GRADO EN INGENIERÍA INFORMÁTICA
2018-2019
CONVOCATORIA ORDINARIA DE ENERO
BASES DE DATOS AVANZADAS
–
LABORATORIO

PRACTICA 1: ARQUITECTURA POSTGRESQL Y ALMACENAMIENTO FÍSICO

ALUMNO 1:

Nombre y Apellidos: Marcos Barranquero Fernández

DNI: 51129104N

ALUMNO 2:

Nombre y Apellidos: Eduardo Graván Serrano

DNI: 03212337L

Fecha: 17/09/2018

Profesor Responsable: Iván González Diego

TABLA DE CONTENIDO

ALMACENAMIENTO FÍSICO EN POSTGRESQL

CUESTIÓN 1: CREAR UNA NUEVA BASE DE DATOS QUE SE LLAME **MIBASEDATOS**. ¿EN QUÉ DIRECTORIO SE CREA DEL DISCO DURO Y CUANTO OCUPA EL MISMO? ¿POR QUÉ?

Se crea en "C:\Program Files\PostgreSQL\10\data\base\30275" y ocupa 7,38 MB.

A pesar de no haber insertado ningún registro, no está vacía. Esto es debido a que se crean todos los archivos iniciales tales como el esquema público, estadísticas iniciales, etc.

CUESTIÓN 2: CREAR UNA NUEVA TABLA QUE SE LLAME **MITABLA** QUE CONTENGA UN CAMPO QUE SE LLAME CÓDIGO DE TIPO INTEGER QUE SEA LA PRIMARY KEY, OTRO CAMPO QUE SE LLAME NOMBRE DE TIPO TEXT, OTRO QUE SE LLAME DESCRIPCIÓN DE TIPO TEXT Y OTRA REFERENCIA QUE SEA DE TIPO INTEGER. ¿QUÉ FICHEROS SE HAN CREADO EN ESTA OPERACIÓN? ¿QUÉ GUARDAN CADA UNO DE ELLOS? ¿CUÁNTO OCUPAN? ¿POR QUÉ?

Al ejecutar:

```
CREATE TABLE public.mitabla
(
    codigo integer,
    nombre text,
    descripcion text,
    referencia integer,
    PRIMARY KEY (codigo)
)
WITH (
    OIDS = FALSE
);

ALTER TABLE public.mitabla
    OWNER to postgres;
```

Se han creado los ficheros 24611,24614, 24616, 24617. Usando el módulo oid2name con el siguiente comando obtenemos los siguientes resultados:

```
oid2name.exe -U postgres -d MiBaseDatos -f 3033X
```

FICHERO	ASOCIADO A	TAMAÑO	POR QUÉ
30330	mitabla	0 KB	Porque aún no contiene ningún dato.
30333	pg_toast_30276	0 KB	Porque no hay ninguna tupla en la tabla que exceda el tamaño de su bloque. De hecho, no hay ninguna tupla en la tabla.
30335	pg_toast_index	8 KB	Archivo de índices asociado al toast.
30336	mitabla_pk	8 KB	Archivo de claves primarias.

CUESTIÓN 3. INSERTAR UNA TUPLA EN LA TABLA. ¿CUÁNTO OCUPA AHORA LA TABLA? ¿SE HA PRODUCIDO ALGUNA ACTUALIZACIÓN MÁS? ¿POR QUÉ?

El archivo asociado a la tabla (30330) ocupa ahora 8 KB, debido a que se ha insertado una nueva tupla.
El archivo asociado a las primary keys (24617) ocupa ahora 16 KB, debido a la nueva tupla insertada.
Los otros dos siguen ocupando lo mismo, debido a que no se ha almacenado nada que exceda el límite de tamaño de bloque.

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? ¿POR QUÉ?

Tras crear el módulo pg_buffercache, ejecutamos lo siguiente:

```
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
```

Observamos que tenemos 102 tablas con distinta cantidad de buffers dedicados. Destacan, por ejemplo, la tabla pg_proc o la tabla pg_attribute por su gran cantidad de buffers dedicados. Sin embargo, nuestra tabla, "MiTabla", solo posee un buffer dedicado en estos momentos. Esto es debido a que tan sólo hemos insertado una tupla, por lo que no ha sido necesario más de un buffer para insertar dichos datos.

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?

Primero eliminamos la tabla. Después, la volvemos a crear. Para insertar los datos, elegimos insert y el archivo, marcando como delimitador el ";". Tarda 213 segundos en importar todos los datos.

La información original a insertar (el archivo datos_mitabla.txt) ocupa 574.245 KB.

