde se almacenan estos datos ocupa 1048576 KB + 230832 KB = 1.279.408 KB por lo que la diferencia entre el original y el de PostgreSQL es de 1.279.408 KB – 921093 KB = 358315 KB tiene de más el archivo PostgreSQL. Esto es debido en primer lugar, a que los registros no se parten en diferentes bloques (ya que aumentaría considerablemente el tiempo de búsqueda) cada registro va en un solo bloque, por lo tanto, los bloques no se aprovechan al 100%. En segundo lugar, la diferente variabilidad de la longitud de las tuplas, así como las cabeceras y espacios reservados de cada bloque, hacen que todo el espacio no sea destinado a almacenar los registros.

A continuación, vamos a realizar el cálculo teórico de los bloques de la relación **MiTabla**.

- **Tamaño del bloque (B):** 8KB (tamaño por defecto de PostgreSQL).

- **Número de registros (n_r):** $15 \cdot 10^6$.
- **Longitud de registro (L_r):** Como los campos de los registros varían su tamaño realizaremos una consulta a cada campo para hallar el valor medio de cada uno.

```
Select avg(pg_column_size(nombre))
From mitabla
```

	avg numeric
1	14.259259333333333

Realizamos esta consulta con cada campo y obtenemos que la longitud de cada registro es: $4 + 14 + 17 + 17 + 4 = 56$ Bytes

- **Factor de bloque (F_R):** $F_R = B/L_R = 8192 \text{ Bytes} / 56 \text{ Bytes} = 146$ registros por bloque.
- **Número de bloques del archivo (b_R):** $b_R = n_R/F_R = 15 \cdot 10^6 / 146 = 102740$ bloques.
- **Conversion a bytes:** $102740 \text{ bloques} \cdot 8 \text{ KBytes/bloque} = 821920 \text{ KBytes}$.

El tamaño teorico dista bastante del tamaño real en postgresQL debido a que no se tiene en cuenta todas las reglas y variabilidades de PostgreSQL como por ejemplo las cabeceras de las tuplas.

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?

Volvemos a realizar la consulta de la cuestión 4:

```
Select *
from pg_buffercache
Where reldatabase = 16393
```

Tras realizar la consulta escogemos los **oid** de los archivos que contienen los índices y los datos de tabla ya que los archivos TOAST al igual que en la cuestión 4 no contienen datos referentes a esta base de datos.

Data Output Explain Messages

	bufferid integer	reelfilenode oid	relextendspace oid	reldatabase oid	reelforknumber smallint	relextendspace bigint	isdirty boolean	usagecount smallint	pinning_backends integer
1	1	16417	1663	16393	0	15285	false	1	0
2	2	16417	1663	16393	0	44467	false	0	0
3	3	16417	1663	16393	0	20596	false	1	0
4	4	16417	1663	16393	0	37456	false	1	0
5	5	16417	1663	16393	0	8521	false	1	0
6	6	16417	1663	16393	0	21792	false	0	0
7	7	16417	1663	16393	0	2167	false	1	0
8	8	16417	1663	16393	0	8137	false	1	0
9	9	16417	1663	16393	0	1801	false	0	0
10	10	16417	1663	16393	0	29932	false	1	0

Imagen 1. Muestra los 10 primeros buffers correspondientes al archivo 16417 de MiTabla

Data Output Explain Messages

	bufferid integer	reelfilenode oid	relextendspace oid	reldatabase oid	reelforknumber smallint	relextendspace bigint	isdirty boolean	usagecount smallint	pinning_backends integer
1	90	16411	1663	16393	0	159894	false	1	0
2	171	16411	1663	16393	0	159816	false	1	0
3	282	16411	1663	16393	0	159808	false	1	0
4	336	16411	1663	16393	0	159856	false	1	0
5	622	16411	1663	16393	0	159786	false	1	0
6	907	16411	1663	16393	0	159919	false	1	0
7	927	16411	1663	16393	0	159866	false	1	0
8	930	16411	1663	16393	0	159868	false	1	0
9	941	16411	1663	16393	0	159872	false	1	0
10	960	16411	1663	16393	0	159794	false	1	0

Imagen 2. Muestra los 10 primeros buffers correspondientes al archivo 16411 de MiTabla.

Al realizar estas consultas observamos a primera vista que el número de buffers ha aumentado en gran medida. Esto se debe a la reciente inserción de 15 millones de registros en nuestra tabla lo que implica una carga de datos en la memoria que a su vez se cargará en la caché. Sin embargo, no todos los datos se cargan en la caché porque solo permanecen aquellos datos que han sido utilizados recientemente. Por lo que, si un archivo ya ha sido cargado con éxito y se requiere de espacio libre de buffers, este será reemplazado.

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?

Primero creamos la extensión pgstattuple :

```
Create Extension pgstattuple;
```

Este modulo nos permite obtener estadísticas de nuestros registros. A continuación la aplicaremos a **MiTabla**:

```
Select * From pgstattuple('mitabla');
```

Data Output Explain Messages

	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

- **table_len:** muestra el tamaño total de nuestra tabla en bytes.

- **tuple_count:** muestra el número de tuplas vivas (tuplas que no se han eliminado) que tiene la tabla.
- **tuple_len:** muestra la suma de la longitud de las tuplas vivas en bytes.
- **tuple_percent:** muestra el grado de ocupación de los bloques de la tabla.
- **dead_tuple_count:** muestra el número de tuplas muertas (tuplas que se han eliminado) de la tabla.
- **dead_tuple_len:** muestra la suma de la longitud de las tuplas muertas en bytes.
- **dead_tuple_percent:** muestra el porcentaje de tuplas muertas que hay en la tabla.
- **free_space:** muestra el espacio libre de la tabla en bytes.
- **free_percent:** muestra el espacio libre de la tabla en porcentaje.

El grado de ocupación de los bloques es de 93.09% y la cantidad de espacio libre total es de 5761792 bytes porque los registros no se fragmentan en bloques (para reducir el tiempo de respuesta) pero provocan que los bloques no se aprovechen totalmente.

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 realizamos la siguiente consulta:

Query Editor

Notifications

Query History

1

Select (8192/ (tuple_len / tuple_count)) as Fr

2

From pgstattuple('"mitabla"');

Data Output

Explain

Messages

	fr	
	bigint	
1	101	

Después de realizar la consulta obtenemos el factor de bloque medio real, 101 registros por bloque. El factor de bloque teórico lo calculamos en la cuestión 5 y nos da como resultado 146 registros por bloque. Esta diferencia se debe a que en el factor de bloque teórico no se tiene en cuenta ninguna información de control o cabeceras, además de la escasa precisión en los cálculos dando lugar a que se puedan guardar más registros por bloque.

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?








Primero creamos la extensión:

```
Create Extension pageinspect;
```

Este módulo sirve para inspeccionar el contenido de las páginas de la base de datos a bajo nivel, esto es útil para los procesos de *debugging*. A continuación, vamos a analizar la cabecera con la función **page_header** más la función `get_raw_page` para poder leer un bloque x:

Primera página del primer bloque:

```
Select * From page_header(get_raw_page('mitabla', 0));
```

Data Output		Explain	Messages							
	lsn pg_lsn	 checksum smallint	 flags smallint	 lower smallint	 upper smallint	 special smallint	 pagesize smallint	 version smallint	 prune_xid xid	
1	0/16D0170	0	0	400	448	8192	8192	4		0










La página del bloque situado a la mitad:

```
Select * From page_header(get_raw_page('mitabla', 74258));
```

Data Output		Explain	Messages						
	lsn pg_lsn	checksum smallint	flags smallint	lower smallint	upper smallint	special smallint	pagesize smallint	version smallint	prune_xid xid
1	0/427A4180	0	0	400	400	8192	8192	4	0

La página del último bloque:

```
Select * From page_header(get_raw_page('mitabla', 148515));
```

Data Output		Explain	Messages							
	lsn pg_lsn	 checksum smallint	 flags smallint	 lower smallint	 upper smallint	 special smallint	 pagesize smallint	 version smallint	 prune_xid xid	
1	0/9834A050		0	0	400	440	8192	8192	4	0

- **Isn:** localiza la entrada WAL (consiste en escribir en un registro los cambios que se van a producir, antes de que se apliquen) más reciente relacionada con este bloque. Cada uno tiene un **Isn** diferente porque está relacionada con cada archivo específico.
- **checksum:** se encarga de comprobar si los datos del bloque están disponibles. Todos los bloques están en 0 lo que significa que sus datos están disponibles.
- **flags:** Los flags sirven para ajustar los parámetros y opciones de PostgreSQL. Es el mismo valor para los tres.
- **lower y upper:** indican donde empieza el espacio libre y donde acaba. Viendo el **lower** y **upper** de estas tres páginas observamos que la del primero y el último bloque son similares, de unos 40 bytes. En cambio, la del

bloque intermedio es o el espacio libre debido a que se ha completado su espacio libre.

