Titulación: **Grado en Ingeniería Informática y Sistemas de Información**
Curso: **2018-2019. Convocatoria Ordinaria de Enero**
Asignatura: **Bases de Datos Avanzadas – Laboratorio**

Practica 1: Arquitectura PostgreSQL y almacenamiento físico

ALUMNO 1:

Nombre y Apellidos: Álvaro de las Heras Fernández

DNI: 03146833L

ALUMNO 2:

Nombre y Apellidos: Luis Alejandro Cabanillas Prudencio

DNI: 04236930P

Fecha: **21/10/2018**

Profesor Responsable: Iván González

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 puntuará **TODA** la práctica como Suspenso – Cero.

Plazos

Trabajo de Laboratorio: Semana 17 Septiembre, 24 Septiembre, 1 de Octubre, 8 de Octubre y 15 de Octubre.

Entrega de práctica: Día 22 de Octubre. 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 10.4. 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 y cuanto ocupa el mismo? ¿Por qué?

Respuesta

Postgresql se encarga de almacenar sus tablas en el archivo de programa junto a todos los datos y bases de datos del sistema, para poder identificar nuestra base de datos basta con saber su OID que podemos obtener del catálogo de nuestra base de datos. Una vez tenemos el OID, véase *Imagen 1*, basta con encontrar el archivo con el mismo nombre que se corresponderá con la base de datos, que en este caso es 16393.

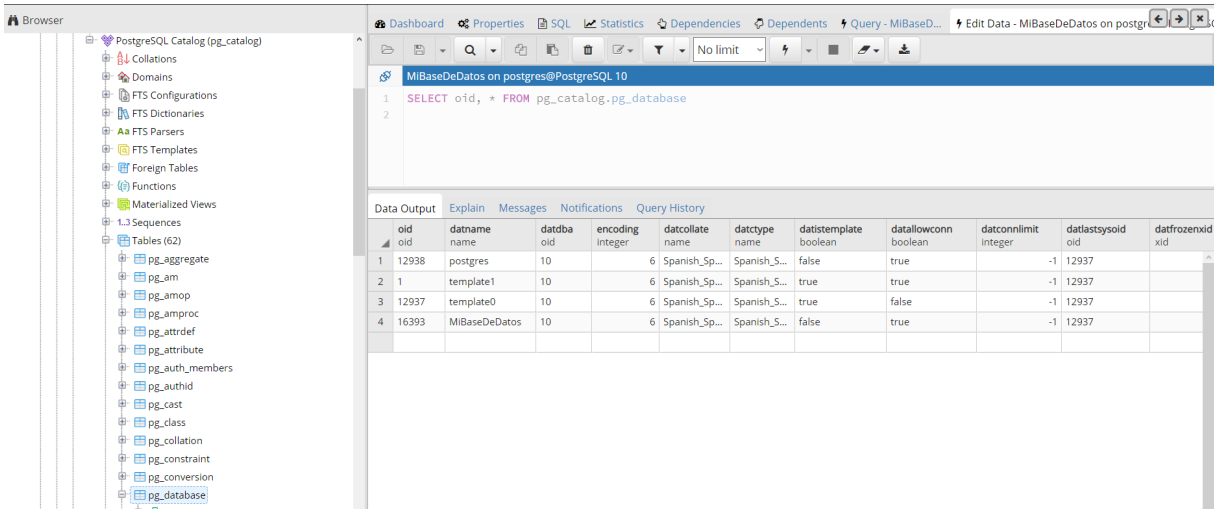


Imagen 1. Identificación de OID

En este caso el tamaño de nuestra base de datos es el que se muestra en la *Imagen 2*. Este tamaño se debe primero a las tablas que crea Postgresql internamente para manejar nuestra base de datos, estas las podemos consultar en el catálogo. Además hemos seleccionado la template1 para que introduzca ya más herramientas que las básicas.

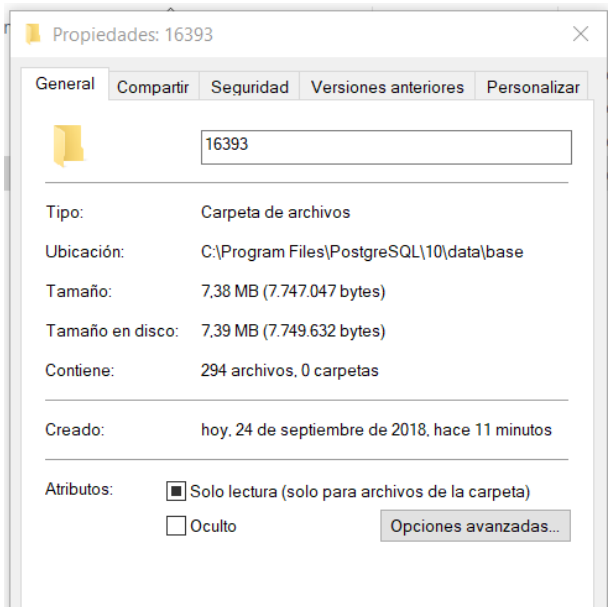


Imagen 2. Tamaño en disco de la base de datos

Por tanto hemos podido identificar nuestra base de datos en el disco por su OID y hemos determinado

que la diferencia de tamaño se debe a las propias tablas que crea Postgresql y el uso de la plantilla.

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?

Respuesta

Se han creado 4 archivos dos tablas que se corresponden a la tabla y su primary key (índice), y otras dos de tipo TOAST[1] una índice y otra para los datos de los campos de longitud variable. Las hemos identificado consultando sus OID como se ve en la *Imagen 3*.

	oid oid	relname name	relnamespace oid	reltype oid	reloftype oid	relowner oid	relam oid	relfileno oid
1	16397	pg_toast_16394	99	16398	0	10	0	16397
2	16399	pg_toast_16394_ind...	99	0	0	10	403	16399
3	16394	MiTabla	2200	16396	0	10	0	16394
4	16400	MiTabla_pkey	2200	0	0	10	403	16400
5	2619	pg_statistic	11	11258	0	10	0	2619
6	1247	pg_type	11	71	0	10	0	0
7	2830	pg_toast_2604	99	11515	0	10	0	2830
8	2831	pg_toast_2604_index	99	0	0	10	403	2831
9	2832	pg_toast_2606	99	11516	0	10	0	2832
10	2833	pg_toast_2606_index	99	0	0	10	403	2833
11	2834	pg_toast_2609	99	11517	0	10	0	2834
12	2835	pg_toast_2609_index	99	0	0	10	403	2835
13	2836	pg_toast_1255	99	11518	0	10	0	0
14	2837	pg_toast_1255_index	99	0	0	10	403	0

Imagen 3. OID y elementos creados

La primera guarda la tabla los datos en general, la segunda guarda los indices puesto que es un constraint de forma que así verifica que no hay duplicados al haber definido la clave, una tabla TOAST almacenará los datos de tipo text al ser variables en tamaño mientras que la otra actuara de índice para estos.

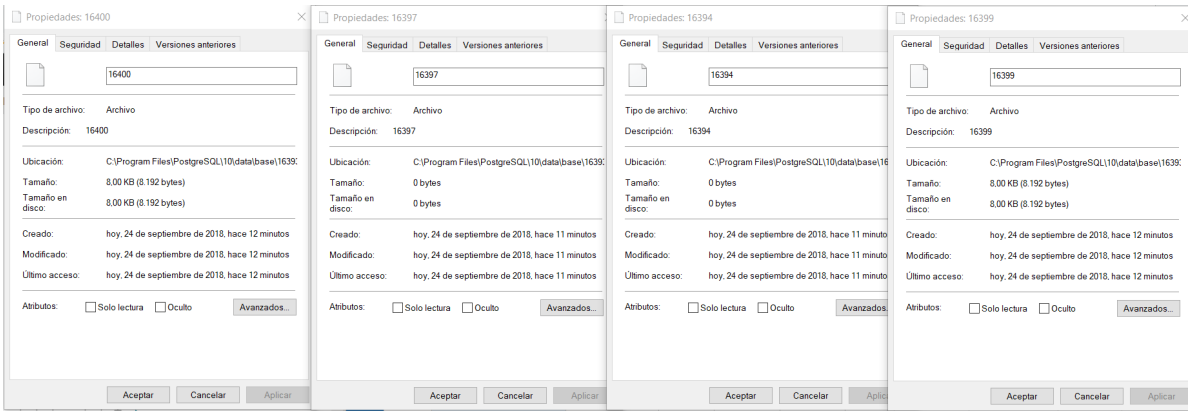


Imagen 4. Tamaño de los archivos creados

1 **TOAST:** The Oversized-Attribute Storage Technique

Los archivos relativos al índice y la tabla ocupan 8KB y 0KB respectivamente y los archivos de tipo TOAST 8KB para el índice y 0KB para los datos como se puede comprobar en la *Imagen 4*.

La razón de que las tablas que se encarguen de contener los datos valgan 0KB es que están vacías y no tienen ningún dato dentro. Mientras que las que ocupan 8KB han sido creadas con condiciones para asegurarse de la no duplicidad de índices.

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

Respuesta

Al volver a consultar el tamaño tras la inserción este ha aumentado en el archivo de los datos de la tabla que ahora tiene 8KB lo correspondiente con un bloque. También el índice de MiTabla tiene 8KB más lo que indica que ha cogido un bloque más de memoria, que hace que tenga 16KB. Mientras que las tablas de TOAST siguen con los mismos tamaños que antes.

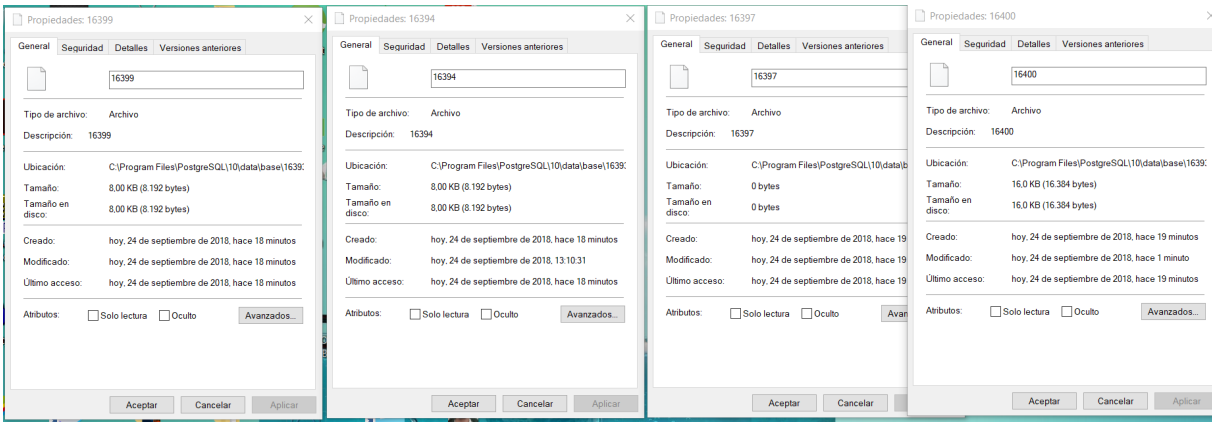


Imagen 5. Nuevos tamaños tras inserción

Esto se debe a que al insertar datos en la tabla Postgresql ha asignado un nuevo bloque que no se ocupará al completo para almacenar estos nuevos datos, es por eso que haya aumentado en 8KB los ficheros relativos a la tabla. Sin embargo, los ficheros de TOAST siguen igual porque no ha sido necesario el almacenamiento de datos por ser demasiado grandes.