Ejecutando la función para ver el tamaño de tabla:

```
select pg_relation_size('mitabla');
```

Observamos que la table ocupa 918642688 KB. (Unos 897112 MB). Vemos que hay bastante diferencia de tamaño. Esto es debido a la distinta forma de almacenar los datos y el formato de archivo. PostgreSQL almacena los datos asignando espacios fijos para los enteros y variables para los textos, frente a todo el formato texto que guarda el .txt.

CÁLCULO TEÓRICO DEL NÚMERO DE BLOQUES

Tenemos:

2 enteros en el registro * tamaño integer = 2 x 4 bytes/int = 8 bytes.

Tamaño medio del texto descripción * tamaño char = 18 chars de media * 1 byte/carácter = 18 bytes.

Tamaño medio del texto nombre * tamaño char = 18 chars de media * 1 byte/carácter = 15 bytes.

Tamaño medio del registro = (8 + 18 + 15) = 41 bytes/registro.

Los bloques de PostgreSQL son de 8192 bytes.

En un bloque caben, por tanto, (8192 bytes/bloque / 41 bytes/registro) = **200 registros/bloque.**

Si tenemos 12 millones de registros, debería haber (12000000 registros / 200 registros/bloque) = **60000 bloques de registros.**

VISTA REAL DEL NÚMERO DE BLOQUES

Vemos que **tenemos 112138 páginas (bloques)** almacenando nuestra tabla. Si tenemos 12 millones de registros, habrá unos 107 registros por bloque. Esto es debido a la cabecera de página, a ciertos elementos al final de página que también ocupan bytes, y a que los bloques no se rellenan completamente.

Por lo tanto, **no concuerda el cálculo teórico con el cálculo real** por las razones explicadas arriba.

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?

Observamos que ahora las relaciones “mitabla_pkey” y “mitabla” se llevan la mayoría de buffers:

	relname name	buffers bigint
1	mitabla_pkey	15929
2	mitabla	224
3	pg_proc	27
4	pg_proc_proname_args_nsp_index	13
5	pg_class	11
6	pg_attribute	8

Esto es debido a que hemos metido muchísimos datos en esas relaciones, y por tanto se han asignado más buffers a esas tablas. Deducimos que existe tal cantidad de buffers para el archivo asociado a las PK debido a que primero realizará las operaciones de inserción de datos en la tabla, y tras esto la asociación con el archivo de las PK. Por ello, primero, existirá gran cantidad de buffers a “mitabla”, y tras esto, se liberarán esos buffers para trabajar con “mitabla_pkey”.

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? ¿CUÁL ES EL FACTOR DE BLOQUE REAL DE LA TABLA? COMPARAR CON EL FACTOR DE BLOQUE TEÓRICO OBTENIDO.

Ejecutando:

```
select * from pgstattuple('mitabla')
```

Se muestran las estadísticas asociadas a las tuplas: el tamaño de tabla, la cantidad de tuplas que hay, la longitud de las tuplas, el porcentaje de tuplas vivas y muertas, etc.

Vemos que el porcentaje de espacio libre es un 0,47%. El porcentaje de ocupación es, por tanto, un 99,53%.

Quedan 4307312 bytes libres. Esto es debido a que los bloques no se completan con los datos.

Como hemos visto en la cuestión 5, teóricamente el factor de bloque debería ser 107 reg/bloque.

En la práctica, veíamos antes que en realidad hay una media de 199 registros/bloque.

Esto indica que los bloques no se encuentran ocupados al 100%, sino que hay algo de espacio libre por bloque.

CUESTIÓN 8 CON EL MÓDULO PAGEINSPECT, ANALIZAR LA CABECERA 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?

Ejecutando:

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

Vemos lo siguiente:

Primera página

Data Output | Explain | Messages | Notifications | Query History

	lsn pg_lsn	checksum smallint	flags smallint	lower smallint	upper smallint	special smallint	pagesize smallint	version smallint	prune_xid xid
1	0/9CB0...	0	0	452	504	8192	8192	4	0

Página intermedia

	lsn pg_lsn	checksum smallint	flags smallint	lower smallint	upper smallint	special smallint	pagesize smallint	version smallint	prune_xid xid
1	0/D631...	0	0	452	496	8192	8192	4	0

Página final

	lsn pg_lsn	checksum smallint	flags smallint	lower smallint	upper smallint	special smallint	pagesize smallint	version smallint	prune_xid xid
1	1/291E...	0	0	340	2504	8192	8192	4	0

Vemos ciertas similitudes, como el tamaño de bloque y la versión que utilizan, coinciden siempre. También, en la mayoría de páginas, lower y upper, parámetros que marcan el comienzo y final de espacio libre, son bastante similares.