- **special:** muestra donde empieza un espacio especial para el bloque. En este caso es el mismo valor para los tres, 8KB que coincide con el final del bloque.
- **pagesize:** Muestra el tamaño del bloque en bytes. En este caso es el mismo para todos ya que todos tienen el mismo tamaño de bloque, 8KB por defecto en PostgreSQL.
- **versión:** Indica la versión de **pgAdmin** donde todas coinciden.

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 el índice tipo árbol realizamos la siguiente operación:

```
Create Index indiceArbol On "mitabla" Using btree (puntos)
```

Una vez realizado el índice comprobamos en que archivo se ha guardado y cuantos bloques ocupa con la siguiente consulta:

```
Select oid, relname, relpages From pg_class
```

	oid oid	relname name	relpages integer
1	16457	indicearbol	41188
2	2619	pg_statistic	19
3	1247	pg_type	10
4	16404	pg_buffercache	0

Podemos observar que se ha creado el archivo 16457 el cual contiene el índice árbol y ocupa 41188 bloques y como podemos observar en la siguiente imagen el tamaño en bytes es de 329504 Bytes.

PG_VERSION	04/02/2020 16:41	Archivo	1 KB
pg_internal.init	13/02/2020 14:41	Archivo INIT	145 KB
pg_filenode.map	04/02/2020 16:41	Archivo MAP	1 KB
16457	26/02/2020 16:23	Archivo	329.504 KB
16417	21/02/2020 11:39	Archivo	436.920 KB
16416	21/02/2020 10:26	Archivo	8 KB

Para obtener la información de los niveles del árbol emplearemos el módulo **pgstatindex**.

```
Select * From pgstatindex ('indicearbol')
```

Data Output		Explain	Messages							
	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	

La columna **tree_level** muestra el nivel en el que se encuentra la raíz del índice árbol por lo tanto el índice tiene los niveles 0, 1 y 2 por tanto es un árbol de 3 niveles.

En el nodo intermedio tenemos 202 bloques ya que la columna **internal_pages** nos muestra el número de bloques que hay por encima del nivel hoja, en este caso el nodo intermedio y el nodo raíz. El nodo raíz ocupa un bloque por lo tanto el resto de bloques deben ser del nodo intermedio. En la columna **leaf_pages** podemos ver los bloques que ocupan los nodos hojas, en este caso, 40984 bloques.

Por último, hayaremos las tuplas que tiene cada bloque. Para ello empezamos en nuestro bloque raíz el cual nos lo muestra la columna **root_block_no**: 209. Para ver el número de tuplas de cada nivel realizamos la siguiente consulta, en este caso con el nodo raíz:

```
Select count(*) From bt_page_items('indicearbol', 209)
```

	Data Output	Explain	Messages
	count bigint		
1	202		

Como podemos ver el bloque raíz tiene 202 tuplas. Ahora veremos el número de bloques de un bloque del nivel intermedio. Para ello realizamos la siguiente consulta:

```
Select * From bt_page_items('indicearbol', 209)
```

Data Output	Explain	Messages	
<div><div></div><div>itemoffset</div><div>smallint</div></div>	<div><div></div><div>ctid</div><div>tid</div></div>	<div><div></div><div>itemlen</div><div>smallint</div></div> <div><div></div><div>nulls</div><div>boolean</div></div> <div><div></div><div>vars</div><div>boolean</div></div> <div><div></div><div>data</div><div>text</div></div>	
1	1 (3,0)	8 false false	
2	2 (208,4097)	24 false false	03 00 00 00 00 00 00 00 01 00 27 2...
3	3 (413,4097)	24 false false	06 00 00 00 00 00 00 00 02 00 07 4...
4	4 (617,4097)	24 false false	0a 00 00 00 00 00 00 00 00 00 00 3a fe...
5	5 (821,4097)	24 false false	0d 00 00 00 00 00 00 00 02 00 68 2...
6	6 (1025,4097)	24 false false	11 00 00 00 00 00 00 00 00 00 00 f0 cd...
7	7 (1229,4097)	24 false false	14 00 00 00 00 00 00 00 01 00 26 fb...
8	8 (1433,4097)	24 false false	18 00 00 00 00 00 00 00 00 00 00 df a6...
9	9 (1637,4097)	24 false false	1b 00 00 00 00 00 00 00 01 00 cb cc...
10	10 (1841,4097)	24 false false	1f 00 00 00 00 00 00 00 00 00 00 79 81...
11	11 (2045,4097)	24 false false	22 00 00 00 00 00 00 00 01 00 47 a...
12	12 (2249,4097)	24 false false	26 00 00 00 00 00 00 00 00 00 00 f2 58...
13	13 (2453,4097)	24 false false	2a 00 00 00 00 00 00 00 01 00 52 7...

Al realizar esta consulta, podemos saber que bloques pertenecen al nivel intermedio. Nosotros escogeremos el bloque número 208 y realizaremos otra vez la consulta anteriormente usada para comprobar el número de tuplas.

```
Select count(*) From bt_page_items('indicearbol', 208)
```

Data Output	Explain	Messages
	count bigint	
1	204	

Podemos observar que en el nivel intermedio los bloques contienen 204 tuplas cada uno. A continuación, realizamos esta consulta para escoger un bloque perteneciente al nivel hoja:

```
Select * From bt_page_items('indicearbol', 208)
```

Data Output		Explain	Messages			
	itemoffset smallint	ctid tid	itemlen smallint	nulls boolean	vars boolean	data text
1		1 (410,4097)		24 false	false	06 00 00 00 00 00 00 00 02 00 07 47 4...
2		2 (205,0)		8 false	false	
3		3 (206,4097)		24 false	false	03 00 00 00 00 00 00 00 00 01 00 3b 33 2...
4		4 (207,4097)		24 false	false	03 00 00 00 00 00 00 00 00 01 00 bd 3d 1...
5		5 (210,4097)		24 false	false	03 00 00 00 00 00 00 00 00 01 00 78 48 2...
6		6 (211,4097)		24 false	false	03 00 00 00 00 00 00 00 00 01 00 8b 52 1...
7		7 (212,4097)		24 false	false	03 00 00 00 00 00 00 00 00 01 00 2f 5d 5...
8		8 (213,4097)		24 false	false	03 00 00 00 00 00 00 00 00 01 00 08 67 3...
9		9 (214,4097)		24 false	false	03 00 00 00 00 00 00 00 00 01 00 fe 71 5...
10		10 (215,4097)		24 false	false	03 00 00 00 00 00 00 00 00 01 00 a9 7c 0...
11		11 (216,4097)		24 false	false	03 00 00 00 00 00 00 00 00 01 00 0d 87 5...
12		12 (217,4097)		24 false	false	03 00 00 00 00 00 00 00 00 01 00 2d 91 2...

Escogemos el bloque número 210 y volvemos a realizar la consulta para ver el número de tuplas que tienen los bloques en este nivel.

```
Select count(*) From bt_page_items('indicearbol', 210)
```

Data Output	Explain	Messages									
<table> <tr> <th></th><th>count</th><th></th></tr> <tr> <td></td><td>bigint</td><td></td></tr> <tr> <td>1</td><td>367</td><td></td></tr> </table>		count			bigint		1	367			
	count										
	bigint										
1	367										

Como podemos ver cada bloque de este nivel contiene 367 tuplas.

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.

En primer lugar, hallamos la longitud del campo empleado para el índice. En este caso es puntos que es un integer que en PostgreSQL tiene un tamaño de 4 bytes. Por tanto, **$L_k = 4$ Bytes**. El tamaño del bloque en PostgreSQL es de 8192 Bytes (**$B = 8192$ Bytes**). Para hallar la longitud del puntero a registro y puntero a bloque miramos la columna **ítemlen** de las tuplas de nodo intermedio y nodo hoja respectivamente. En ambos casos lo mismo y por tanto valdrán lo mismo, 24. A continuación, hallamos la longitud restando al valor anterior la longitud del campo. Por tanto, **$L_{Pb} = L_{Pr} = 20$ bytes**.