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?

Respuesta

Al aplicar el módulo pg_buffercache nos crea una vista con los archivos de postgresql que hay en memoria incluyendo nuestra base de datos. Lo que muestra es lógico si pensamos que PostgreSQL emplea tablas para manejar nuestra base de datos por lo que es lógico que se encuentren cargadas en memoria, especialmente si ocupan poco espacio. Simplemente buscando entre todos los OID que hay en memoria encontramos el de nuestra base de datos lo que indica que esta cargado en memoria, véase *Imagen 6*. Esto es especialmente útil para ahorrar accesos a disco al realizar consultas al tener ya cargado éstos en memoria.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT bufferid, relfilenode, reltablespace, reldatabase, relforknumber, relblocknumber, isdirty, usagecount, pinni
    FROM public.pg_buffercache
    where reldatabase=16393;
```

Data Output | Explain | Messages | Notifications | Query History

	bufferid integer	relfilenode oid	reltablespace oid	reldatabase oid	relforknumber smallint	relblocknumber bigint	isdirty boolean	usagecount smallint	pinning_backends integer
1	229	1259	1663	16393	0	0	false	5	0
2	230	1259	1663	16393	0	1	false	5	0
3	231	1259	1663	16393	0	2	false	5	0
4	232	1249	1663	16393	0	0	false	5	0
5	233	1249	1663	16393	0	1	false	5	0
6	234	1249	1663	16393	0	2	false	5	0
7	235	1249	1663	16393	0	3	false	5	0
8	236	1249	1663	16393	0	4	false	5	0
9	237	1249	1663	16393	0	5	false	5	0
10	238	1249	1663	16393	0	6	false	5	0
11	239	1249	1663	16393	0	7	false	5	0
12	240	1249	1663	16393	0	8	false	5	0
13	241	1249	1663	16393	0	9	false	5	0
14	242	1249	1663	16393	0	10	false	5	0
15	243	1249	1663	16393	0	11	false	5	0
16	244	1249	1663	16393	0	12	false	5	0

Imagen 6. MiBaseDeDatos cargada en memoria

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?

Respuesta

El tamaño de los datos antes de insertarlo se corresponde con 560 MB como se puede ver en la Imagen 7.

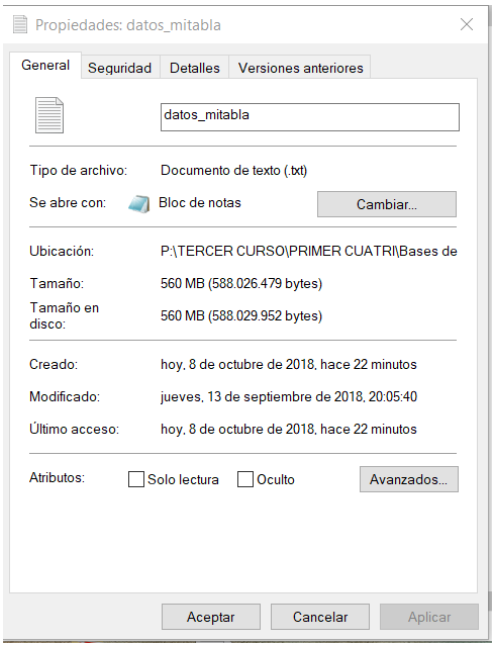


Imagen 7. Tamaño de los datos.

Tras insertar los datos en el sistema nuestra tabla pasa a ocupar 876MB y el archivo del campo clave 334 MB como se puede ver en la *Imagen 8* e *Imagen 9*.

Esto se debe a que al tener un determinado factor bloque pueden quedar espacios libres que se van acumulando sin ser aprovechados aumentando así el tamaño. También PostgreSQL añade información de control a la tabla por lo que acaba ocupando más. Otro factor es que el almacenamiento de un tipo de datos en un fichero de texto es menor que en una tabla, por ejemplo un entero como '3' ocupa un byte en el fichero de texto y ocupa 4 bytes en la tabla.

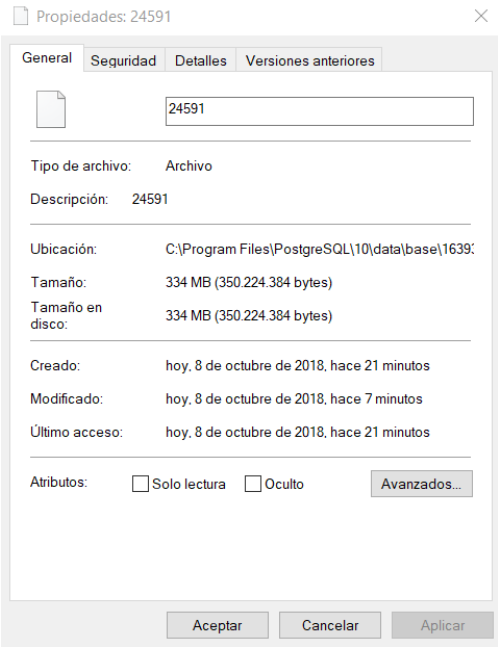


Imagen 8. Tamaño de la tabla de la clave.

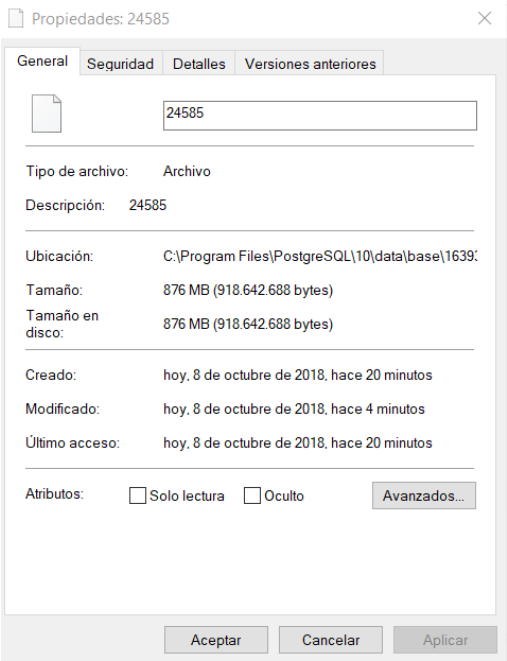


Imagen 9. Tamaño de la tabla.

Datos

Cálculo teórico

$B = 8192\ bytes$ (obtenido del archivo conf)

$L_R = L_{codigo} + L_{nombre} + L_{descripcion} + L_{referencia} + L_{control}$

Los campos de texto son las medias estadísticas del campo y el tamaño de control se ha obtenido del layout de la página.

$L_R = 4 + 16 + 19 + 4 + 24 = 67\ bytes$

$f_r = \left\lfloor \frac{B}{L_R} \right\rfloor \quad f_r = \left\lfloor \frac{8192}{67} \right\rfloor = 122\ registros/bloque$

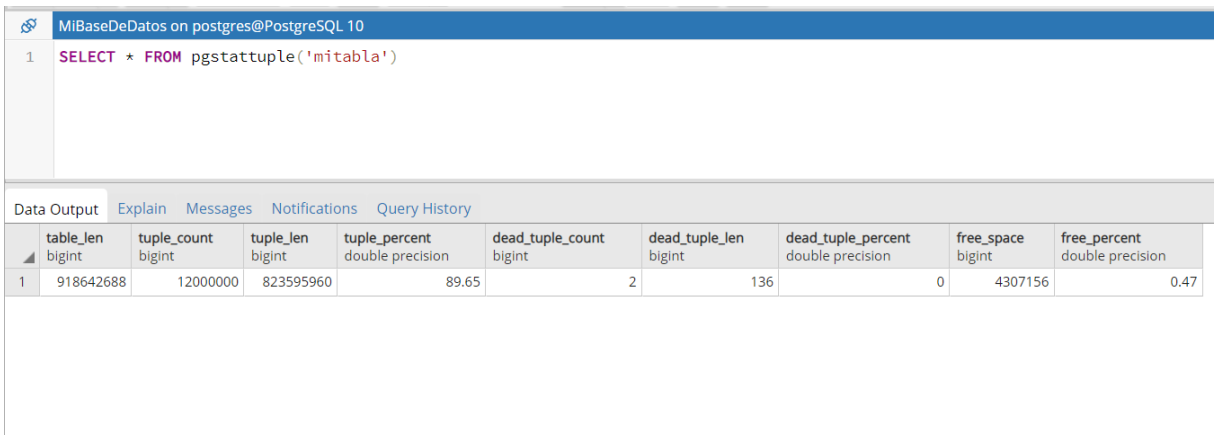
$n_b = \lceil \frac{n_r}{f_r} \rceil = \lceil \frac{12000000}{122} \rceil = 98361\ Bloques$

Tamaño en bloques de PostgreSQL

$L_{tabla} = 918642688\ bytes \quad B = 8192\ bytes$ El total se ha conseguido con pg_stattuple

$n_b = \frac{918642688}{8192} = 112139\ bloques$

No concuerda con el número bloques. Esto se debe a que las tuplas contienen espacios vacíos y algún byte más de control. PostgreSQL se encarga de asignar el espacio que cree conveniente, esto lo hace por heurística, para comprobar esta afirmación basta con consultar el porcentaje de tuplas que tiene la tabla. Para ello empleamos el módulo pg_stattuple que nos muestra que el porcentaje de tuplas es del 89.65% (véase *Imagen 10*) que encaja con nuestro resultado teórico, confirmando nuestra suposición.



	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	918642688	12000000	823595960	89.65	2	136	0	4307156	0.47

Imagen 10. Uso del módulo pg_stattuple

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?

Respuesta

De los datos que se muestran se puede deducir que la tabla solo tiene cargados unos pocos valores de ella en si mientras que estan casi todos los valores del indice como se muestra en las capturas

Tras hacer un análisis de los datos más relevantes que hay cargados en memoria se puede deducir que casi todo el índice está cargado en memoria. Mientras que apenas hay registros de datos cargados como se ve en la *Imagen 7* e *Imagen 8,* con 186 páginas para los datos y 15644 páginas para las claves.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT relfilenode
FROM pg_buffercache
WHERE relfilenode = 24585;
```

Data Output | Explain | Messages | Notifications | Query Histor

	relfilenode oid
170	24585
171	24585
172	24585
173	24585
174	24585
175	24585
176	24585
177	24585
178	24585
179	24585
180	24585
181	24585
182	24585
183	24585
184	24585
185	24585
186	24585