Como hemos visto en la pregunta anterior, las páginas no se ocupan completamente. Por ello, la mayoría de páginas tienen algo de espacio libre.

En la última página, debido a que no está tan completa como el resto de páginas, existe más espacio libre. Por tanto, lower empieza antes y upper señala que hay más desplazamiento (espacio libre) hasta alcanzar el final de página o bloque.

CUESTIÓN 9. ANALIZAR LOS ELEMENTOS QUE SE ENCUENTRAN EN LA PÁGINA DEL PRIMER BLOQUE, DEL BLOQUE SITUADO EN LA MITAD DEL ARCHIVO Y DEL ÚLTIMO BLOQUE DE LA TABLA **MITABLA**. ¿QUÉ DIFERENCIAS SE APRECIAN ENTRE ESOS BLOQUES? ¿POR QUÉ?

Para ver los elementos de páginas utilizamos:

```
SELECT * FROM heap_page_items(get_raw_page('mitabla', numero_pagina));
```

Al ejecutarlo nos salen 107 registros.

El primer archivo muestra que está lleno desde la posición 504 hasta el final. Vemos que la longitud de los elementos es de 72 bytes, aunque algunos de ellos ocupan 68 bytes y dejan libres los 4 restantes.

En la página del medio, vemos que el desplazamiento a final de espacio libre comienza un poco antes, y por lo demás es similar a la página 0.

En la página final, sobresale que el desplazamiento a final de espacio libre (desde donde empieza a estar ocupada) es en la posición 2504, frente a las posiciones 496-504 de las páginas anteriores. Esto demuestra una vez más, que por ser la última aún no está tan llena como las demás. Al ejecutar el comando, nos salen tan solo 79 registros.

CUESTIÓN 10. CREAR UN ÍNDICE DE TIPO ÁRBOL PARA EL CAMPO CODIGO. ¿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 un índice de árbol sobre el campo código ejecutamos:

```
CREATE INDEX indice_codigo_arbol ON mitabla(codigo);
```

Tras unos segundos, tenemos nuestro índice creado. Se almacena físicamente en "C:\Program Files\PostgreSQL\10\data\base\30275" como 30327. Ocupa 263240 KB.

Llamando a la función pg_relpages del módulo pgstattuple:

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

Esto nos devuelve un total de **32905 bloques para el índice b-tree.**

Viendo las estadísticas, vemos que el árbol **tiene 2 niveles + raiz.**

La raíz se encuentra en la página 290.

En el nivel 1 tiene 117 páginas.

En el nivel 2 tiene 32787 páginas.

Para ver la información de las tuplas ejecutamos:

```
SELECT * FROM pgstatindex('indice_codigo_arbol');
```

Vemos que la media de tuplas de hojas son 90'09 tuplas/hoja.

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

CÁLCULO TEÓRICO

Para ver el número de bloques, sabemos que ocupa 263240 KB / 8 KB/bloque = **32905 bloques**.

Este resultado es exactamente igual al numero de bloques obtenidos en el apartado anterior.

En cuanto a los niveles:

Sabiendo que los punteros en postgre ocupan 16B, tenemos que el número de punteros necesarios para redireccionar los registros en nodos intermedios es de 228 punteros, mientras que en los nodos raíz necesitamos 409 punteros.

En el primer nivel (raíz) tenemos 1 nodo con 228 punteros = 227 registros.

En el segundo nivel tenemos 228 nodos con 228 punteros cada uno = 51756 registros.

En el nivel hoja tenemos 51984 nodos con 409 punteros cada uno = 21209472 registros.

Esto quiere decir que se necesitan un total de 2 niveles + el nivel raíz para poder almacenar todo el índice, lo cual es congruente con los resultados obtenidos en el apartado anterior.

CUESTIÓN 12. CREAR UN ÍNDICE DE TIPO HASH PARA EL CAMPO CÓDIGO Y OTRO PARA EL CAMPO REFERENCIA.

Introduzco en consola:

```
CREATE INDEX índice_codigo ON mitabla USING hash (codigo);
```

```
CREATE INDEX indice_referencia ON mitabla USING hash (referencia);
```

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?