Ahora vamos a calcular el número de los nodos hoja:

$$n_1 * (L_{Pr} + L_k) + L_{Pb} \leq B \rightarrow n * (20 + 4) + 20 \leq 8192 \rightarrow n_h = 340.$$

Ahora calculamos los nodos intermedios:

$$n_2 * L_{Pb} + (n-1) * L_k \leq B \rightarrow n * 20 + (n - 1) * 4 \leq 8192 \rightarrow n = 341$$

Por último, calculamos el número de bloques por nivel:

- **Nodo hoja:**

$$b_{hoja} = \lceil n_r / n_h \rceil \rightarrow b_{hoja} = \lceil 15.000.000 / 340 \rceil = 44.118 \text{ bloques.}$$

- **Nodo intermedio:**

$$b = \lceil b_{hoja} / n \rceil \rightarrow b = \lceil 44.118 / 341 \rceil = 130 \text{ bloques.}$$

- **Nodo raíz:**

$$b_{raiz} = \lceil b / n \rceil \rightarrow b_{raiz} = \lceil 130 / 341 \rceil = 1 \text{ bloque.}$$

El número de niveles sí que coinciden con el número de niveles real. Pero los bloques calculados teóricamente no coinciden con los de PostgreSQL. Esto se debe a que en la teoría no se tiene en cuenta la información de control. Los registros no solo contienen información del puntero y la clave, sino que también contiene información del registro como su posición dentro del bloque, posición del registro en el bloque que está almacenado, etc.

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

Para crear los índices hash realizamos las siguientes operaciones:

```
Create index indice_hash_id_cliente on "mitabla" using hash (id_cliente)
```

```
Create index indice_hash_puntos on "mitabla" using hash (puntos)
```

Para ver en que archivos se han almacenado realizamos la siguiente consulta:

```
Select oid, relname From pg_class
```









Data Output			Explain	Messages
	oid oid	relname name		
1	16457	indicearbol		
2	2619	pg_statistic		
3	1247	pg_type		
4	16404	pg_buffercache		
5	16458	indice_hash_id_cliente		
6	16459	indice_hash_puntos		
7	4159	pg_toast_2600		
8	4160	pg_toast_2600_index		
9	2830	pg_toast_2604		
10	2831	pg_toast_2604_index		
11	4161	pg_toast_3456		
12	4162	pg_toast_3456_index		
13	2832	pg_toast_2606		

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?

Primero aplicamos el módulo **pgstattuple**:


- **Id_cliente:**

```
Select * From pgstattuple('indice_hash_id_cliente')
```

Data Output		Explain	Messages															
	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	535683072		15000000		240000000		44.8		0		0		0		232789572		43.46	

- **puntos:**

```
Select * From pgstattuple('indice_hash_puntos')
```

Data Output		Explain	Messages															
	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	701440000		15000000		240000000		34.22		0		0		0		397648056		56.69	

En la columna **table_len** se almacena el tamaño del índice en bytes. Observamos que es diferente en ambos, es mayor en el campo puntos. Esto se debe a que el campo `id_cliente` es clave primaria y por tanto todos los valores son diferentes y el índice debe apuntar a todos los registros. Sin embargo, el campo puntos es un índice secundario y por tanto no debe apuntar a todos los registros. Pero el campo puntos ocupa más espacio, esto probablemente se deba a que el archivo inicial no esté ordenado según el campo puntos, creándose un índice secundario y no clave que necesitaría de cajones de punteros y cajones de desbordamiento ya que podría tener un valor repetido muchas veces, lo que aumentaría considerablemente su tamaño. La cantidad de espacio libre como podemos ver en la columna **free_space** es superior también en el campo puntos que en el `id_cliente` debido a que el índice de puntos presenta un mayor número de estructuras (cajones de punteros y cajones de desbordamiento) y el número de bloques, por lo que la suma de los espacios libres de los bloques será mayor. Los porcentajes vistos en la columna **free_percent** también varían debido a lo nombrado anteriormente.

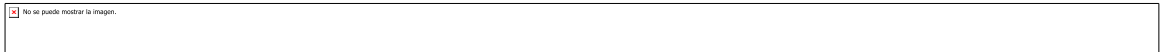
Ahora aplicaremos el módulo **pageinspect**. Este módulo presenta diferentes opciones para las funciones hash. Utilizaremos la función **hash_page_stats** para ver la información sobre una página del índice hash y la función general **get_raw_page()** para mostrar la primera línea :

- **Id_cliente:**

```
Select * From hash_page_stats(get_raw_page('indice_hash_id_cliente', 1))
```

	Data Output	Explain	Messages								
	live_items integer	dead_items integer	page_size integer	free_size integer	hasho_prevbkno bigint	hasho_nextbkno bigint	hasho_bucket bigint	hasho_flag integer	hasho_page_id integer		
1	245	0	8192	3248	49151	4294967295	0	2	65408		

- puntos:



Al igual que el modulo pgstattuple vemos una diferencia en el espacio libre que es debido principalmente a las causas comentadas anteriormente.

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.

Antes de realizar las operaciones mencionas en la cuestión vamos a realizar las operaciones con los módulos que vamos a utilizar para comparar posteriormente los resultados:

En primer lugar, vamos a realizar la inserción de la siguiente tupla:

Una vez insertada vamos a ejecutar el modulo **pgstattuple** y **pg_class** para comprobar las modificaciones que ha podido crear en los archivos:

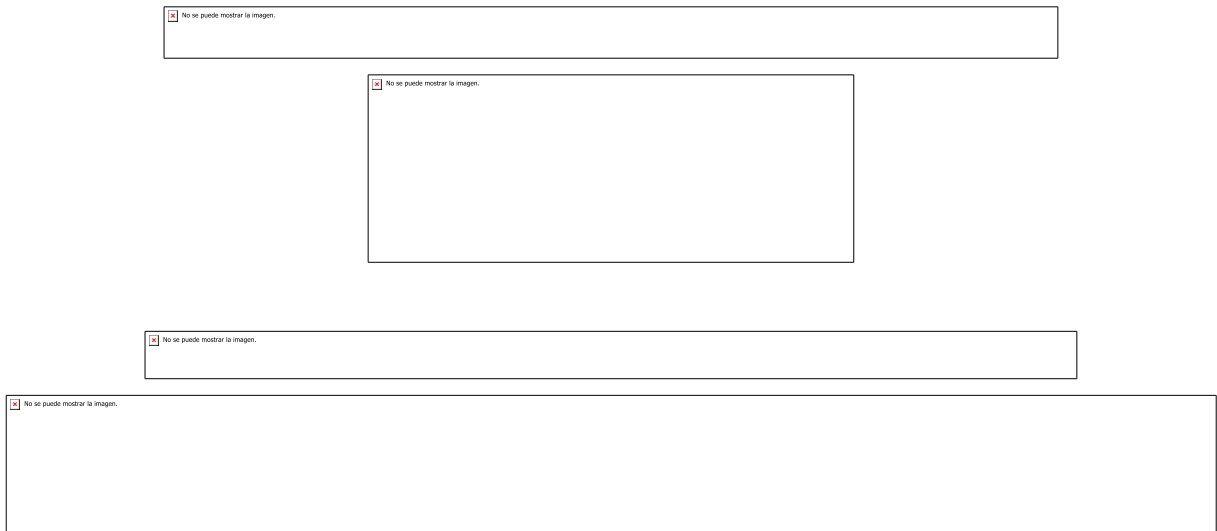
Podemos observar en los resultados del módulo **pgstattuple** el número de tuplas ha aumentado en uno debido a la inserción de la nueva tupla, como cabría esperar. Sin embargo, si nos fijamos en el módulo **pg_class**, no se ha modificado el tamaño de **mitabla** debido a que la tupla se ha guardado en un bloque previamente creado ya que no ocupa lo suficiente para que la base de datos necesite crear otro bloque para almacenarla.

Ahora vamos a modificar la tupla insertada. Para ello localizaremos la tupla con la siguiente operación y luego la modificaremos esta tupla para ver como varían los archivos:

2.