Imagen 7. Páginas correspondientes a la tabla de datos

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT relfilenode
FROM pg_buffercache
WHERE relfilenode = 24591;
```

Data Output | Explain | Messages | Notifications | Query History

	relfilenode oid
15628	24591
15629	24591
15630	24591
15631	24591
15632	24591
15633	24591
15634	24591
15635	24591
15636	24591
15637	24591
15638	24591
15639	24591
15640	24591
15641	24591
15642	24591
15643	24591
15644	24591

Imagen 8. Páginas correspondientes a la tabla de la clave

PostgreSQL realiza esto porque es mucho más eficiente acceder por la clave a los datos que por el resto de valores. Es por eso por lo que solo carga ese índice con el que trabaja para así optimizar las consultas, al menos la mayoría que empleen su PK. Aunque en caso de usar otro valor de búsqueda el coste aumentará ya que tendrá que ir cargando en memoria los bloques de tuplas. Sin embargo, PostgreSQL por heurística ya sabe que la mayoría de consultas emplearán la clave.

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.

Respuesta

En las estadísticas se muestran todos los datos relativos a las tuplas de nuestra tabla y sus bloques. En este caso nos informa sobre el espacio en bytes, tuplas vivas y tuplas muertas como se ve en la anterior *Imagen 10*.

El grado de ocupación de las tuplas es el espacio libre que nos lo da como free_percent por lo que será el espacio disponible que nos sirve para obtener el ocupado que sería de 89.65%.

El espacio que libre lo muestra en free_space en bytes, que tiene un valor 4307156 bytes.

Porque no es posible ocupar todo el espacio de un bloque sin particionar tuplas siendo este espacio libre la suma de todos estos espacios además del espacio que PostgreSQL añade extra. El factor de bloque teórico que hemos calculado anteriormente es de 122 registros/bloque mientras que el factor de bloque real es de:

$$f_r = \lceil \frac{12000000}{112139} \rceil = 108\, registros/bloque$$

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?

Respuesta

Lo primero que tenemos hacer es calcular el número de bloques que tendrán nuestras páginas. Para ello cogemos el número máximo de registros que entran en un bloque, siendo el primer bloque el elegido que nos da un factor de bloque de 107. Al dividir los 12 millones de registros entre el factor obtenemos un total de 112139 bloques que contendrán las páginas.

MiBaseDeDatos on postgres@PostgreSQL 10

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

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	0/8E098AA0	0	0	452	504	8192	8192	4	0

Imagen 11. Análisis de cabecera de la primera página.

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 SELECT * FROM page_header(get_raw_page('mitabla', 56069));
```

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/C7843610	0	0	452	496	8192	8192	4	0

Imagen 12. Análisis de la cabecera de la página del bloque intermedio.

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 SELECT * FROM page_header(get_raw_page('mitabla', 112138));
```

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	1/1AB9F980	0	0	340	2504	8192	8192	4	0

Imagen 13. Análisis de la página del último bloque.

Una vez hecho eso solo hay que observar las diferencias que son los valores que cambian entre páginas son el lsn que es el siguiente byte tras el xlog, el upper que indica el offset al final del espacio libre y el lower que indica el offset del comienzo de la memoria libre. Todas estas diferencias se aprecian en la *Imagen 11*, *Imagen 12* e *Imagen 13*.

Estas diferencias están porque en cada bloque van cambiando la cantidad de datos y su longitud. En este caso el último tiene más espacio porque el bloque no se ocupa al completo, por eso su lower es menor; mientras que en la primera e intermedia es el mismo al estar ocupada al máximo, variando únicamente el upper porque éste depende de la longitud de los campos variables de la tupla, que son los text.

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?

Respuesta

MiBaseDeDatos on postgres@PostgreSQL 10

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

Data Output | Explain | Messages | Notifications | Query History

	lp smallint	lp_off smallint	lp_flags smallint	lp_len smallint	t_xmin xid	t_xmax xid	t_field3 integer	t_ctid tid	t_infomask2 integer	t_infomask integer	t_hoff smallint	t_bits text	t_oid oid	t_data bytea
91	91	1656	1	68	578	0	0	(0,91)	4	2306	24	[null]		[binary da
92	92	1584	1	68	578	0	0	(0,92)	4	2306	24	[null]		[binary da
93	93	1512	1	68	578	0	0	(0,93)	4	2306	24	[null]		[binary da
94	94	1440	1	68	578	0	0	(0,94)	4	2306	24	[null]		[binary da
95	95	1368	1	72	578	0	0	(0,95)	4	2306	24	[null]		[binary da
96	96	1296	1	72	578	0	0	(0,96)	4	2306	24	[null]		[binary da
97	97	1224	1	68	578	0	0	(0,97)	4	2306	24	[null]		[binary da
98	98	1152	1	68	578	0	0	(0,98)	4	2306	24	[null]		[binary da
99	99	1080	1	68	578	0	0	(0,99)	4	2306	24	[null]		[binary da
100	100	1008	1	68	578	0	0	(0,100)	4	2306	24	[null]		[binary da
101	101	936	1	72	578	0	0	(0,101)	4	2306	24	[null]		[binary da
102	102	864	1	72	578	0	0	(0,102)	4	2306	24	[null]		[binary da
103	103	792	1	68	578	0	0	(0,103)	4	2306	24	[null]		[binary da
104	104	720	1	68	578	0	0	(0,104)	4	2306	24	[null]		[binary da
105	105	648	1	68	578	0	0	(0,105)	4	2306	24	[null]		[binary da
106	106	576	1	68	578	0	0	(0,106)	4	2306	24	[null]		[binary da
107	107	504	1	68	578	0	0	(0,107)	4	2306	24	[null]		[binary da

Imagen 14. Análisis de los elementos del primer bloque.

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 SELECT * FROM heap_page_items(get_raw_page('mitabla', 56069));
```

Data Output | Explain | Messages | Notifications | Query History

	lp smallint	lp_off smallint	lp_flags smallint	lp_len smallint	t_xmin xid	t_xmax xid	t_field3 integer	t_ctid tid	t_infomask2 integer	t_infomask integer	t_hoff smallint	t_bits text	t_oid oid	t_data bytea
91	91	1648	1	68	578	0	0	(56069...	4	2306	24	[null]		[binary da
92	92	1576	1	68	578	0	0	(56069...	4	2306	24	[null]		[binary da
93	93	1504	1	68	578	0	0	(56069...	4	2306	24	[null]		[binary da
94	94	1432	1	72	578	0	0	(56069...	4	2306	24	[null]		[binary da
95	95	1360	1	68	578	0	0	(56069...	4	2306	24	[null]		[binary da
96	96	1288	1	68	578	0	0	(56069...	4	2306	24	[null]		[binary da
97	97	1216	1	72	578	0	0	(56069...	4	2306	24	[null]		[binary da
98	98	1144	1	68	578	0	0	(56069...	4	2306	24	[null]		[binary da
99	99	1072	1	68	578	0	0	(56069...	4	2306	24	[null]		[binary da
100	100	1000	1	72	578	0	0	(56069...	4	2306	24	[null]		[binary da
101	101	928	1	68	578	0	0	(56069...	4	2306	24	[null]		[binary da
102	102	856	1	68	578	0	0	(56069...	4	2306	24	[null]		[binary da
103	103	784	1	72	578	0	0	(56069...	4	2306	24	[null]		[binary da
104	104	712	1	68	578	0	0	(56069...	4	2306	24	[null]		[binary da
105	105	640	1	68	578	0	0	(56069...	4	2306	24	[null]		[binary da
106	106	568	1	72	578	0	0	(56069...	4	2306	24	[null]		[binary da
107	107	496	1	68	578	0	0	(56069...	4	2306	24	[null]		[binary da

Imagen 15. Análisis de los elementos del bloque intermedio.

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 SELECT * FROM heap_page_items(get_raw_page('mitabla', 112138));
```

Data Output | Explain | Messages | Notifications | Query History

	lp smallint	lp_off smallint	lp_flags smallint	lp_len smallint	t_xmin xid	t_xmax xid	t_field3 integer	t_ctid tid	t_infomask2 integer	t_infomask integer	t_hoff smallint	t_bits text	t_oid oid	t_data bytea
63	63	3656	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
64	64	3584	1	72	578	0	0	(11213...	4	2306	24	[null]		[binary da
65	65	3512	1	72	578	0	0	(11213...	4	2306	24	[null]		[binary da
66	66	3440	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
67	67	3368	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
68	68	3296	1	72	578	0	0	(11213...	4	2306	24	[null]		[binary da
69	69	3224	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
70	70	3152	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
71	71	3080	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
72	72	3008	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
73	73	2936	1	72	578	0	0	(11213...	4	2306	24	[null]		[binary da
74	74	2864	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
75	75	2792	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
76	76	2720	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
77	77	2648	1	72	578	0	0	(11213...	4	2306	24	[null]		[binary da
78	78	2576	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
79	79	2504	1	72	578	0	0	(11213...	4	2306	24	[null]		[binary da

Imagen 16. Análisis de los elementos del último bloque.

En este caso tenemos la información más detallada sobre cada tupla de nuestra tabla. Podemos observar que por cada bloque completo solo entran 107 tuplas como máximo en este caso.

Se contabilizan con el parámetro lp, que es distinto en la última página con respecto a la primera y la intermedia, al tener está espacio sobrante sin llegar a completar el bloque. También difieren entre estos sus lp_off correspondientes a su offset donde comienza la tupla y los lp_len que contienen el tamaño de la tupla, este se debe a que son de tamaño variable. Los valores ctid también difieren respecto los unos con los otros, la razón de esto es que los ctid almacenan lo posición en memoria física indicando número de bloque y fila por lo que tienen que ser distintas.

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?

Respuesta

El índice se crea con la herramienta que ofrece PgAdmin de tal forma que genera automáticamente la query para generar el índice (Veáse *Imagen 17*).

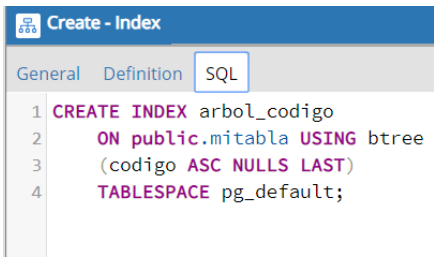


Imagen 17. Creacción del índice árbol.

Para saber donde se almacena físicamente este nuevo índice creado basta con saber su OID y buscarlo en nuestra carpeta de la base de datos correspondiendose a la ruta que se ve en la *Imagen 18*.

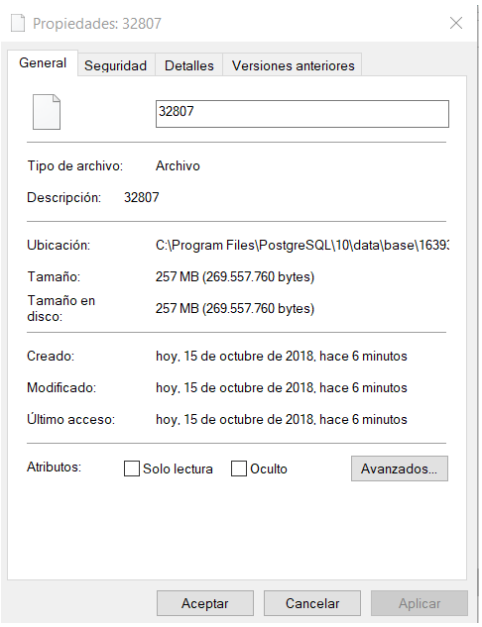


Imagen 18. Obtención de la ubicación y tamaño.

El tamaño como se puede ver en la anterior imagen es de 257 MB.

En cuanto a los bloques que ocupa lo podemos observar en las estadísticas de nuestro índice es de 32905 bloques como se puede ver en la *Imagen 19*. También se podrían haber obtenido con el módulo pgstattuple aplicando pgstatindex.

Statistics	Value
Index scans	0
Index tuples read	0
Index tuples fetched	0
Index blocks read	32905
Index blocks hit	0
Index size	257 MB
Version	2
Tree level	2
Index size-2	269557760
Root block no	290
Internal pages	117
Leaf pages	32787
Empty pages	0
Deleted pages	0
Average leaf density	90.09
Leaf fragmentation	0

Imagen 19. Estadísticas del índice.

Estos 32905 bloques se organizan en 3 niveles como indica el campo tree level al que se añade el nivel de la raíz.

Los bloques que tiene en cada nivel se corresponden a 1 como bloque raíz, a 117 bloques que serán los nodos intermedios y a 32787 que serán los nodos hoja, sumando todos estos bloques obtenemos 32905 bloques correspondientes al total.

Para saber las tuplas que tienen las hojas nos basta con dividir los bloques obtenidos con las tuplas. Además también podemos inspeccionar el bloque para verificarlo como se comprueba en la *Imagen 20*.

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 SELECT * FROM bt_page_items('arbol_codigo',1)
```