Resultado de PGSTATTUPLE con el índice hash sobre código:

	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	40960	385	1	0	12000000	0	28.7927728462158

Resultado de PGSTATTUPLE con el índice hash sobre referencia:

	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	40960	29217	1	0	12000000	0	58.0480384360516

Última página índice hash código(41346):

	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	4	0	8192	8068	32701	4294967295	32700	1	65408

Última página índice hash referencia(70178):

Data Output | Explain | Messages | Notifications | Query History

	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	392	0	8192	308	70177	4294967295	40687	1	65408

Observando la ruta de la base de datos, vemos que el índice sobre referencia ocupa 561432 KB y el índice hash sobre código ocupa 330776 KB.

Esto se refleja también en el número de páginas que encontramos en los índices hash de referencia y código con el módulo pageinspect. (41346 vs 70178).

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

Al inspeccionar la última página en ambos, vemos que realmente no es la última, sino que es la última sin overflow. (Ambas comparten el mismo nº de ID en sus respectivas tablas). Sin embargo, el hash sobre referencia contiene muchísimas más páginas con overflow respecto al hash de código.

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

CUESTIÓN 14. ACTUALIZAR LA TUPLA CON CÓDIGO 7000020 PARA PONER LA REFERENCIA A 240. ¿QUÉ OCURRE CON LA SITUACIÓN DE ESA TUPLA DENTRO DEL FICHERO? ¿POR QUÉ?

En un primer momento el ct_id de la tupla con ese código es el siguiente: (69951,57)

Lo cual quiere decir que se encuentra en la página 69951 y es la tupla colocada en la posición 57.

Ahora ejecutamos:

```
UPDATE mitabla SET referencia = 240 WHERE codigo = 7000020;
```

Después de ejecutar esta consulta, el ct_id para la tupla con ese código es el siguiente: (112138,80)

Esto quiere decir que la tupla se ha movido dentro del archivo, ya que se encuentra en un bloque totalmente diferente, que resulta ser el último bloque del archivo.

Esto es debido al hacer update la tupla con el nuevo valor se inserta en la primera posición libre que encuentre, en este caso, el bloque final. Mientras que los valores de la posición que ocupada en el bloque antiguo son puestos a null, generando una tupla muerta.

```
--select * from mitabla where(codigo=7000020)
--UPDATE mitabla SET referencia = 240 WHERE codigo = 7000020;
--SELECT * FROM pgstattuple('mitabla');
SELECT * FROM mitabla WHERE(CODIGO>7000000 and codigo<7001000) order by
(codigo)
```

CUESTIÓN 15. BORRAR LAS TUPLAS DE LA TABLA MITABLA CON CÓDIGO ENTRE 7.000.OOO Y 8.000.000 ¿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?

Se ejecuta la sentencia:

```
DELETE from "MiTabla" where codigo>= 7000000 and codigo<=8000000
```

Si miramos la información que nos proporciona el módulo pageinspect pasándole como parámetro las paginas donde estaban almacenadas esas tuplas, vemos que los valores de las filas dentro de esas páginas que correspondían con el ctid de las tuplas se encuentran a null.

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

CUESTIÓN 16. INSERTAR UNA NUEVA TUPLA QUE CONTENGA LA INFORMACIÓN DE (7.500.010,PRODUCTO7500010,DESCRIPCION7500010,187). ¿EN QUÉ BLOQUE Y POSICIÓN DE BLOQUE SE INSERTA ESA TUPLA? ¿POR QUÉ?

Se ejecuta la sentencia:

```
INSERT INTO "MiTabla" VALUES(7500010, 'producto7500010', 'descripcion7500010', 187)
```

Comprobando el ct_id de la nueva tupla tenemos: (112138,81)

Es decir, se ha insertado en el ultimo bloque, en la posición siguiente a la tupla que fue actualizada en la cuestión 14.

Como ya se ha comentado en la cuestión anterior, esto es porque POSTGRESQL **no libera el espacio de los bloques directamente**, si no que estos han de ser compactados para que se libere el espacio y pueda ser reutilizado.

CUESTIÓN 17. EN LA SITUACIÓN ANTERIOR, ¿QUÉ OPERACIONES SE PUEDE APLICAR A LA BASE DE DATOS 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. ¿DÓNDE SE ENCUENTRA AHORA LA TUPLA DE LA CUESTIÓN 14 Y 16? ¿POR QUÉ?

Para liberar el espacio que están ocupando las tuplas muertas que han sido creadas por el update y el delete de los puntos anteriores se puede hacer uso de la utilidad vacuum. Se ejecuta la siguiente consulta:

Vacuum full

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

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

La tupla de la cuestión 14 fue borrada siguiendo los pasos de la cuestión 15, por lo que no se encuentra en la base de datos y por lo tanto no tiene ctid.