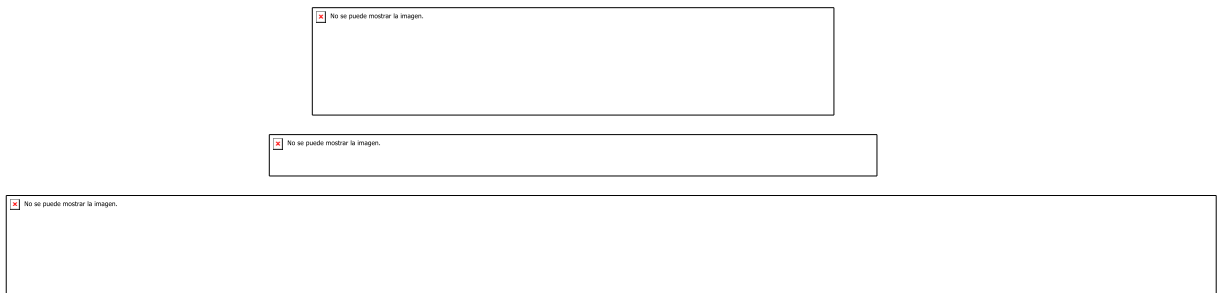
3.

Una vez localizada nuestra tupla, vamos a realizar la modificación de la misma y realizar las operaciones anteriores:



Podemos observar que la tupla ha cambiado de localización esto se debe a que PostgreSQL cuando actualiza una tupla la marca como tupla vacía e inserta la nueva tupla como podemos ver con el módulo **pg_header** donde cambia su **isn** y mirando la consulta realizada, su localización.

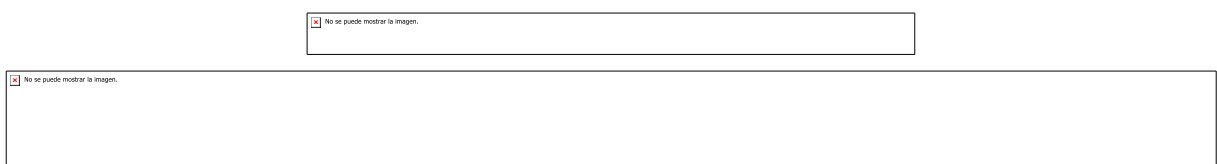
Por último realizamos la operación de borrado:

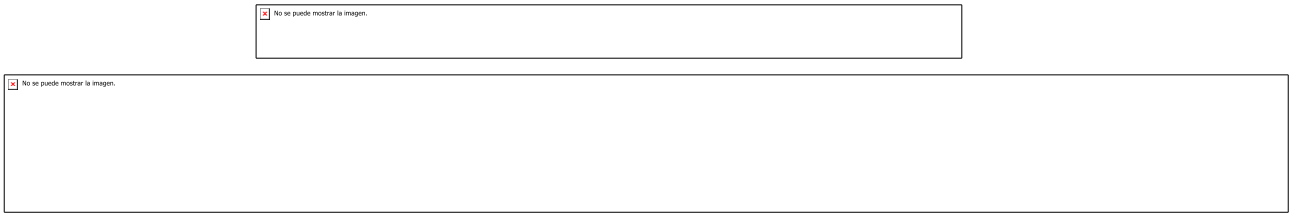


Como cabría esperar se reduce en uno el total de tuplas de la tabla y aumenta en uno la columna **dead_tuple_count** lo cual muestra que PostgreSQL lo que realiza es marcar la tupla como muerta pero no la borra completamente de la tabla, sigue ocupando espacio. Para eliminarla completamente se necesitaría aplicar un **vacuum**.

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

Antes de realizar el borrado realizamos la siguiente operación para poder comparar resultados:





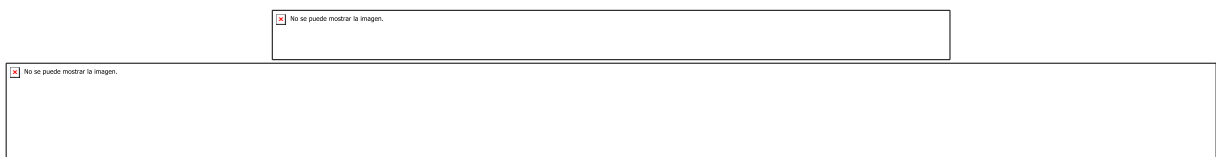
Comenzamos realizando la función de borrado aleatorio de las 2.000.000 de tuplas:



Una vez eliminadas las 2000000 de tuplas volvemos a ejecutar el módulo **pgstattuple**:



Observando la columna **tuple_count** podemos ver que se han “borrado” correctamente las 2000000 de tuplas, ya que observamos que hay 13000000 de tuplas vivas. Pero si observamos la columna **table_len** vemos que no ha variado, esto se debe a que PostgreSQL no ha borrado las tuplas, sino que las ha marcado para un borrado posterior representado en la columna **dead_tuple_len** y por lo tanto siguen ocupando espacio. Ahora veremos cómo se ha modificado el índice árbol:



Como podemos ver el árbol no se ha modificado porque las tuplas borradas siguen estando en los bloques del árbol ya que solo han sido marcadas.

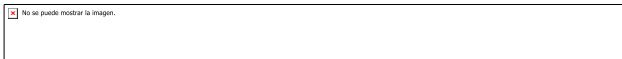
Cuestión 16. En la situación anterior, ¿Qué operaciones se puede aplicar a la base de datos **MiBaseDatos** para optimizar el rendimiento de esta? Aplicarla a la base de datos **MiBaseDatos** y comentar cuál es el resultado final y qué es lo que ocurre físicamente.

Una manera de optimizar su rendimiento es mediante la herramienta **Vacuum**. Esta herramienta sirve para recoger la basura, lo que significa que elimina las tuplas que

hayan sido marcadas para borrado, las tuplas muertas. Ya que, tras un **Delete**, por ejemplo, las tuplas no son eliminadas como vimos en la cuestión anterior. Además, con **VACUUM FULL** también se encarga de mover los datos a espacios que se hayan quedado libres, lo que permite aprovechar los espacios libres que se hayan quedado en los bloques. Vamos a aplicar el **Vacuum Full**:



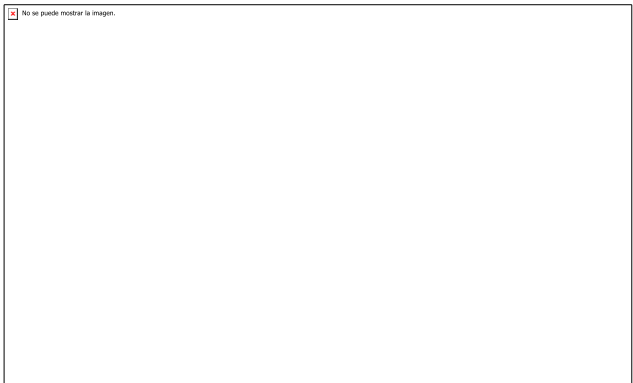
A continuación, utilizamos el módulo **pgstattuple** para ver lo cambios realizados:



Como podemos ver, han desaparecido las tuplas muertas en la columna de **dead_tuple_len** y el tamaño de la tabla se ha reducido como podemos ver en la columna **table_len**, debido a que las tuplas muertas han dejado de ocupar espacio y se ha reorganizado las tuplas para aprovechar los espacios libres de los bloques que han dejado las tuplas eliminadas y por tanto se ha reducido el número de bloques necesarios para almacenar las tuplas.

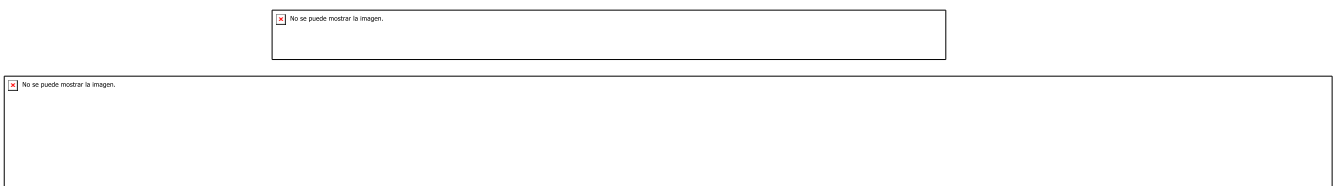
Cuestión 17. Crear una tabla denominada **MiTabla2** de tal manera que tenga un factor de llenado de tuplas que sea un 40% que el de la tabla **MiTabla** y cargar el archivo de datos anterior Explicar el proceso seguido y qué es lo que ocurre físicamente.