Data Output | Explain | Messages | Notifications | Query History

	itemoffset smallint	ctid tid	itemlen smallint	nulls boolean	vars boolean	data text
351	351	(62351,17)	16	false	false	5d 0...
352	352	(21184,53)	16	false	false	5e 0...
353	353	(100621,60)	16	false	false	5f 01...
354	354	(67969,90)	16	false	false	60 0...
355	355	(91745,65)	16	false	false	61 0...
356	356	(4694,92)	16	false	false	62 0...
357	357	(87680,28)	16	false	false	63 0...
358	358	(45268,68)	16	false	false	64 0...
359	359	(74964,50)	16	false	false	65 0...
360	360	(99005,56)	16	false	false	66 0...
361	361	(91739,25)	16	false	false	67 0...
362	362	(34,75)	16	false	false	68 0...
363	363	(95767,32)	16	false	false	69 0...
364	364	(99754,84)	16	false	false	6a 0...
365	365	(1804,71)	16	false	false	6b 0...
366	366	(99242,4)	16	false	false	6c 0...
367	367	(66769,95)	16	false	false	6d 0...

Imagen 20. Contenido de un bloque del índice.

$$\lceil \frac{12000000}{32787} \rceil = 367\, registros/bloque$$

Mientras que para el nivel intermedio nos bastará con dividir $\lfloor \frac{32787}{117} \rfloor = 280\, registros/bloque$

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.

Respuesta

Datos

$n_r = 12000000\, registros \quad B = 8192\, bytes$

Cálculo de longitud de puntero

$L_{registro} = L_{codigo} + L_{puntero} \quad 16 = 4 + L_{puntero} \quad L_{puntero} = 12\, bytes$

Cálculo de registros por nodo

$n_h \rightarrow n_h \cdot (L_{codigo} + L_{puntero}) + L_{puntero} \leq B \quad n_h \rightarrow n_h \cdot (4+12) + 12 \leq 8192$

$n_h \leq 511\, registros/nodo\ hoja$

$n \rightarrow n \cdot (L_{puntero}) + (n-1) \cdot L_{codigo} \leq B \quad n \rightarrow n \cdot (12) + (n-1) \cdot 4 \leq 8192$

$n \leq 512\, registros/nodo$

Cálculo de bloques de niveles

Nivel hoja $\lceil \frac{n_r}{n_h} \rceil = \lceil \frac{12000000}{511} \rceil = 23483\, bloques$

Nivel intermedio 1 $\lceil \frac{n_b}{n} \rceil = \lceil \frac{23483}{512} \rceil = 46\, bloques$

Al comparar los resultados vemos que teóricamente se requieren muchos menos bloques que en la práctica. Esto se puede deber a que PostgreSQL añada más espacio libre a los bloques y más información de control a los datos como se puede ver al tener un average leaf de 90.09 que indica que el 90% están ocupadas.

Cuestión 12. Crear un índice de tipo hash para el campo código y otro para el campo referencia.

Respuesta

Para crear los índices hemos seguido el mismo criterio que antes seleccionando en este caso los nuevos campos y su tipo hash . La Imagen 21 muestra la query que se ha usado, la otra sería similar cambiando simplemente el campo.

Create - Index

General Definition SQL

```
CREATE INDEX codigo_hash
    ON public.mitabla USING hash
    (codigo)
    TABLESPACE pg_default;
```

Imagen 21. Creacción índice hash sobre codigo.

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?

Respuesta

Al realizar un exhaustivo análisis con el módulo pageinspect y pgstattuple, específico para índices hash obtenemos los resultados de las imágenes que hay a continuación.

En este caso los datos que más relevantes nos son se corresponden con el número de live_items que como se ve en referencia se corresponde con 0 mientras que con código se corresponde con 206. Esto nos dirá los objetos que se encuentran cargados en memoria en nuestro caso habrá un índice que no tiene ninguno, esto se verá reflejado en las estadísticas de los bloques de items sin mostrar ninguno. También cambia la distribución del espacio libre como se muestra en los metadatos de los índices. Esto provoca que ocupe mucho más el índice hash sin que se emplee porque no aparece ningún objeto vivo. Esto se produce porque se han aplicado sobre un campo clave y sobre un campo con valores repetidos, por lo que da preferencia al primer índice sobre clave frente al otro.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT * FROM hash_page_stats(get_raw_page('referencia_hash',1))
```

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	0	0	8192	8148	40959	4294967295	0	2	65408

Imagen 22. Datos sobre los cajones Hash del campo referencia.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT * FROM hash_page_stats(get_raw_page('codigo_hash',1))
```

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	206	0	8192	4028	40959	4294967295	0	2	65408

Imagen 23 Datos sobre los cajones Hash del campo codigo.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT * FROM hash_page_items(get_raw_page('referencia_hash',1))
```

Data Output | Explain | Messages | Notifications | Query History

itemoffset integer	ctid tid	data bigint

Imagen 24. Tuplas del hash de referencia.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT * FROM hash_page_items(get_raw_page('codigo_hash',1))
```

Data Output | Explain | Messages | Notifications | Query History

	itemoffset integer	ctid tid	data bigint
190	190	(79056,44)	725312
191	191	(46096,34)	036864
192	192	(53164,56)	495616
193	193	(13605,65)	378560
194	194	(37976,49)	702464
195	195	(47585,3)	833536
196	196	(38084,44)	608448
197	197	(95985,105)	030976
198	198	(21356,65)	571584
199	199	(6865,69)	964800
200	200	(99476,89)	319488
201	201	(14967,33)	744320
202	202	(104722,52)	399680
203	203	(53614,16)	223168
204	204	(103837,83)	597120
205	205	(33871,23)	495360
206	206	(92312,40)	641088

Imagen 25. Tuplas del hash de campo código.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT magic, version, ntuples, ffactor, bsize, bmsize, bmshift,
     maxbucket, highmask, lowmask, ovflpoint, firstfree, nmaps, procid,
     regexp_replace(spares::text, '(,0)*}', '}') as spares,
     regexp_replace(mapp::text, '(,0)*}', '}') as mapp
      FROM hash_metapage_info(get_raw_page('referencia_hash', 0));
```

Data Output | Explain | Messages | Notifications | Query History

	magic bigint	version bigint	ntuples double precision	ffactor integer	bsize integer	bmsize integer	bmshift integer	maxbucket bigint	highmask bigint	lowmask bigint	ovflpoint bigint	firstfree bigint	nmaps bigint	procid oid
1	05121344	4	12000000	307	8152	4096	15	40959	65535	32767	34	29218	1	450

Imagen 26. Metadatos del índice hash sobre referencia.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT magic, version, ntuples, ffactor, bsize, bmsize, bmshift,
     maxbucket, highmask, lowmask, ovflpoint, firstfree, nmaps, procid,
     regexp_replace(spares::text, '(,0)*}', '}') as spares,
     regexp_replace(mapp::text, '(,0)*}', '}') as mapp
      FROM hash_metapage_info(get_raw_page('codigo_hash', 0));
```

Data Output | Explain | Messages | Notifications | Query History

	magic bigint	version bigint	ntuples double precision	ffactor integer	bsize integer	bmsize integer	bmshift integer	maxbucket bigint	highmask bigint	lowmask bigint	ovflpoint bigint	firstfree bigint	nmaps bigint	procid oid
1	05121344	4	12000000	307	8152	4096	15	40959	65535	32767	34	386	1	450

Imagen 27. Metadatos del índice hash sobre código.

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?

Respuesta

MiBaseDeDatos on postgres@PostgreSQL 10

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

Data Output | Explain | Messages | Notifications | Query History

```
UPDATE 1

Query returned successfully in 33 msec.
```

Imagen 28. Consulta del UPDATE.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT codigo, nombre, descripcion, referencia, CTID
    FROM public.mitabla
    WHERE codigo = 7000020;
```

Data Output | Explain | Messages | Notifications | Query History

	codigo integer	nombre text	descripcion text	referencia integer	ctid tid
1	7000020	producto...	descripcion70...	240	(112138,80)

Imagen 29. Posición en disco en la que se encuentra.

Lo que ha ocurrido con la tupla ha sido que PostgreSQL ha actualizado su valor pero la ha añadido al final del todo en vez de en su posición. Esto ocurre porque PostgreSQL para ser más eficaz y eficiente marca las tuplas como inactivas al cambiar un valor para reducir así costes de acceso y modificación de espacio libre o índices.

Para verificar que eso era lo que ocurría modificamos varias veces la tupla comprobando que siempre se situaba con el nuevo valor en el último bloque de disco.

Cuestión 15. Borrar las tuplas de la tabla **MiTabla** con código entre 7.000.000 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?

Respuesta

MiBaseDeDatos on postgres@PostgreSQL 10

```
DELETE FROM public.mitabla
    WHERE codigo > 7000000 and codigo<8000000;
```

Data Output | Explain | Messages | Notifications | Query History

```
DELETE 999999

Query returned successfully in 14 secs 284 msec.
```

Imagen 30. Borrado de las tuplas.

Lo que ocurre físicamente en la base de datos es que se marcan como tuplas muertas las tuplas de la query, sin borrarlas físicamente, por lo que aparecen casi un millón de tuplas marcadas como muertas como se muestra en la *Imagen 31*.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT * FROM pgstattuple('mitabla')
```

Data Output | Explain | Messages | Notifications | Query History

	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	918642688	11000001	755596028	82.25	1000001	68000068	7.4	4307156	0.47

Imagen 31. Muestra de las tuplas muertas tras borrado.

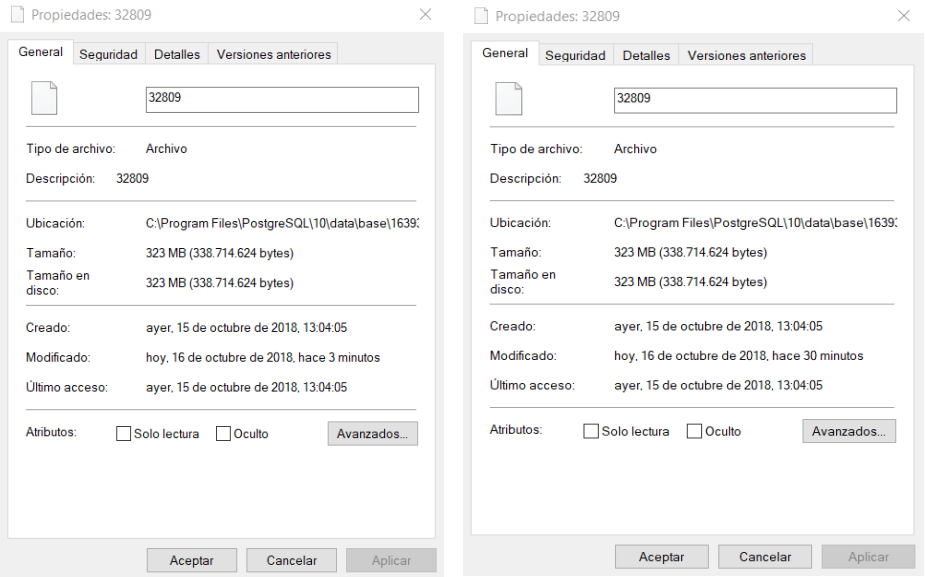


Imagen 32. Índices hash de código antes y después.