El ctid de la nueva tupla insertada en la cuestión 16 es: (102798,29)

Todo lo anterior nos indica que al realizar el vacuum, se han rellenado los espacios que dejaban las tuplas muertas con otras tuplas, liberando de esta forma el espacio que estas estaban ocupando.

CUESTIÓN 18. CREAR UNA TABLA DENOMINADA MITABLA2 DE TAL MANERA QUE TENGA UN FACTOR DE BLOQUE QUE SEA UN TERCIO QUE LA DE LA TABLA MITABLA Y CARGAR EL ARCHIVO DE DATOS ANTERIOR EXPLICAR EL PROCESO SEGUIDO Y QUÉ ES LO QUE OCURRE FÍSICAMENTE.

Se ejecuta la siguiente sentencia para crear dicha tabla:

```
CREATE TABLE public."MiTabla2"(
    codigo integer NOT NULL,
    nombre text COLLATE pg_catalog."default",
    descripcion text COLLATE pg_catalog."default",
    referencia integer,
    CONSTRAINT "MiTabla2_pkey" PRIMARY KEY (codigo))
WITH (
    OIDS = FALSE,
    FILLFACTOR = 33
)
TABLESPACE pg_default;
```

Tal y como esta especificado en la cláusula with, el factor de bloque es de un 33% ya que el predeterminado de postgre (con el que se ha creado la otra tabla) es del 100%.

Se han generado los archivos 30720, 30720.1 y 30720.2

Físicamente, lo que esta ocurriendo es que los bloques se están llenando a un 33% de su capacidad en comparación del 100% al que se llenan si no se especifica nada al crear la tabla en postgresql. Hay un total de **342858 bloques** para esta tabla. Para comparar, la primera tabla con la carga inicial de datos (sin haber borrado nada) tenía 112138 bloques.

CUESTIÓN 19. INSERTAR UNA NUEVA TUPLA QUE CONTENGA LA INFORMACIÓN DE (33.500.010, PRODUCTO3500010, DESCRIPCION13500010,185) EN LA TABLA MITABLA2. ¿EN QUÉ BLOQUE Y POSICIÓN DE BLOQUE SE INSERTA ESA TUPLA? ¿POR QUÉ?

Se ejecuta la sentencia:

```
INSERT INTO "MiTabla2" VALUES(33500010, 'producto3500010', 'descripcion13500010', 185);
```

Al consultar la tabla, devuelve el siguiente ctid: (342857,6)

Si comprobamos el numero de bloques totales descritos en la cuestión anterior, vemos que la inserción se ha producido en el último bloque. Al llamar al módulo pageinspect, podemos comprobar que la posición 6 dentro de dicho bloque es la última, por lo que esa inserción se ha producido al final de la tabla.

Esto es así ya que, a la hora de insertar, debido a que hemos declarado un fill factor del 33%, todos los bloques anteriores al ultimo cuentan como llenos desde el punto de vista de una inserción nueva. Por lo que, según estos parámetros, el primer bloque libre que encuentra para la inserción es el último.

CUESTIÓN 20. ACTUALIZAR LA REFERENCIA CON CÓDIGO 9.000.010 DE LA TABLA MITABLA2 PARA PONER LA REFERENCIA A 350 ¿QUÉ OCURRE CON LA SITUACIÓN DE ESA TUPLA DENTRO DEL FICHERO? ¿POR QUÉ?

Antes de realizar la consulta que se pide en el enunciado de la cuestión, el ctid de esta tupla es el siguiente: (171777,8)

Se ejecuta la consulta:

```
UPDATE "MiTabla2" SET referencia = 350 WHERE codigo = 9000010;
```

Después de esto, el ctid de esa tupla es el siguiente: (171777,36)

La tupla se encuentra en la última posición dentro del mismo bloque. La posición 8 del bloque se ha puesto a null y se ha convertido en una tupla muerta.

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

CUESTIÓN 21. A LA VISTA DE LAS PRUEBAS REALIZADAS ANTERIORMENTE, ¿SE PUEDE OBTENER ALGUNA CONCLUSIÓN SOBRE LA ESTRUCTURA DE LOS ARCHIVOS QUE UTILIZA POSTGRESQL Y EL MANEJO DE LAS TUPLAS DENTRO DE LOS ARCHIVOS?

La conclusión principal que se puede sacar gracias a estas últimas pruebas realizadas en las cuestiones anteriores es que se debe configurar el fill factor (factor de bloque) para ser inferior al 100%. El 33% definido para esta práctica es excesivo ya que triplica el tamaño de la base de datos sin ganar ningún beneficio aparente. Cuantos más updates se hagan en una tabla, menor deberá ser su fill factor, para que de esta forma se pueda hacer la inserción dentro del mismo bloque y no sea necesario actualizar los índices cada vez que se actualice el valor de alguna tupla.