Para rellenar la tabla **MiTabla2** con un factor de bloque de 40% debemos modificar el valor del campo **fillfactor**. **Fillfactor** es un campo que indica el porcentaje de ocupación que presenta el bloque, que hace referencia al número de tuplas que caben en un bloque. Por defecto este campo se encuentra al 100% así que debemos modificarlo y ponerlo al 40%. Para ello realizamos la siguiente operación y cargamos los datos:





Como podemos comprobar en los archivos de **mibasedatos** se han creado 4 nuevos archivos, esto se debe a que se ha tenido que fragmentar debido a la falta de espacio en **16973,16973.1, 16973.2, 16973.3**, ya que cada archivo no puede superar 1GB de tamaño. Esto se ha producido ya que al tener un factor de bloque mucho más pequeño el número de registros que caben en cada bloque es mucho menor, un 60% menos. Requiriendo un mayor número de bloques para almacenar el mismo número de tuplas. Podemos observar el número de bloques con el módulo **pgstattuple**:



Como podemos ver si lo comparamos con la cuestión 7 casi ha triplicado el número de bloques.

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.

Tal y como realizamos en la cuestión 14, vamos a realizar las operaciones con los módulos para poder comparar posteriormente:

No se puede mostrar la imagen.

No se puede mostrar la imagen.

No se puede mostrar la imagen.

No se puede mostrar la imagen.

Primero, vamos a realizar la insercion de una tupla:

No se puede mostrar la imagen.

Una vez realizada la inserccion utilizamos los módulos **pgstattuple** y **pg_class** para comprobar las modificaciones que haya podido crear en los archivos:

No se puede mostrar la imagen.

No se puede mostrar la imagen.

No se puede mostrar la imagen.

No se puede mostrar la imagen.

Ahora vamos a modificar la tupla insertada. Para ello localizaremos la tupla con la siguiente operación y luego la modificaremos esta tupla para ver como varían los archivos:

No se puede mostrar la imagen.

No se puede mostrar la imagen.



Una vez localizada la tupla vamos a modificarla y comopararla con las operaciones anteriores:



Por último vamos a borrar la tupla:



En coclusion podemos observar que la unica diferencia que existe entre esta tabla y la anterior es la cantidad de registros que se pueden almacenar en cada bloque, que es muy inferior en este caso.

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?

La partición parte una tabla grande en varias más pequeñas. Otorga varios beneficios:

- mejora la eficacia a la hora de realizar una consulta de ciertas situaciones, sobre todo cuando se necesita acceder a pocas columnas de la tabla.
- Cuando las consultas o actualizaciones acceden a un largo porcentaje de una sola partición.
- Las cargas masivas y borrados masivos pueden ser realizados mediante la adición o borrado de una partición.
- Los datos pocos usados son enviados a almacenamiento más lento y barato.

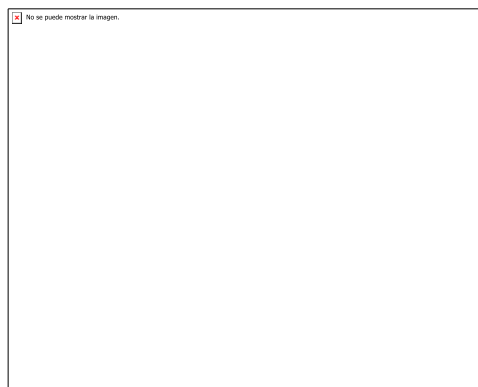
Los tipos de particionamiento que existe son:


- 1) **Range Partitioning**, la tabla se parte en rangos definidos por una columna clave o por un conjunto de columnas, sin overlap entre los rangos de valores asignados a diferentes particiones.
- 2) **List Partitioning**, la tabla se particiona listando explícitamente que valores clave se encuentran en cada partición.
- 3) **Hash Partitioning**, la tabla es particionando mediante la especificación de módulos y un recuerdo para cada partición. Cada partición va a almacenar las filas para los que el valor hash de la clave de partición dividida entre el módulo especificado dará como resto el especificado.

El particionamiento será útil cuando existan tablas muy grandes y a la hora de consultar o leer datos normalmente no se necesite acceder a todas las filas de datos.

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 partición hash realizamos la siguiente operación:



 No se puede mostrar la imagen.

Una vez creada la tabla y particionarla en 10 módulos, para ello coge el valor de puntos lo divide entre 10 y según el resto lo irá introduciendo en una partición de tabla u otra. Por ejemplo, si la tupla tiene en el campo **puntos** 72, al dividirlo entre 10 el resto es 2 y por lo tanto se almacenaría en la partición **MiTabla3_2**.

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.

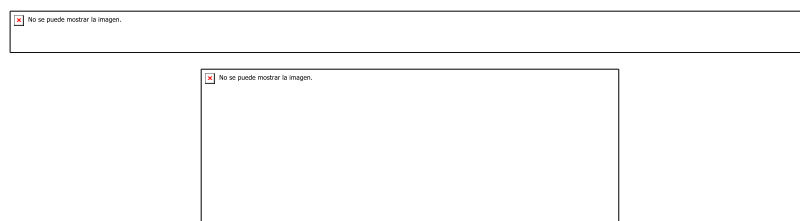
En primer lugar, hemos introducido los datos dados para esta práctica en la tabla **Mitabla3**. Una vez realizado esto miramos el número de bloque en cada partición de la tabla con el módulo **pg_class** y realizando la siguiente consulta:



Los bloques que ocupa cada partición se pueden ver en la columna **relpages**, (Mitabla3 no tiene bloques porque no es una partición y los datos solo se almacenan en las particiones de tabla). El número de bloques de cada partición es más o menos similar debido a que el valor de puntos es aleatorio y con grandes cifras de datos tiende a realizar una distribución uniforme más o menos.

Ahora realizaremos su cálculo teórico. En primer lugar, debido a que la probabilidad de un valor para el campo puntos es equiprobable respecto a cualquier otro valor de mismo campo, tomaremos el campo como si estuviese uniformemente distribuido.

- **Tamaño del bloque (B):** 8KB (tamaño por defecto de PostgreSQL).
- **Número de registros (n_r):** $15 \cdot 10^6$.
- **Longitud de registro (L_r):** Como los campos de los registros varían su tamaño realizaremos una consulta a cada campo para hallar el valor medio de cada uno.



Realizamos esta consulta con cada campo y obtenemos que la longitud de cada registro es: $4 + 4 + 17 + 17 + 4 = 46$ Bytes.

- **Factor de bloque (F_R):** $F_R = B/L_R = 8192 \text{ Bytes} / 46 \text{ Bytes} = 178$ registros por bloque.
- **Número de bloques del archivo (b_R):** $b_R = n_R/F_R = 15 \cdot 10^6 / 178 = 84270$ bloques.
- **Número de bloques por partición:** $b_R/n^o \text{ de particiones} = 84270/10 = 8427$ bloques por partición.

El tamaño teórico es prácticamente la mitad del tamaño real en PostgreSQL debido a que no se tiene en cuenta todas las reglas y variabilidades de PostgreSQL como por ejemplo las cabeceras de las tuplas.

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 se puede mostrar con esas herramientas? ¿Sobre qué tipo de estructuras se puede recopilar información de la actividad? Describirlo brevemente.

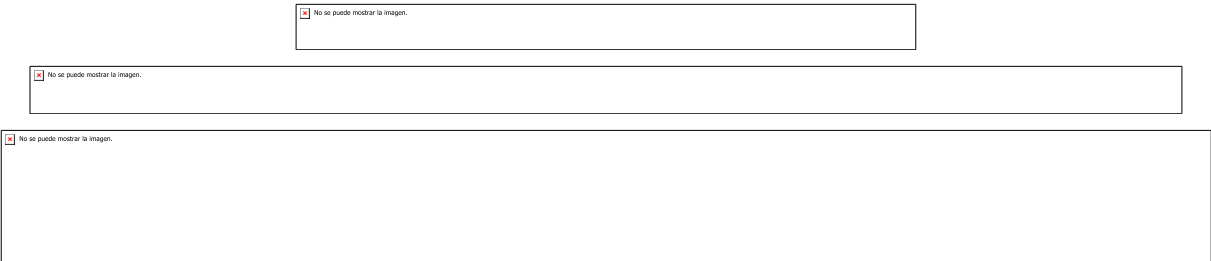
La principal herramienta que podemos emplear para monitorizar la actividad de la base de datos es **Statistics Collector**, que permite recolectar y reportar información sobre la actividad de la base de datos. También podemos emplear el comando **explain**, que nos muestra el plan de ejecución de una consulta y nos da información de lo que PostgreSQL realiza internamente, como puede ser el coste de una operación. Otra herramienta útil es el módulo **auto_explain** que es muy similar al comando **explain** pero que se va actualizando de manera automática.

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

En primer lugar, creamos el índice:

 No se puede mostrar la imagen.