Index size	257 MB
Table size	876 MB

Index size	257 MB
Table size	876 MB

Imagen 33. Tamaño del índice árbol y tabla antes y después.

Viendo las Imagenes correspondientes al antes y el después se observa que el tamaño sigue siendo exactamente el mismo en todos los casos. Esto se debe a que PostgreSQL no las borra físicamente del disco sino que las marca para no leerlas en otras consultas. Esto se hace porque es muy costoso ir realizando borrados físicos además de los problemas que supondría como reordenar la tabla para poder utilizar ese espacio o tener que detener la base de datos para realizar esta tarea.

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?

Respuesta

MiBaseDeDatos on postgres@PostgreSQL 10

```
INSERT INTO public.mitabla(
    codigo, nombre, descripcion, referencia)
    VALUES (7500010, 'producto7500010', 'descripcion7500010', 187);
```

Data Output | Explain | Messages | Notifications | Query History

```
INSERT 0 1
```

Imagen 34. Insercción de tupla en la tabla.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT codigo, nombre, descripcion, referencia, CTID
    FROM public.mitabla
    WHERE codigo=7500010;
```

Data Output | Explain | Messages | Notifications | Query History

	codigo integer	nombre text	descripcion text	referencia integer	ctid tid
1	7500010	producto7500...	descripcion75...	187	(112138,83)

Imagen 35. Localización de dónde esta la tupla.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT * FROM heap_page_items(get_raw_page('mitabla', 112138));
```

Data Output | Explain | Messages | Notifications | Query History

	lp smallint	lp_off smallint	lp_flags smallint	lp_len smallint	t_xmin xid	t_xmax xid	t_field3 integer	t_ctid tid	t_infomask2 integer	t_infomask integer	t_hoff smallint	t_bits text	t_oid oid	t_data bytea
67	67	3368	1	68	578	597	0	(11213...	8196	1282	24	[null]		[binary da
68	68	3296	1	72	578	0	0	(11213...	4	2306	24	[null]		[binary da
69	69	3224	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
70	70	3152	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
71	71	3080	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
72	72	3008	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
73	73	2936	1	72	578	0	0	(11213...	4	2306	24	[null]		[binary da
74	74	2864	1	68	578	597	0	(11213...	8196	1282	24	[null]		[binary da
75	75	2792	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
76	76	2720	1	68	578	0	0	(11213...	4	2306	24	[null]		[binary da
77	77	2648	1	72	578	0	0	(11213...	4	2306	24	[null]		[binary da
78	78	2576	1	68	578	597	0	(11213...	8196	1282	24	[null]		[binary da
79	79	2504	1	72	578	0	0	(11213...	4	2306	24	[null]		[binary da
80	80	2432	1	68	594	595	0	(11213...	4	9474	24	[null]		[binary da
81	81	2360	1	68	595	596	0	(11213...	4	9474	24	[null]		[binary da
82	82	2288	1	68	596	597	0	(11213...	8196	9474	24	[null]		[binary da
83	83	2216	1	68	598	0	0	(11213...	4	2050	24	[null]		[binary da

Imagen 36. Comprobación de posición de la tupla.

Una vez insertada la tupla en la tabla analizamos todos los datos obtenidos de las *Imagen 35* e *Imagen 36* sabemos que la tupla se insertará en la última posición del último bloque , en este caso en el bloque 112138 fila 83.

Esto se debe a que PostgreSQL realiza las insercciones en el último bloque de memoria disponible y aunque se hayan borrado tuplas, este borrado no ha sido del espacio físico porque siguen ocupando espacio. Por eso se ha insertado en último bloque y posición.

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?

Respuesta

El problema de dejar tuplas muertas es que a la larga el sistema crece mucho de tamaño y pierde eficiencia. Por esto es necesario introducir mecanismos para solucionar esto, los que más se usan son VACUUM, CLUSTER y la variante VACUUM FULL. VACUUM permite desmarcar las tuplas muertas liberando memoria para reutilizar por la tabla además durante su uso se permiten operaciones como SELECT, DELETE y UPDATE. La variante VACUUM FULL ya no permite esta concurrencia de operaciones pero ofrece una mayor liberación de espacio, aunque es recomendable evitarla porque es lenta ya que realiza una copia de la primera tabla. Por último, CLUSTER permite una reordenación física de la tabla en base a un índice, es decir, cogera los valores del índice y los irá poniendo contiguamente tampoco permite las operaciones concurrentes ni borra las tuplas muertas. Otro problema que presenta CLUSTER es que tras realizarla si se añaden, eliminan o modifican tuplas estos cambios ya no presentarán el orden físico por lo que será necesario volver a aplicarlo. Otro tipo de optimización que se puede hacer es sobre el índice con REINDEX que reconstruye el índice sobre las datos nuevos.

En nuestro caso hemos aplicado VACUUM y REINDEX dado que al estudiar ventajas e incovenientes de estos métodos nos parecieron los mejores métodos. En este caso al aplicarlo ha borrado las tuplas muertas y las ha añadido a la memoria disponible. En el caso de REINDEX ha vuelto a crear el índice con los datos y sustituido el antiguo para así gestionar mejor esos posibles huecos que ha podido haber. En la Imagen 37 se puede observar el VACUUM y sus datos, en la Imagen 38 el REINDEX y en la imagen 39 el resultado de éstos.

MiBaseDeDatos on postgres@PostgreSQL 10

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

Data Output | Explain | Messages | Notifications | Query History

```
INFO:  haciendo vacuum a «public.mitabla»
INFO:  se recorrió el índice «MiTabla_pkey» para eliminar 1000002 versiones de filas
DETAIL:  CPU: user: 4.03 s, system: 0.31 s, elapsed: 6.29 s
INFO:  se recorrió el índice «arbol_codigo» para eliminar 1000002 versiones de filas
DETAIL:  CPU: user: 3.40 s, system: 0.18 s, elapsed: 3.76 s
INFO:  se recorrió el índice «codigo_hash» para eliminar 1000002 versiones de filas
DETAIL:  CPU: user: 4.76 s, system: 0.71 s, elapsed: 7.42 s
INFO:  se recorrió el índice «referencia_hash» para eliminar 1000002 versiones de filas
DETAIL:  CPU: user: 3.01 s, system: 1.18 s, elapsed: 7.31 s
INFO:  «mitabla»: se eliminaron 1000002 versiones de filas en 112125 páginas
DETAIL:  CPU: user: 1.10 s, system: 1.98 s, elapsed: 11.45 s
INFO:  el índice «MiTabla_pkey» ahora contiene 11000002 versiones de filas en 42752 páginas
DETAIL:  1000002 versiones de filas del índice fueron eliminadas.
3526 páginas de índice han sido eliminadas, 0 son reusables.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  el índice «arbol_codigo» ahora contiene 11000002 versiones de filas en 32905 páginas
DETAIL:  1000002 versiones de filas del índice fueron eliminadas.
2728 páginas de índice han sido eliminadas, 0 son reusables.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  el índice «codigo_hash» ahora contiene 11000002 versiones de filas en 41347 páginas
DETAIL:  1000002 versiones de filas del índice fueron eliminadas.
```

Imagen 37. Empleo de VACUUM y ANALYZE para ver el proceso.

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 REINDEX TABLE mitabla;
```

Data Output | Explain | Messages | Notifications | Query History

```
REINDEX

Query returned successfully in 1 min 46 secs.
```

Imagen 38. Empleo de REINDEX en la tabla.

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 SELECT * FROM pgstattuple('mitabla')
```

Data Output | Explain | Messages | Notifications | Query History

	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	918642688	11000002	755596096	82.25	0	0	0	76302716	8.31

Imagen 39. Muestra de la eliminación de las tuplas muertas.

En nuestro caso la tupla del ejercicio 14 no se encuentra en la tabla al haber sido eliminada con VACUUM porque se eliminó en el ejercicio 15, y la tupla del ejercicio 16 se encuentra en la misma posición esto se debe a que hemos aplicado VACUUM sin CLUSTER, en ese caso la tupla estaría en un bloque menor ya que se habría ordenado físicamente por un índice reorganizando el espacio, como previamente hemos explicado. Estos datos se ven en la *Imagen 40* y la *Imagen 41*.

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 SELECT codigo, nombre, descripcion, referencia, CTID
2     FROM public.mitabla
3     WHERE codigo=7000020;
```

Data Output | Explain | Messages | Notifications | Query History

codigo integer	nombre text	descripcion text	referencia integer	ctid tid

Imagen 40. Tupla del ejercicio 14.

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 SELECT codigo, nombre, descripcion, referencia, CTID
2     FROM public.mitabla
3     WHERE codigo=7500010;
```

Data Output | Explain | Messages | Notifications | Query History

	codigo integer	nombre text	descripcion text	referencia integer	ctid tid
1	7500010	producto7500...	descripcion75...	187	(112138,83)

Imagen 41. Tupla del ejercicio 16.

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.

Respuesta

Para crear la tabla con un factor de bloque de un tercio tenemos que modificar el valor de fill factor. Pondremos en el fill factor un valor de 33 para obtener así un tercio de factor de bloque, una vez hecho eso la tabla que se crea tendrá bloques que se rellenen únicamente un tercio del total de 8192 bytes. Por lo que ocupará en memoria tres veces más. En la *Imagen 42* podemos ver el proceso seguido y en la *Imagen 43* el resultado con su nuevo tamaño y porcentaje de espacio libre y ocupado.

Dashboard | Properties | SQL | Statistics | Deper

Heap blocks hit	685628
Index blocks read	3088587
Index blocks hit	32928066
Toast blocks read	0
Toast blocks hit	0
Toast index blocks read	0
Toast index blocks hit	0
Last vacuum	
Last autovacuum	
Last analyze	
Last autoanalyze	
Vacuum counter	0
Autovacuum counter	0
Analyze counter	0
Autoanalyze counter	0
Table size	2679 MB
Toast table size	8192 bytes
Indexes size	334 MB
Tuple count	12000000
Tuple length	785 MB
Tuple percent	29.32
Dead tuple count	0
Dead tuple length	0 bytes
Dead tuple percent	0
Free space	1800 MB
Free percent	67.22

Imagen 43. Estadísticas del tamaño de la tabla.

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?

Respuesta

MiBaseDeDatos on postgres@PostgreSQL 10

```sql
INSERT INTO public.mitabla2(
    codigo, nombre, descripcion, referencia)
    VALUES (33500010, 'producto3500010', 'descripcion13500010', 185);
```

Data Output | Explain | Messages | Notifications | Query History

```
INSERT 0 1

Query returned successfully in 46 msec.
```

Imagen 44. Insertar tupla a la tabla.