Además, es importante liberar el espacio que dejan las tuplas muertas al hacer updates o deletes de la base de datos, ya que es esencialmente espacio perdido que puede ser reutilizado por otros bloques, reduciendo considerablemente el espacio que ocupa la base de datos.

MONITORIZACIÓN DE LA ACTIVIDAD DE LA BASE DE DATOS

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.

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

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

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

CUESTIÓN 23. CREAR UN ÍNDICE PRIMARIO BTREE SOBRE EL CAMPO REFERENCIA. ¿CUÁL HA SIDO EL PROCESO SEGUIDO?

Primero se crea el índice b-tree sobre el campo referencia usando la consulta:

```
CREATE INDEX indice_primario_arbol_referencia ON "MiTabla"(referencia);
```

Para hacer que este índice sea primario sobre el campo referencia, se debe ejecutar un comando cluster que modifique el orden físico del fichero con respecto al orden de este índice. Se ejecuta la siguiente consulta:

```
CLUSTER "MiTabla" using "indice_primario_arbol_referencia";
```

CUESTIÓN 24. CREAR UN ÍNDICE HASH SOBRE EL CAMPO REFERENCIA

Se ejecuta la consulta:

```
CREATE INDEX indice_hash_referencia ON "MiTabla" USING hash(referencia);
```

CUESTIÓN 25. CREAR UN ÍNDICE SOBRE EL CAMPO CÓDIGO DE TIPO BTREE Y OTRO DE TIPO HASH SOBRE EL MISMO CAMPO.

Para el índice btree:

```
CREATE INDEX indice_arbol_codigo ON "MiTabla"(codigo);
```

Para el índice hash:

```
CREATE INDEX indice_hash_codigo ON "MiTabla" USING hash(codigo);
```

CUESTIÓN 26. ANALIZAR EL TAMAÑO DE CADA ÍNDICE CREADO Y COMPARARLOS ENTRE SÍ. ¿QUÉ CONCLUSIONES SE PUEDEN EXTRAER DE DICHO ANÁLISIS?

La información se puede resumir con la siguiente tabla:

OID	Índice	Tamaño (bloques)
38485	Árbol sobre código	32.905
38486	Hash sobre código	41.347
38484	Hash sobre referencia	70.179
38475	Árbol primario sobre referencia	32.905

Los índices de tipo hash ocupan más bloques que los de tipo árbol.

Además, se observa que el índice en el cual se ha hecho cluster ocupa exactamente el mismo numero de bloques que el índice que se ha creado sobre la clave primaria de la tabla. Esto es debido a que, internamente, la base de datos añade un valor único a cada una de las referencias repetidas en la base de datos. Con esto se crea que el número de registros a indexar no son los valores diferentes de la columna referencia, si no todos los registros de la tabla. Esencialmente, al menos en este caso, crear un índice con cluster sobre un atributo no único es lo mismo que crear un índice sobre una clave primaria.

CUESTIÓN 27. PARA CADA UNA DE LAS CONSULTAS QUE SE MUESTRAN A CONTINUACIÓN, ¿QUÉ INFORMACIÓN SE PUEDE OBTENER DE LOS DATOS MONITORIZADOS POR LA BASE DE DATOS AL REALIZAR LA CONSULTA? ¿COMENTAR CÓMO SE HA REALIZADO LA RESOLUCIÓN DE LA CONSULTA? ¿CUÁNTOS BLOQUES SE HAN LEÍDO? ¿POR QUÉ? IMPORTANTE, REINICIALIZAR LOS DATOS RECOLECTADOS DE LA ACTIVIDAD DE LA BASE DE DATOS ANTES DE LANZAR CADA CONSULTA:

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

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

```
SELECT pg_stat_reset();
```

Se va a usar la siguiente expresion en SQL para cada una de las consultas:

```
EXPLAIN (buffers true, analyze true, format json)
```

Se tienen las siguientes consultas:

1. Mostar la información de las tuplas con codigo=9.001.000.

Para esta consulta se ha utilizado el índice de tipo hash declarado sobre código y se han leído un total de 2 bloques.

2. Mostar la información de las tuplas con codigo=90.001.000.

Se ha usado el índice de tipo hash declarado sobre código, se ha leído 1 bloque. Esto es debido a que ese valor no se encuentra en la tabla.

3. Mostrar la información de las tuplas con código<2000

Se ha usado el índice de tipo árbol definido sobre código, se han leído un total de 1997 bloques.