El índice se ha especificado que debe ser primario, por lo tanto, ahora debemos ordenar secuencialmente el archivo respecto al campo puntos. Para realizar esto vamos a emplear la herramienta **Cluster** que nos realiza un reordenamiento físico de la tabla en base al campo sobre el que se crea el índice.



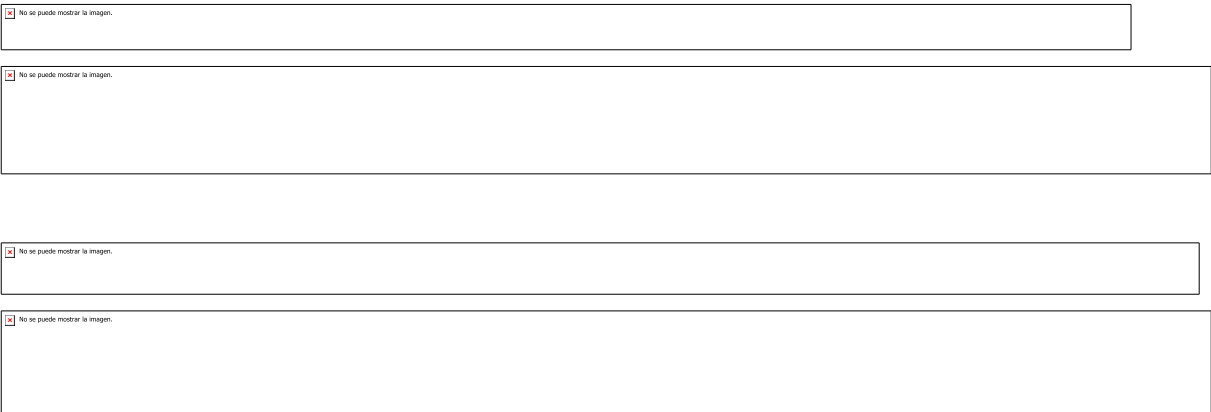
Es importante recordar que esta ordenación no se va actualizando y, por tanto, si en algún momento se borran, actualizan o insertan nuevos archivos el índice puede perder la condición de primaria.

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

Para realizar el índice hash realizamos las siguientes operaciones:



Para comprobar que se han creado correctamente realizamos las siguientes operaciones:



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

Para ver el tamaño de los índices realizamos las siguientes operaciones:



Existen diferencias entre ambos índices hash, debido a que uno es un índice secundario clave y el otro un primario+ no clave. Esto se deberá a las diferentes distribuciones de la función hash en cada campo y los cajones de desbordamiento del índice del no clave ya que se repetirán muchos valores y se necesitarán de cajones de desbordamiento. En el clave todos los valores son únicos y por tanto la probabilidad de que necesite cajones de desbordamiento es menor. A continuación, comparando los índices hash con los **btree** observamos que los **btree** ocupan menos. Esto se produce debido a la necesidad de bloques de desbordamiento en el caso de los has, que incrementará su tamaño. Y como el **btree** va construyendo el índice (no está predeterminado) ya indexa de manera directa el registro del dato correspondiente, en conclusión, haciendo que ocupe menos que el hash.

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 monitorización de los datos nos aporta información del índice o los índices que decide emplear PostgreSQL en cada consulta (el más eficiente) y también nos aporta más información del número de accesos a bloques necesario para leer los datos y los índices además de mostrar si esos datos se obtienen del disco o de la caché.

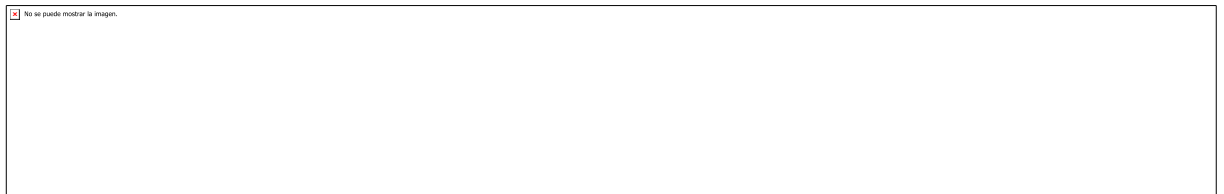
Para reinicializar los datos de la actividad de la base de datos ejecutamos la siguiente operación:



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



Una vez realizada la consulta, el paso siguiente es comprobar que índice ha usado PostgreSQL para la búsqueda de la tupla. Para ello utilizaremos el modulo **pg_stat_user_indexes**:



Como podemos observar, el índice seleccionada por PostgreSQL es el **índice_hash_id_cliente** ya que es el más óptimo para esta búsqueda ya que este índice esta ordenado según el campo que se esta buscado y además es un campo clave, por lo que solo necesitará realizar un acceso. La columna **idx_scan**, donde se indica las veces que se ha leído el índice, en este caso uno ya que solo hemos realizado una consulta, **idx_tup_read**, muestra el número de tuplas que se han recuperado de disco tras el acceso al índice como podemos ver en este caso es uno ya que el campo es clave y la columna **idx_tup_fetch**, muestra el número de tuplas leídas del índice, en este caso es uno ya que el campo es clave.

A continuación, vamos a comprobar cuantos bloques se han leído para realizar esta consulta, para ello utilizaremos el modulo **pg_statio_user_tables** :



Como podemos observar, se necesita leer 4 bloques de disco, como se muestra en la columna **idx_blks_read**, debido a que un bloque ya estaba cargado en caché como podemos ver en la columna **idx_blks_hit** y por último mostrado en la columna **heap_blks_read** podemos ver que se necesitó leer un bloque de disco para leer el dato de la tabla ya que es un campo clave. Por tanto, el coste total de lectura del índice es de 5 (4 de disco + 1 coste de lectura del dato (el acceso a caché no tiene coste)).

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



Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:



En este caso, PostgreSQL opta por utilizar el índice secuencial que crea por defecto cada vez que se crea una tabla, **MiTabla_pkey**, con el campo clave `id_cliente`. Como podemos ver en la columna **idx_tup_read**, lee 30001 tuplas que equivale al número de diferentes tuplas que vamos a leer. En este caso la columna **idx_tup_fetch**, es uno ya que solo necesita acceder a la primera tupla del índice porque el resto de valores van a continuación debido a que esta ordenada según ese campo.

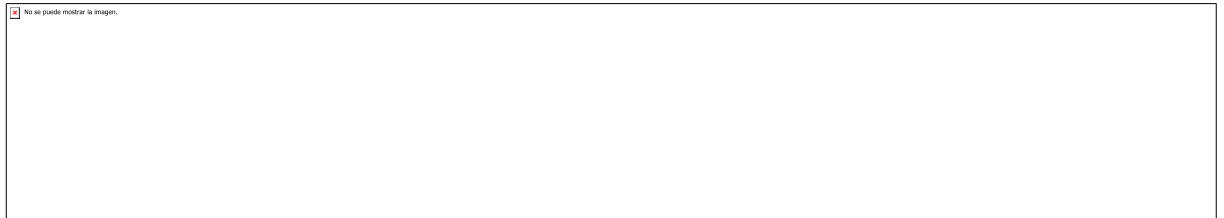


Como podemos observar, el número de bloques a los que necesita acceder para leer los datos de la tabla es más elevado que el anterior (27432 de disco + 52 de caché) porque necesita leer más tuplas diferentes y el número de accesos del índice es de 84 de disco y 3 de caché, presenta un coste superior al anterior debido a que es un índice secuencial por clave primaria y por lo tanto debe almacenar un puntero por cada tupla de la tabla. El coste total de la consulta es de $27432 + 84 = 27516$ bloques.

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



Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:

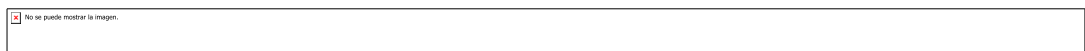


Como podemos ver, en este caso PostgreSQL al igual que en la anterior consulta ha optado por el índice secuencial para realizar la consulta pero esta vez ha tenido que realizar dos accesos al índice, uno por cada condición.



Como podemos observar, el número de bloques a los que necesita acceder para leer los datos de la tabla es mas elevado que el anterior (70152 de disco + 44 de caché) porque en este caso debe recuperar 92000 tuplas y el número de accesos del índice es de 253 de disco y 8 de caché, mas que el anterior ya que debe acceder dos veces al índice. El coste total de la consulta es de $70152 + 253 = 70405$ bloques.