Lo primero que realizamos es insertar la tupla en nuestra nueva tabla. Una vez insertada la tupla solo nos queda comprobar la posición en la que se ha insertado que en nuestro caso se correponde con el último bloque de disco en la última fila, esto se ve en la *Imagen 45* y se comprueba en la *Imagen 46*. En este caso a pesar de tener espacio disponible en los bloques lo inserta en el último, porque PostgreSQL en las tablas con un fill factor pequeño solo permite INSERT hasta llenar el porcentaje de fill factor. El resto de espacio libre que queda se dejará para actualizar valores como veremos a continuación.

MiBaseDeDatos on postgres@PostgreSQL 10

```sql
SELECT codigo, nombre, descripcion, referencia, CTID
    FROM public.mitabla2
    WHERE codigo= 33500010;
```

Data Output | Explain | Messages | Notifications | Query History

	codigo integer	nombre text	descripcion text	referencia integer	ctid tid
1	33500010	producto3500...	descripcion13...	185	(342857,6)

Imagen 45. Obtener posición de la tupla.

MiBaseDeDatos on postgres@PostgreSQL 10

```sql
SELECT * FROM heap_page_items(get_raw_page('mitabla2',342857))
```

Data Output | Explain | Messages | Notifications | Query History

	lp smallint	lp_off smallint	lp_flags smallint	lp_len smallint	t_xmin xid	t_xmax xid	t_field3 integer	t_ctid tid	t_infomask2 integer	t_infomask integer	t_hoff smallint	t_bits text	t_oid oid	t_data bytea
1	1	8120	1	68	655	0	0	(34285...	4	2306	24	[null]		[binary d
2	2	8048	1	68	655	0	0	(34285...	4	2306	24	[null]		[binary d
3	3	7976	1	72	655	0	0	(34285...	4	2306	24	[null]		[binary d
4	4	7904	1	68	655	0	0	(34285...	4	2306	24	[null]		[binary d
5	5	7832	1	72	655	0	0	(34285...	4	2306	24	[null]		[binary d
6	6	7760	1	68	657	0	0	(34285...	4	2306	24	[null]		[binary d

Imagen 46. Verificación de la tupla en el último bloque.

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?

Respuesta

Antes de realizar la actualización de la tupla miramos el lugar que ocupa en la tabla porque en este caso la tupla ya estaba en la tabla, como se ve en la *Imagen 47*. Una

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT * FROM heap_page_items(get_raw_page('mitabla2',171777))
```

Data Output | Explain | Messages | Notifications | Query History

	lp smallint	lp_off smallint	lp_flags smallint	lp_len smallint	t_xmin xid	t_xmax xid	t_field3 integer	t_ctid tid	t_infomask2 integer	t_infomask integer	t_hoff smallint	t_bits text	t_oid oid
21	21	6752	1	68	655	0	0	(17177...	4	2306	24	[null]	
22	22	6680	1	68	655	0	0	(17177...	4	2306	24	[null]	
23	23	6608	1	68	655	0	0	(17177...	4	2306	24	[null]	
24	24	6536	1	68	655	0	0	(17177...	4	2306	24	[null]	
25	25	6464	1	68	655	0	0	(17177...	4	2306	24	[null]	
26	26	6392	1	68	655	0	0	(17177...	4	2306	24	[null]	
27	27	6320	1	68	655	0	0	(17177...	4	2306	24	[null]	
28	28	6248	1	68	655	0	0	(17177...	4	2306	24	[null]	
29	29	6176	1	72	655	0	0	(17177...	4	2306	24	[null]	
30	30	6104	1	68	655	0	0	(17177...	4	2306	24	[null]	
31	31	6032	1	68	655	0	0	(17177...	4	2306	24	[null]	
32	32	5960	1	68	655	0	0	(17177...	4	2306	24	[null]	
33	33	5888	1	68	655	0	0	(17177...	4	2306	24	[null]	
34	34	5816	1	72	655	0	0	(17177...	4	2306	24	[null]	
35	35	5744	1	68	655	0	0	(17177...	4	2306	24	[null]	

Imagen 47. Posición del bloque antes del UPDATE.

vez hecho esto realizamos la actualización de la tupla.

MiBaseDeDatos on postgres@PostgreSQL 10

```
UPDATE public.mitabla2
    SET referencia=350
    WHERE codigo=9000010;
```

Data Output | Explain | Messages | Notifications | Query History

```
UPDATE 1

Query returned successfully in 33 msec.
```

Imagen 48. Query de UPDATE.

Los resultados obtenidos lo tenemos al comparar el antiguo CTID con el nuevo obtenido tras actualizar la tupla. En este caso podemos ver en la *Imagen 49* como se ha colocado en el mismo bloque pero en la última fila disponible. Esto se debe a que cuando se le da un fill factor menor que el de defecto PostgreSQL emplea el hueco que se crea en el bloque para almacenar los UPDATE además de que es mucho más eficiente almacenarlo en su mismo bloque que en otro distinto.

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 SELECT codigo, nombre, descripcion, referencia, CTID
2     FROM public.mitabla2
3     WHERE codigo=9000010;
```

Data Output | Explain | Messages | Notifications | Query History

	codigo integer	nombre text	descripcion text	referencia integer	ctid tid
1	9000010	producto...	descripcion90...	350	(171777,36)

Imagen 49. Situación del bloque tras la actualización

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?

Respuesta

La conclusión que se puede obtener de todas las pruebas realizadas anteriormente es que PostgreSQL tiene un sistema que funciona por debajo muy eficentemente. Esto se debe a que cuando crea tablas, índices etc. lo hace añadiendo bastante información de control, como se podía observar con módulos como pageinspect y pgstattuple, que además se reflejaba en el tamaño. También podemos saber de esta estructura su funcionamiento con las operaciones de INSERT, UPDATE y DELETE, que se realizan dependiendo del factor de bloque y espacio libre en este caso con las insercciones respetando siempre el fill factor y con las actualizaciones ocupando el espacio disponible en el bloque. De estas operaciones también se puede obtener como trata a las tuplas muertas PostgreSQL, simplemente marcándolas hasta que viniera VACUUM y las eliminara, y otras funciones como CLUSTER o REINDEX.

En definitiva con este conocimiento sobre el manejo y funcionamiento de archivos y tuplas se puede mejorar la eficiencia de nuestra base de datos para ello aplicando fill factor si hay muchas actualizaciones, tareas de mantenimiento como VACUUM y CLUSTER y conocer cómo se almacenarán las tuplas y los tamaños que podrán tener para su posterior uso.

Monitorización de la actividad de la base de datos

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

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

Respuesta

PostgreSQL permite monitorizar la actividad de la base de datos sobre el disco de tres formas posibles. La primera es usando funciones de la tabla de funciones de objetos de base de datos, la segunda es usando oid2name que es el módulo que ofrece PostgreSQL y la última es inspeccionando manualmente los catálogos que hay de la bases de datos.

La información que se puede obtener es el número de bloques de cada elemento y también su nombre o OID. Al saber el tamaño de bloque esto nos permite calcular el tamaño de cada elemento.

Entre las estructuras de las que puede obtener datos están las tablas, índices, tablas TOAST, el tamaño de la base de datos y las columnas entre otros.

En definitiva PostgreSQL permite un seguimiento del tamaño de cada elemento de la base de datos y su correspondiente OID o nombre.

La información que puedes obtener es

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

Respuesta

Como se pedía un índice primario esto implicaba que tendría que estar ordenado físicamente por el campo seleccionado. Por lo que había que indicar que el índice fuera CLUSTERED. Haciendo esto tendríamos ordenado el índice por el campo referencia. Esto se ve en la query generada en la *Imagen 50*.

Únicamente puede haber índices primarios sobre un único campo porque solo es posible ordenar las tuplas sobre un campo, a excepción de que sean iguales o tengan una correspondencia proporcional.

Create - Index

General Definition SQL

```
1 CREATE INDEX referencia_btree
2     ON public.mitabla USING btree
3     (referencia ASC NULLS LAST)
4     TABLESPACE pg_default;
5
6 ALTER TABLE public.mitabla
7     CLUSTER ON referencia_btree;
```

Imagen 50. Creación del índice primario.

Cuestión 24. Crear un índice hash sobre el campo referencia.

Respuesta

```
-- DROP INDEX public.referencia_hash;

CREATE INDEX referencia_hash
    ON public.mitabla USING hash
    (referencia)
    TABLESPACE pg_default;
```

Imagen 51. Creación del índice hash sobre referencia.

Cuestión 25. Crear un índice sobre el campo código de tipo btree y otro de tipo hash sobre el mismo campo.

Respuesta

```
-- DROP INDEX public.codigo_btree;

CREATE INDEX codigo_btree
    ON public.mitabla USING btree
    (codigo)
    TABLESPACE pg_default;
```

Imagen 52.A Creación del índice btree sobre código.

```
-- Index: codigo_hash

-- DROP INDEX public.codigo_hash;

CREATE INDEX codigo_hash
    ON public.mitabla USING hash
    (codigo)
    TABLESPACE pg_default;
```

Imagen 52.B Creación del índice hash sobre código.

Cuestión 26. Analizar el tamaño de cada índice creado y compararlos entre sí. ¿Qué conclusiones se pueden extraer de dicho análisis?

Respuesta

Una vez creados todos los índices procedemos a obtener sus tamaños para compararlos en este caso los que menos ocupan son los btree porque requieren de almacenar únicamente los nodos. Mientras que los hash son mucho más grandes ya que tienen que almacenar más punteros como cajones. Además el hash sobre

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 SELECT relname, relpages
2 FROM pg_class
3 ORDER BY relpages DESC;
```

Data Output | Explain | Messages | Notifications

	relname name	relpages integer
1	mitabla	112139
2	referencia_hash	70179
3	mitabla_pkey	42752
4	codigo_hash	41347
5	codigo_btree	32905
6	referencia_btree	32905

Imagen 53. Tamaños de los índices.

referencia ocupa mucho más espacio en disco sin tuplas vivas, es decir, que no hay ninguna cargada en el sistema.

Por tanto los btree son índices que ocupan mucho menos espacio que los hash y más utilizados aunque estos últimos son más útiles en búsquedas de igualdad.

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:

```
SELECT pg_stat_reset();
```

Data Output | Explain | Messages | Query History

	pg_stat_reset void
1	

Imagen 54. Reinicialización de los datos.

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

Respuesta

Se pueden obtener estadísticas para la tabla, índice, actividades, usuarios, costes, tiempos, bloques, tuplas, OIDs y más.

MiBaseDeDatos on postgres@PostgreSQL 10

```
EXPLAIN ANALYZE SELECT codigo, nombre, descripcion, referencia
    FROM public.mitabla
    WHERE codigo=9001000;
```

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Index Scan using codigo_hash on mitabla (cost=0.00..8.02 rows=1 width=43) (actual time=1.781..1.783 rows=1 loops=1)
2	Index Cond: (codigo = 9001000)
3	Planning time: 47.718 ms
4	Execution time: 1.805 ms

Imagen 55. EXPLAIN de la query a realizar.

Para mostrar esta información simplemente hacemos la query con la condición de que sea 9001000 añadimos EXPLAIN para obtener más información de como realiza la consulta PostgreSQL internamente. En este caso usa el código_hash dado que para operaciones de igualdad un hash es más eficiente que un btree. Para obtener los bloques tenemos dos opciones la primera es obtenerlos del explain al que habría que descontar el valor inicial y el de CPU al coste y la otra opción es a través del Statistics Collector como se ve en la *Imagen 56*.

Dashboard | Properties | SQL | Statistics | Dependencies | Dependents | Que

No limit

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT blks_read,blks_hit,tup_returned
FROM pg_stat_database
WHERE datname='MiBaseDeDatos'
```

Data Output | Explain | Messages | Query History

	blks_read bigint	blks_hit bigint	tup_returned bigint
1	0	9	5

Imagen 56. Bloques leídos en la consulta.