4. Mostrar el número de tuplas cuyo código >2000 y código <5000.

Se ha usado el índice de árbol definido sobre código, se han leído un total de 2796 bloques.

5. Mostrar las tuplas cuyo código es distinto de 25000.

Al tener que devolver toda la tabla menos 1 solo valor, esta vez la base de datos ha decidido usar la búsqueda secuencial sobre la tabla. Se han leído un total de 112.148 bloques, que coincide con el numero total de bloques de la tabla.

6. Mostrar las tuplas que tiene un nombre igual a 'producto234567'.

Ya que no había índices definidos sobre esta columna, se ha usado la búsqueda secuencial. En este caso, postgre ha realizado una búsqueda paralela sobre la tabla, lanzado 2 "Workers" que recorriesen la tabla. A parte de los 112.148 bloques totales de la tabla, se han producido otros 38.440 accesos a bloques, dando un total de 150.588 accesos a bloque.

7. Mostar la información de las tuplas con referencia=350.

Se ha usado el índice de árbol primario definido sobre la columna referencia. Se han leído un total de 360 bloques.

8. Mostrar la información de las tuplas con referencia<20.

Para esta consulta también ha usado el índice de árbol primario sobre referencia. Se han leído un total de 7.116 bloques.

9. Mostrar la información de las tuplas con referencia>300.

En este caso postgre ha decidido hacer una búsqueda secuencial sobre el archivo, leyendo un total de 112.148, es decir, ha leído toda la tabla.

10. Mostrar la información de las tuplas con codigo=70000 y referencia=200

Se ha utilizado el índice de tipo hash definido sobre codigo, se han leído un total de 2 bloques. Esto es debido a que código es la primary key de la tabla, por lo que no hace falta buscar por referencia.

11. Mostrar la información de las tuplas con codigo=70000 o referencia=200

Se ha utilizado el índice hash para buscar sobre codigo, y el índice árbol sobre referencia para buscar los valores de referencia que cumpliesen la condición. Una vez se han encontrado estos valores que coincidiesen, se han juntado realizando un OR sobre el bitmap combinado y se ha buscado en la tabla usando dicho bitmap. Se han leído un total de 69 bloques en la búsqueda de índices (1 de codigo y 68 de referencia) y 295 en la tabla, haciendo un total de 364 bloques accedidos.

CUESTIÓN 28. BORRAR LOS 4 ÍNDICES CREADOS Y CREAR UN ÍNDICE MULTICLAVE BTREE SOBRE LOS CAMPOS REFERENCIA Y PRODUCTO.

Se borran los índices y se ejecuta el siguiente comando para crear el nuevo índice:

```
CREATE INDEX indice_multiclave_arbol ON "MiTabla" USING btree(referencia, nombre);
```

CUESTIÓN 29. 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:

Al igual que con la cuestión 27, se llamará a la función que reinicia las estadísticas almacenadas por la base de datos con cada consulta. La consulta de explain será exactamente igual.

Se tienen las siguientes consultas:

1. Mostrar las tuplas cuya referencia vale 200 y su nombre es producto6300031.

Postgre ha usado el índice multiclave creado. Se han hecho un total de 5 accesos a bloque.

2. Mostrar las tuplas cuya referencia vale 200 o su nombre es producto6300031.

En este caso, postgre ha usado búsqueda secuencial sobre la tabla. Al igual que con la consulta numero 6 de la cuestión 27, postgre ha lanzado 2 workers para hacer un acceso paralelo de la tabla. Se han leído un total de 152.004 bloques.

3. Mostrar las tuplas cuyo código vale 6000 y su nombre es producto6300031.

Postgre construye un índice sobre la clave primaria de una tabla de forma automática como forma de agilizar las consultas sobre claves primarias. Este es el índice que se ha utilizado para esta consulta. Se han leído un total de 2 bloques. No se ha tenido que hacer búsqueda sobre nombre ya que, como se explica en la consulta 10 de la cuestión 27, codigo es la primary key de la tabla y solo hay que comprobar que se satisfaga la condición de nombre sobre una única tupla.

4. Mostrar las tuplas cuyo código vale 6000 o su nombre es producto6300031.

Se ha realizado búsqueda secuencial con 2 workers, realizándose un total de 151.659 accesos a bloque.

CUESTIÓN 30. A LA VISTA DE LOS RESULTADOS OBTENIDOS DE ESTE APARTADO, COMENTAR LAS CONCLUSIONES SE PUEDEN OBTENER DEL ACCESO DE POSTGRESQL A LOS DATOS ALMACENADOS EN DISCO.