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



Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:



Como podemos ver, en este caso PostgreSQL al igual que en la primera consulta ha optado por el índice hash para realizar la consulta debido a que es el mas eficiente a la hora de localizar un valor o un número de valores reducido.

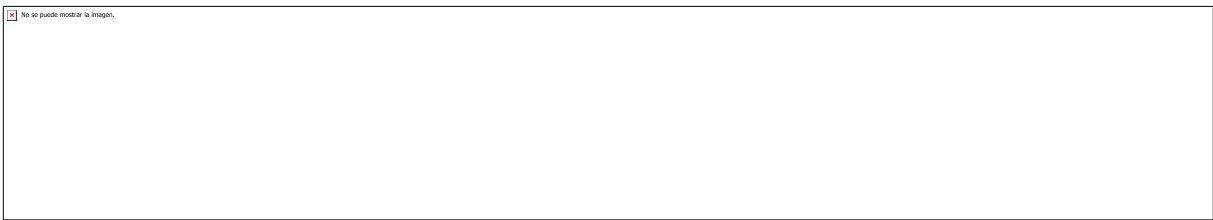


Como podemos observar, se necesita leer 1 bloques de disco, debido a que solo se realiza una lectura ya que la otra condición no se encuentra en la tabla y el coste del índice es bajo ya que solo debe acceder a un cajón hash (2 bloques). El coste total de la consulta es de $2 + 1 = 3$.

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



Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:



En este caso no utiliza ningun índice ya que en este caso lo más óptimo es recorrer todos los datos debido a que debe leer todos las tuplas a excepción de una.

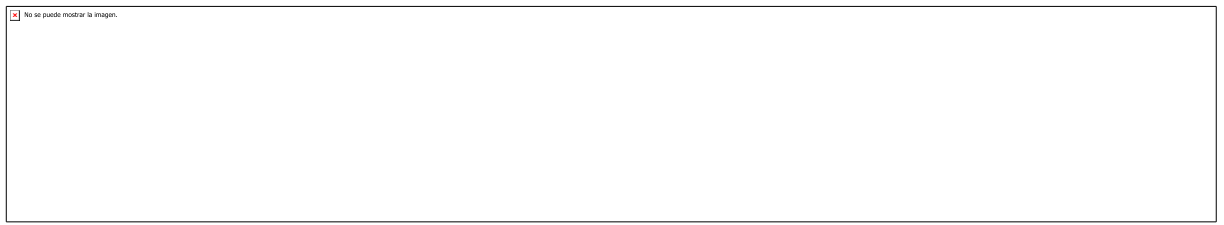


Como podemos observar a primera vista, no hay costes en el uso de accesos a índices ya que no se emplea ningún índice en este caso. Sin embargo, sí que se realizan una gran cantidad de accesos a bloques para leer tuplas, ya que tiene que leer prácticamente toda la tabla. Pero en este caso el número de bloques en caché es considerable ya que se han realizado consultas anteriores con estas tuplas. Por tanto, el coste total de esta consulta fue de 143686 bloques.

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



Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:



En este caso no escoge ningún índice ya que los índices creados no emplean el campo nombre.



Es similar a la consulta anterior pero en este caso debe acceder a todas las tuplas de la tabla por lo que el número de accesos a bloques del disco es un poco superior. Sin embargo, los accesos a caché no varían porque probablemente sea su tamaño máximo. El coste total de esta consulta fue de 143768 bloques.

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



Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:



En este caso emplea el índice árbol porque aunque el índice `_hash_puntos` sería mejor para valores individuales, en este caso existen muchas tuplas con el campo `puntos = 650` y por lo tanto el hash requiere de cajones de desbordamiento que reducen mucho su eficiencia.



Como podemos observar no se ha utilizado el acceso a los bloques de la memoria caché sobre el índice ya que es la primera vez que se emplea. También podemos observar que el coste de acceso al índice es de 61 y el coste de acceso a los datos en el disco es de 137 teniendo ya almacenado 91 en la memoria caché. El coste total de esta consulta es de $137 + 91 = 228$ bloques.

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



Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:



En este caso observamos que tampoco emplea ningún índice ya que tiene que leer una cantidad muy elevada de tuplas y por lo tanto es más eficiente

simplemente leer todas las tuplas que emplear alguno de los índices creados teniendo en cuenta que los índices de este campo son secundarios y no clave y por tanto todos los índices emplean cajones de punteros que los hacen menos eficientes, sobretodo a la hora de leer muchos datos.

No se puede mostrar la imagen.

No se puede mostrar la imagen.

Podemos ver que muestra datos muy similares a anteriores consultas donde no se emplean índices. El coste total es de 143760 bloques.

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

No se puede mostrar la imagen.

Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:

No se puede mostrar la imagen.

No se puede mostrar la imagen.

En este caso se emplea un índice árbol porque no se espera una gran cantidad de tuplas en la salida y porque es un intervalo de valores por tanto es mas eficiente que el índice hash_puntos.

No se puede mostrar la imagen.

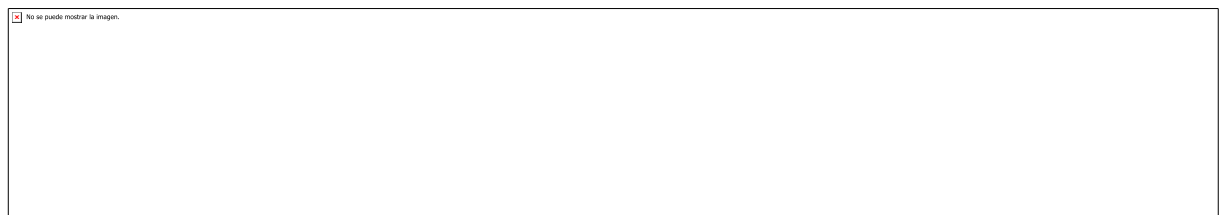
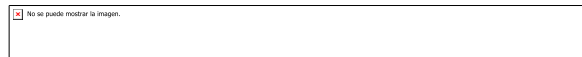
No se puede mostrar la imagen.

Como ninguna tupla cumple la condición no se realizan accesos al disco ni al índice. Sin embargo, como el árbol se utilizo recientemente sigue manteniendose en memoria caché donde vemos que ha realizado 6 accesos para intentar localizar las tuplas. El coste de esta consulta fue de 0.

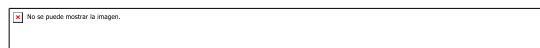
10. Mostrar la información de las tuplas con id_cliente=90000 o puntos=230



Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:



En este caso emplea dos índices, el índice_hash_id_cliente para localizar el campo id_cliente y el índice árbol para localizar el campo puntos. Esto es debido a que la condición es **or** y por tanto debe obtener todas las tuplas en las que cumplan tanto la primera como la segunda condición. Se utiliza el índice hash en el campo id_cliente ya que es muy eficiente a la hora de localizar campos clave concretos siempre que no haya demasiados cajones de desbordamiento y es el único que opera con ese campo y el índice árbol en vez del índice hash ya se mencionó anteriormente.



En este caso el coste de la consulta fue 231 (de los accesos a disco) + 63 (del coste de acceso de los índices que es la suma del coste de los dos índices empleados) = 294 bloques

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



Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:



En este caso solo se emplea el índice hash_id_cliente ya que es una condición **and** y uno de los dos valores es clave por lo tanto lo único que tendría que localizar es el campo clave con ese valor concreto y luego ver si cumple el resto de condiciones.



Como podemos observar el coste de acceso a disco es 1 y el acceso a índice es uno porque busca la tupla pero no existe ninguna tupla con esas condiciones por tanto el coste total es $1 + 1 = 2$ bloques.

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

Una vez eliminados los índices anteriores, realizamos la siguiente operación para crear un índice multiclave btree en la tabla **MiTabla**:



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:

Mediante la monitorización de los datos se puede información del índice o índices que decide emplear PostgreSQL en cada consulta (el más eficiente) y también nos aporta

más información del número de accesos a bloques necesario para leer los datos y los índices además de mostrar si esos datos se obtienen del disco o de la caché.

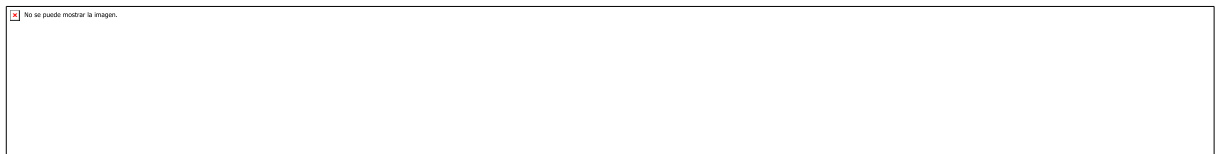
Para reinicializar los datos de la actividad de la base de datos ejecutamos la siguiente operación:



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



Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:



Como el árbol contiene índice para los dos valores que se están buscando solo se realiza un acceso al índice tal y como se muestra en la columna **idx_scan**.