El resultado que arroja es de 9 bloques menos 1 que se corresponde con el de la propia consulta se queda en 8 bloques que ya estaban en memoria por ser hit.

La razón de que sean estos pocos bloques se debe en que la mayoría del coste es de la búsqueda en el índice hash más el coste de recuperar el bloque.

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

Respuesta

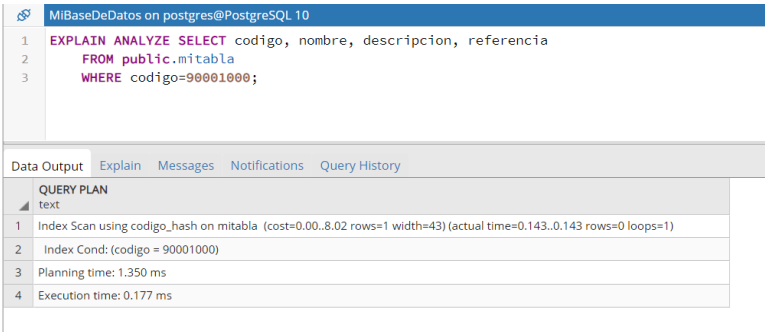


Imagen 57. EXPLAIN de la query.

Para realizar la query seguimos con EXPLAIN que nos muestra ahora que ha vuelto a usar el índice hash del campo código para buscar la tupla porque sigue siendo una igualdad; aunque ahora la tupla ya no existe por lo que los bloques a leer se reducen únicamente a los del índice. Como se ve en la *Imagen 57*.

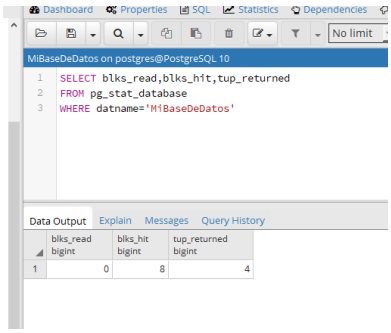


Imagen 58. Bloques y tuplas de la consulta.

El número de bloques que ha leído esta vez es de 7 bloques porque no hay tenido que recuperar el dato del bloque, es decir, que no habia coste de datos. Véase *Imagen 58*.

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

Respuesta

MiBaseDeDatos on postgres@PostgreSQL 10

```
EXPLAIN ANALYZE SELECT codigo, nombre, descripcion, referencia
    FROM public.mitabla
    WHERE codigo<2000;
```

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Bitmap Heap Scan on mitabla (cost=40.25..7342.62 rows=2041 width=43) (actual time=2.107..66.142 rows=2000 loops=1)
2	Recheck Cond: (codigo < 2000)
3	Heap Blocks: exact=1979
4	-> Bitmap Index Scan on codigo_btree (cost=0.00..39.74 rows=2041 width=0) (actual time=1.694..1.694 rows=2000 loops=1)
5	Index Cond: (codigo < 2000)
6	Planning time: 1.828 ms
7	Execution time: 66.342 ms

Imagen 59. EXPLAIN de la consulta.

Para obtener todas las tuplas con código menor que 2000 lanzamos la consulta que podemos ver en la Imagen 59. Como anteriormente, añadimos la cláusula EXPLAIN para obtener los detalles de monitorizacion. Cómo podemos ver, la base de datos utiliza el btree creado sobre el campo código, ya

Dashboard | Properties | SQL | Statistics | Dependencies | De

No limit

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT blks_read,blks_hit,tup_returned
FROM pg_stat_database
WHERE datname='MiBaseDeDatos'
```

Data Output | Explain | Messages | Query History

	blks_read bigint	blks_hit bigint	tup_returned bigint
1	1987	25	2009

Imagen 60. Bloques empleados en la consulta.

que es la opción más eficiente en este caso, es decir, la que menos accesos requiere.

Por otro lado, vemos que se han leído 1986 bloques de disco y 24 de memoria (se descuenta el de la query de las estadísticas). Aquí si tenemos coste de datos ya que tenemos que recuperar un gran número de tuplas. También podemos ver que la consulta ha devuelto 2009 tuplas.

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

Respuesta

Ahora deberemos tener en cuenta dos condiciones en nuestra consulta: tuplas con código mayor que 2000 y menor que 5000. Las tuplas deben cumplir las dos condiciones a la vez ya que es un AND. Para ello realizamos la consulta que podemos ver en la *Imagen 61*, de nuevo con el Explain y con el analyze para monitorizar la consulta. Así, podemos ver que la búsqueda se ha realizado sobre el btree del campo código, ya que es de nuevo la opción más eficiente porque los hash solo serán más rápidas en operaciones de igualdad frente a los btree.

MiBaseDeDatos on postgres@PostgreSQL 10

```
EXPLAIN ANALYZE SELECT codigo, nombre, descripcion, referencia
    FROM public.mitabla
    WHERE codigo>2000 AND codigo<5000;
```

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Bitmap Heap Scan on mitabla (cost=67.82..10710.21 rows=3062 width=43) (actual time=1.276..86.078 rows=2999 loops=1)
2	Recheck Cond: ((codigo > 2000) AND (codigo < 5000))
3	Heap Blocks: exact=2951
4	-> Bitmap Index Scan on codigo_btree (cost=0.00..67.06 rows=3062 width=0) (actual time=0.807..0.808 rows=2999 loops=1)
5	Index Cond: ((codigo > 2000) AND (codigo < 5000))
6	Planning time: 0.316 ms
7	Execution time: 86.331 ms

Imagen 61. EXPLAIN de la query.

Por otro lado, podemos ver que ha leído de disco 2913 bloques y de memoria 77. Ahora encontramos con un coste de datos mayor que el anterior ya que se están utilizando dos condiciones a satisfacer, lo que multiplica por dos nuestros registros a recuperar. Las tuplas devueltas han sido 3009.

Dashboard | Properties | SQL | Statistics | Depen

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT blks_read,blks_hit,tup_returned
FROM pg_stat_database
WHERE datname='MiBaseDeDatos'
```

Data Output | Explain | Messages | Query History

	blks_read bigint	blks_hit bigint	tup_returned bigint
1	2914	77	3009

Imagen 62. Bloques accedidos por la consulta.

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

   Respuesta

En esta ocasión debemos mostrar todas las tuplas cuyo campo código sea distinto de 25000. Para ello ejecutamos la consulta de la Imagen 63. A priori podemos ver que esta consulta devolverá un gran número de tuplas, ya que

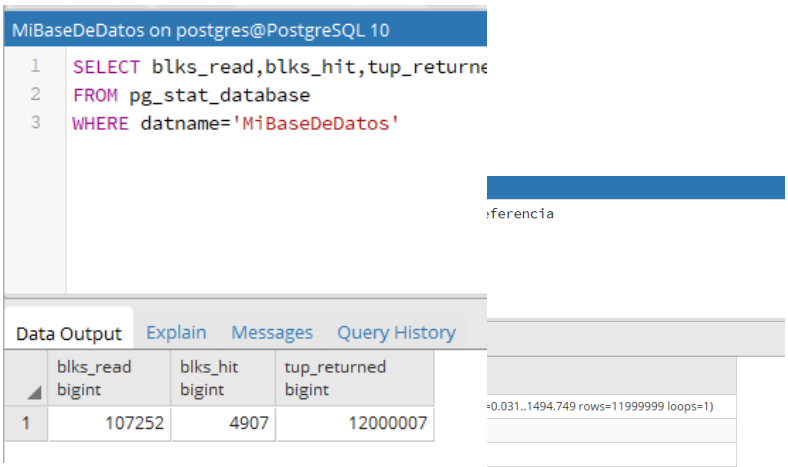


Imagen 64. Bloques de la query.

Imagen 63. EXPLAIN de la query.

solo excluye un valor de todos los posibles. Una vez ejecutamos la consulta podemos ver que así es, por ello, al ser un número tan grande de tuplas a devolver, nuestra base de datos utiliza una búsqueda secuencial sobre la tabla, la más eficiente en este caso porque es más sencillo recorrer todas secuencialmente y quitar la tupla que no se corresponde en vez de usar índices.

En cuanto a los bloques leídos de disco vemos que han sido 107252, un número muy grande ya que, como hemos reflejado anteriormente, se van a devolver un gran número de tuplas. En memoria, de igual manera, también nos encontramos con muchos bloques leídos, 4908en este caso. Las tuplas que devuelve esta consulta son 12000007.

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

Respuesta

Ahora pasamos a las consultas referentes al campo nombre. En el primer caso deberemos buscar las tuplas que tengan como nombre 'producto234567'. La consulta es muy simple, ya que solamente tenemos una condición, la podemos ver en la imagen adjunta. Como podemos ver, se ha hecho una búsqueda secuencial en paralelo sobre la tabla debido a que es la única forma de hacerlo, ya que no tenemos ningún índice sobre este campo.

MiBaseDeDatos on postgres@PostgreSQL 10

```
EXPLAIN ANALYZE SELECT codigo, nombre, descripcion, referencia
    FROM public.mitabla
    WHERE nombre= 'producto234567';
```

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Gather (cost=1000.00..175639.10 rows=1 width=43) (actual time=20.999..677.650 rows=1 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on mitabla (cost=0.00..174639.00 rows=1 width=43) (actual time=402.400..614.066 rows=0 loops=3)
5	Filter: (nombre = 'producto234567'::text)
6	Rows Removed by Filter: 4000000
7	Planning time: 0.220 ms
8	Execution time: 677.677 ms

Imagen 65. EXPLAIN de la query.

Por otro lado, vemos que ha leído de disco un total de 107225 bloques de disco, pueden parecer muchos bloques para una búsqueda tan concreta, pero tenemos que recordar que nombre no es un campo único, por lo que la base de datos tiene que leer un gran número de bloques para un resultado tan concreto. De la misma forma ocurre con los bloques leídos de memoria, un total de 5119 bloques. Sino una vez lo encontrará detendría la búsqueda.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT codigo, nombre, descripcion, referencia
FROM public.mitabla
WHERE nombre= 'producto234567';
```

Data Output | Explain | Messages | Query History

	codigo integer	nombre text	descripcion text	referencia integer
1	234567	producto...	descripcion23...	283

Imagen 66. Bloques accedidos por la query.

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

   Respuesta

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 EXPLAIN ANALYZE SELECT codigo, nombre, descripcion, referencia
2     FROM public.mitabla
3     WHERE referencia=350;
```

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Bitmap Heap Scan on mitabla (cost=448.78..58517.99 rows=23787 width=43) (actual time=15.045..363.481 rows=23899 loops=1)
2	Recheck Cond: (referencia = 350)
3	Heap Blocks: exact=21552
4	-> Bitmap Index Scan on referencia_btree (cost=0.00..442.84 rows=23787 width=0) (actual time=9.194..9.194 rows=23899 loops=1)
5	Index Cond: (referencia = 350)
6	Planning time: 0.104 ms
7	Execution time: 364.927 ms

Imagen 67. EXPLAIN de la query.