Una de las conclusiones a las que se llega es la importancia del buffer que tiene la base de datos, ya que, aunque no se hayan reflejado estos datos en las respuestas de las cuestiones anteriores, se podía ver una diferencia considerable en tiempo de ejecución cuando tenía que leer de disco al realizar una consulta y cuando los datos que se requerían ya se encontraban en memoria.

Es muy importante la creación de índices sobre los atributos de la tabla que se basen en consultas que se puedan dar en la realidad para de esta forma intentar evitar la búsqueda secuencial dentro del fichero, ya que esta es la mas costosa en todos los casos.

También cabe remarcar que, aunque se hayan definido índices sobre un cierto atributo de una tabla, estos no siempre serán usados dependiendo de la evaluación sobre la eficiencia que haga postgre sobre la consulta. Esto refuerza el punto planteado en el párrafo anterior.

En definitiva, es importante definir índices adecuados para las tablas de una base de datos, pero siempre teniendo cuidado de no caer en redundancias, ya que estos índices ocupan bastante espacio en disco.

ANEXO – MÓDULOS Y PROGRAMAS

Anexo de referencia al trabajo realizado.

OID2NAME

Oid2name permite ver el OID de tablas y archivos, y ver información de las mismas.

PG_BUFFERCACHE

Este módulo permite consultar el buffer de caché compartido en tiempo real. Para crear esta extensión debemos escribir:

```
create extension pg_buffercache
```

Una vez ejecutado, podremos realizar las consultas a los buffers.

INFORMACIÓN ÚTIL SOBRE BLOQUES

"Lo primero que tenemos que decir es que la unidad mínima de almacenamiento en PostgreSQL se denomina, indistintamente, página (page) o bloque (block). Un bloque en PostgreSQL ocupa siempre por defecto 8K si no se ha definido un valor diferente durante la compilación. Esto independientemente de si se usa en su totalidad o solo parcialmente."

Consultar número de páginas/bloques:

```
SELECT * FROM pg_relpages('"pg_class"');
```

PGSTATTUPLE

Devuelve información sobre las tuplas, tales como los tamaños de tupla, porcentajes de uso, etc.

Para crearla, ejecutamos:

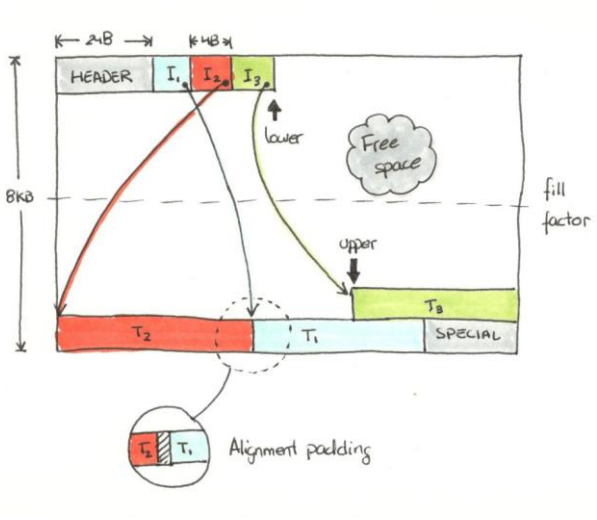
```
CREATE EXTENSION pgstattuple;
```

Para consultar una tabla, ejecutamos:

```
SELECT * FROM pgstattuple('"Nombre_de_tabla"');
```

PAGEINSPECT

Este módulo proporciona información sobre los bloques a bajo nivel. El formato de un bloque en PostgreSQL es el siguiente:



Consultar cabecera:

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

FUENTES

Toda la documentación oficial de PostgreSQL: https://www.postgresql.org/docs/10/static/index.html

Información sobre bloques: https://e-mc2.net/es/donde-estan-nuestros-datos-en-el-disco

Pageinspect: https://www.postgresql.org/docs/10/static/pageinspect.html

Pgstattuple: https://www.postgresql.org/docs/9.5/static/pgstattuple.html

Pgbuffercache: https://www.postgresql.org/docs/9.1/static/pgbuffercache.html

Tipos de datos: https://www.postgresql.org/docs/9.1/static/datatype-numeric.html

Tamaños de DB e índices: http://www.postgresqltutorial.com/postgresql-database-indexes-table-size/

Cluster: https://www.postgresql.org/docs/10/static/sql-cluster.html

A parte de esto, se han consultado foros como:

https://stackoverflow.com

https://dba.stackexchange.com/

https://www.sqlservercentral.com