Podemos observar que no se ha accedido al disco para recuperar la tupla ya que no existe ninguna tupla que cumpla las condiciones dadas tal y como muestra la columna **heap_blks_read**. Sin embargo, sí que se han realizado accesos al índice para comprobar la existencia de la tupla tal y como se puede ver en la columna **idx_blks_read**. Por tanto, el coste total de esta consulta fue de 5 bloques.

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

No se puede mostrar la imagen.

Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:

No se puede mostrar la imagen.

No se puede mostrar la imagen.

En este caso PostgreSQL no emplea ningún índice debido a que la cantidad de tuplas que va leer es muy elevado y por tanto le resulta mas eficiente leer todas las tuplas que utilizar el índice multiclave.

No se puede mostrar la imagen.

No se puede mostrar la imagen.

Como podemos observar, en esta consulta se han tenido que hacer muchos accesos a disco y memoria caché debido al volumen de tuplas que hay que leer y las columnas **idx_blks_read** y **idx_blks_hit** muestran o porque no se ha empleado ningún índice. Por tanto, como los bloques en la memoria caché no tienen coste de lectura, el coste total de la consulta fue 143912 bloques.

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

No se puede mostrar la imagen.

Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:

No se puede mostrar la imagen.

No se puede mostrar la imagen.

La condición es un **or** por tanto es lo mismo que realizar dos comprobaciones por separado y devolver los resultados de ambos, esto es poco óptimo para un índice multiclave. En este caso uno de los dos campos es campo clave, `id_cliente`, pero no emplea el índice **MiTabla_pkey** porque habría que combinarla con la consulta del campo nombre que es secundario no clave y obligaría emplear gran parte del índice multiclave para encontrar el nombre en concreto, por tanto, opta por leer el fichero secuencialmente.



Observamos que este caso es muy similar al caso anterior siendo el coste de la consulta 143878 bloques.

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



Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:



En este caso PostgreSQL decide utilizar el índice **MiTabla_pkey** ya que solo debe buscar el campo clave `id_cliente` y el índice multiclave implicaría un mayor coste porque almacena punteros para dos campos así que es menos eficiente si solo se necesita la información de uno de los dos campos.



En este caso, se lee el bloque en la memoria caché tal y como muestra la columna **heap_blks_read** y el acceso al índice le dentro de la memoria caché como se indica en la columna **idx_blks_hit**. El coste de esta consulta fue de 0 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:

La monitorización de los datos, en este caso, aporta información del número de accesos a bloques necesario para leer los datos y los índices además de mostrar si esos datos se obtienen del disco o de la caché.

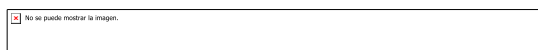
Para reinicializar los datos de la actividad de la base de datos ejecutamos la siguiente operación:



1. Mostrar las tuplas cuyos puntos valen 200.



Una vez realizada la consulta, veremos que particiones ha utilizado PostgreSQL para realizar la consulta:



Podemos observar que todas las tuplas referentes a esta consulta están almacenadas en la misma partición, **MiTabla3_1**, debido a que todas presentan el mismo valor para puntos y los cajones de la partición hash están distribuidos según los valores del campo puntos.

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



Una vez realizada la consulta, veremos que particiones ha utilizado PostgreSQL para realizar la consulta:



Debido a las condiciones de la consulta son imposibles, no se lee ninguna tupla.

3. Mostrar las tuplas cuyos puntos valen 200 o 202



Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:



Vemos que se guardan en particiones diferentes, debido a que el resto de módulo 10 es diferente para ambos valores y por tanto se guardan en diferentes particiones.

4. Mostrar las tuplas cuyos puntos son > 500 .

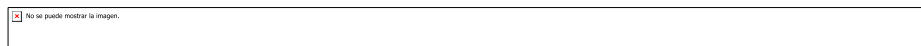


Una vez realizada la consulta, veremos que índice ha utilizado PostgreSQL para realizar la consulta:

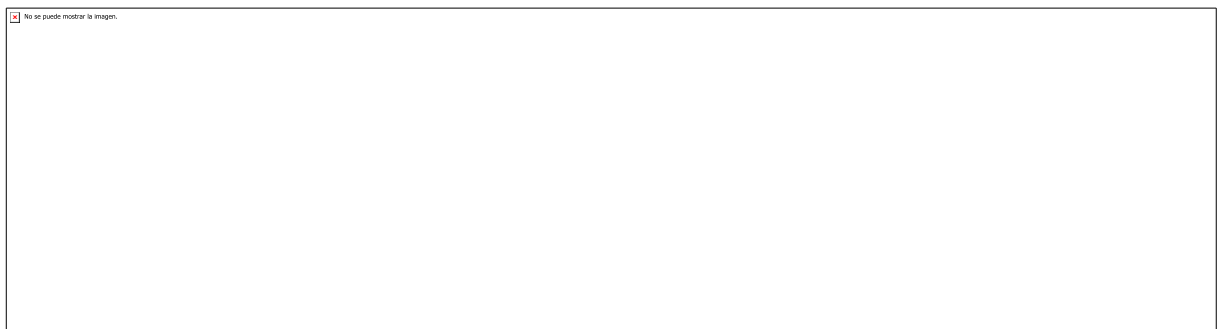


En esta consulta se leen todas las particiones debido a que lee todas las tuplas que presentan puntuaciones superiores a 500 y por tanto el resto del módulo 10 va desde 0 a 9 de una manera más o menos uniforme.

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



Una vez realizada la consulta, veremos que particiones ha utilizado PostgreSQL para realizar la consulta:

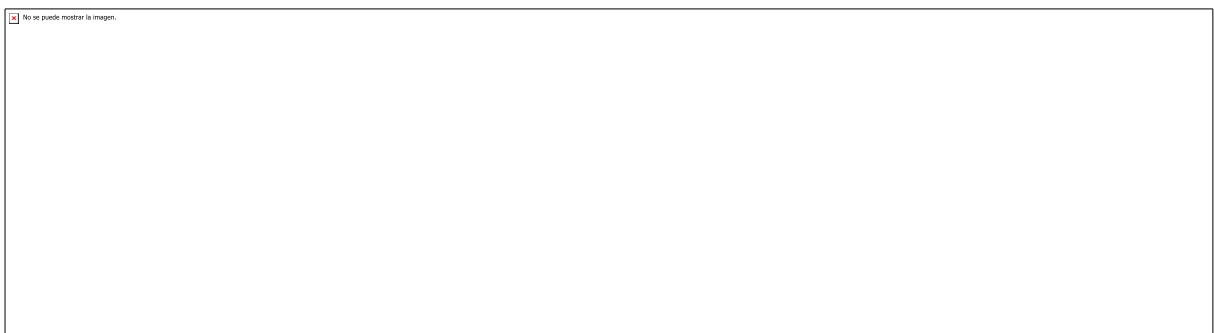


Este caso es similar al caso anterior debido a que va a necesitar leer 5 valores diferentes para cada una de las particiones.

6. Mostrar las tuplas cuyos puntos son 800



Una vez realizada la consulta, veremos que particiones ha utilizado PostgreSQL para realizar la consulta:



Este caso es similar al primero ya que necesitamos leer las tuplas correspondientes a un solo valor para el campo puntos y por lo tanto están almacenados en una sola partición en este caso **MiTabla3_6**.

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 podido ver en la realización de estas cuestiones, PostgreSQL es capaz de seleccionar el índice más efectivo o simplemente no optar por ninguno y leer toda la tabla en función de las previsiones que este realiza en base a sus datos estadísticos, reduciendo así considerablemente los costes de lectura de las consultas que el usuario realice de una forma muy versátil y eficaz. Además, también hemos podido comprobar que las particiones ayudan mucho a la hora de leer tuplas concretas o que se encuentren en una misma partición. Sin embargo, si se quiere leer intervalos o una cantidad de tuplas moderada simultáneamente, este procedimiento no es muy eficiente, así que será recomendable emplearlo en tablas con muchos valores en los que normalmente con cada consulta solo se leen un pequeño porcentaje de tuplas.

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.