En este caso nos encontramos con las tuplas referentes al campo referencia, campo que tiene asociados dos índices: uno btree primario y otro hash. En el enunciado nos piden encontrar las tuplas cuyo valor para el campo referencia sea igual a 350. Para ello ejecutamos la consulta de la imagen. Como resultado obtenemos que, para realizar esta búsqueda, nuestra base de datos ha utilizado el índice del btree, ya que está ordenado por el campo referencia y la búsqueda será más eficiente en este caso.

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 SELECT blks_read,blks_hit,tup_returr
2 FROM pg_stat_database
3 WHERE datname='MiBaseDeDatos'
```

Data Output | Explain | Messages | Query History

	blks_read bigint	blks_hit bigint	tup_returned bigint
1	21432	195	23903

Imagen 68. Bloques que emplea la query.

8. Por otro lado, vemos que los bloques leídos de disco son 21432, muchos menos de los que nos podríamos esperar para una búsqueda tan concreta. Esto ocurre gracias a que el índice btree es primario respecto al campo referencia. De forma análoga ocurre con los bloques leídos de memoria, un total de 194 solamente.Mostrar la información de las tuplas con referencia<20.

Respuesta

Ahora tenemos que encontrar las tuplas cuyo valor referencia sea menor que 20. Como hemos visto en el apartado anterior y de igual forma, la base de datos utilizará en su búsqueda el btree creado sobre referencia, ya que es la forma más eficiente.

MiBaseDeDatos on postgres@PostgreSQL 10

```
EXPLAIN ANALYZE SELECT codigo, nombre, descripcion, referencia
    FROM public.mitabla
    WHERE referencia<20;
```

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Bitmap Heap Scan on mitabla (cost=9071.67..127267.49 rows=484546 width=43) (actual time=86.868..1431.386 rows=480129 loops=1)
2	Recheck Cond: (referencia < 20)
3	Rows Removed by Index Recheck: 6767009
4	Heap Blocks: exact=44577 lossy=66156
5	-> Bitmap Index Scan on referencia_btree (cost=0.00..8950.53 rows=484546 width=0) (actual time=79.411..79.411 rows=480129 loops=1)
6	Index Cond: (referencia < 20)
7	Planning time: 0.093 ms
8	Execution time: 1443.252 ms

Imagen 69. EXPLAIN de la query.

Podemos ver que se ha accedido a 112049 bloques de disco, ya que la búsqueda es más extensa que en el apartado anterior. Por otro lado, tenemos 4 accesos a memoria. Esto nos sirve para ver cómo el btree optimiza en gran parte los accesos a memoria.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT blks_read,blks_hit,tup_returned
FROM pg_stat_database
WHERE datname='MiBaseDeDatos'
```

Data Output | Explain | Messages | Query History

	blks_read bigint	blks_hit bigint	tup_returned bigint
1	112049	5	480133

Imagen 70. Bloques accedidos por la consulta.

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

Respuesta

MiBaseDeDatos on postgres@PostgreSQL 10

```
EXPLAIN ANALYZE SELECT codigo, nombre, descripcion, referencia
    FROM public.mitabla
    WHERE referencia>300;
```

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Seq Scan on mitabla (cost=0.00..262139.00 rows=4784067 width=43) (actual time=0.084..1657.490 rows=4776040 loops=1)
2	Filter: (referencia > 300)
3	Rows Removed by Filter: 7223960
4	Planning time: 0.100 ms
5	Execution time: 1756.547 ms

Imagen 71. EXPLAIN de la consulta.

En este caso nos piden encontrar las tuplas que cumplan la condición de que se campo referencia sea mayor que 300. Para ello, lanzamos la consulta de la *Imagen 71*. Como resultado obtenemos que la base de datos ha hecho una búsqueda secuencial, ya que es la opción más eficiente en este caso.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT blks_read,blks_hit,tup_returned
FROM pg_stat_database
WHERE datname='MiBaseDeDatos'
```

Data Output | Explain | Messages | Query History

	blks_read bigint	blks_hit bigint	tup_returned bigint
1	95856	16290	12000004

Imagen 72. Bloques accedidos por la query.

10. Por otro lado, podemos ver que se han leído de disco un total de 95856 bloques y de memoria 16290 bloques. Así, vemos el efecto de la búsqueda secuencial sobre el campo referencia: tenemos más bloques de memoria leídos que en apartados anteriores.Mostrar la información de las tuplas con codigo=70000 y referencia=200

Respuesta

En este caso hay que aplicar dos condiciones sobre dos campos que son codigo y referencia.

MiBaseDeDatos on postgres@PostgreSQL 10

```
EXPLAIN ANALYZE SELECT codigo, nombre, descripcion, referencia
    FROM public.mitabla
    WHERE codigo=70000 AND referencia=200;
```

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Index Scan using codigo_hash on mitabla (cost=0.00..8.02 rows=1 width=43) (actual time=1.591..1.591 rows=0 loops=1)
2	Index Cond: (codigo = 70000)
3	Filter: (referencia = 200)
4	Rows Removed by Filter: 1
5	Planning time: 0.119 ms
6	Execution time: 1.625 ms

Imagen 73. EXPLAIN de la query.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT blks_read,blks_hit,tup_returned
FROM pg_stat_database
WHERE datname='MiBaseDeDatos'
```

Data Output | Explain | Messages | Query History

	blks_read bigint	blks_hit bigint	tup_returned bigint
1	2	7	5

Imagen 74. Bloques accedidos por la query.

PostgreSQL optará por realizar una búsqueda usando el índice hash sobre el campo código esto se debe a que el campo código es único por lo que encontrando el elemento del campo código solo haría falta comprobar la referencia con la que viene siendo la búsqueda hash la más eficiente en este caso.El número de bloques que tiene es de 6 bloques en memoria y 2 bloques en disco que son los correspondientes al índice.

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

Respuesta

En este caso la query consiste en buscar las tuplas que tengan un código igual a 70000 o una referencia igual a 200. Aplicando el EXPLAIN PostgreSQL nos va a indicar la secuencia que ha realizado para obtener los datos de la consulta y el coste relacionado.

MiBaseDeDatos on postgres@PostgreSQL 10

```
EXPLAIN ANALYZE SELECT codigo, nombre, descripcion, referencia
    FROM public.mitabla
    WHERE codigo=70000 OR referencia=200;
```

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Bitmap Heap Scan on mitabla (cost=458.74..58587.43 rows=23788 width=43) (actual time=12.033..167.127 rows=23857 loops=1)
2	Recheck Cond: ((codigo = 70000) OR (referencia = 200))
3	Heap Blocks: exact=21501
4	-> BitmapOr (cost=458.74..458.74 rows=23788 width=0) (actual time=5.261..5.261 rows=0 loops=1)
5	-> Bitmap Index Scan on codigo_hash (cost=0.00..4.01 rows=1 width=0) (actual time=0.008..0.008 rows=1 loops=1)
6	Index Cond: (codigo = 70000)
7	-> Bitmap Index Scan on referencia_btree (cost=0.00..442.84 rows=23787 width=0) (actual time=5.249..5.250 rows=23856 loops=1)
8	Index Cond: (referencia = 200)
9	Planning time: 0.110 ms
10	Execution time: 168.469 ms

Imagen 75. EXPLAIN de la query.

PostgreSQL empleará un índice hash para la búsqueda del código y un índice btree para la búsqueda de la referencia. En este caso porque es más eficiente el hash para las búsquedas de igualdades y el btree porque el campo está ordenado.Los bloques que ha leído son 21568 de disco y 8 de memoria. Ha leído estos porque al tener estos dos índices PostgreSQL los ha aprovechado lo máximo posible para evitar el acceso a muchos bloques.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT blks_read,blks_hit,tup_returned
FROM pg_stat_database
WHERE datname='MiBaseDeDatos'
```

Data Output | Explain | Messages | Query History

	blks_read bigint	blks_hit bigint	tup_returned bigint
1	21568	9	23861

Imagen 76. Bloques empleados en la query.

Cuestión 28. Borrar los 4 índices creados y crear un índice multiclave btree sobre los campos referencia y producto.

```
-- Index: multiclave_referencia_producto

-- DROP INDEX public.multiclave_referencia_producto;

CREATE INDEX multiclave_referencia_producto
    ON public.mitabla USING btree
    (referencia, nombre COLLATE pg_catalog."default")
    TABLESPACE pg_default;

```

Imagen 77. Sentencia para generar el índice.

Dashboard | Properties | SQL | Statistics | Dependencies | Dependents | Edit Data - MiBas... | Edit Data - MiBas... | Sele

Statistics	Value
Index scans	0
Index tuples read	0
Index tuples fetched	0
Index blocks read	0
Index blocks hit	0
Index size	465 MB
Version	2
Tree level	3
Index size-2	487374848
Root block no	25447
Internal pages	379
Leaf pages	59114
Empty pages	0
Deleted pages	0
Average leaf density	90.13
Leaf fragmentation	0

Imagen 78. Estadísticas del índice.

Para realizar el índice multiclave añadimos más columnas a nuestro índice btree, no todos los índices soportan multiclave por lo que hay que tener cuidado al aplicarlo.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 conulta:

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

   Respuesta

   Ahora tenemos un índice multiclave por lo que dependerá de dos campos su uso, que son referencia y nombre. Su funcionamiento será similar a los índice de rejilla vistos en teoría que trabajaban sobre dos campos. Estos índices son muy eficaces para búsquedas de una tupla. Sin embargo cuando aumenta el número su rendimiento empeora.

   La información que podemos obtener al realizar esta consulta es que ha requerido de muy pocos accesos la obtención de la tupla como se refleja en los bloques de memoria accedidos. También ha empleado el índice multiclave porque empleaba ambos campos siendo muy eficiente.

MiBaseDeDatos on postgres@PostgreSQL 10

```
EXPLAIN ANALYZE SELECT codigo, nombre, descripcion, referencia
    FROM public.mitabla
    WHERE referencia = 200 AND nombre = 'producto6300031';
```

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Index Scan using multiclave_referencia_producto on mitabla (cost=0.56..8.58 rows=1 width=43) (actual time=0.093..0.094 rows=1 loops=1)
2	Index Cond: ((referencia = 200) AND (nombre = 'producto6300031'::text))
3	Planning time: 0.896 ms
4	Execution time: 0.109 ms

Imagen 79. EXPLAIN de la consulta sobre índice multiclave.

La resolución que ha empleado PostgreSQL ha sido el empleo del índice multiclave frente a la otra opción que tenía que era una búsqueda secuencial. En este caso el índice ha funcionado rápido al tener que obtener solo una tupla con las condiciones sobre los campos del índice.

Properties | SQL | Statistics | Dependencies

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT blks_read, blks_hit, tup_returned
FROM pg_stat_database
WHERE datname='MiBaseDeDatos'
```

Data Output | Explain | Messages | Query History

	blks_read bigint	blks_hit bigint	tup_returned bigint
1	0	12	5

Imagen 80. Bloques leídos en la query.

Ha leído un total de 11 bloques para obtener la tupla de entre todas. Estando estos bloques ya cargados en memoria. Véase la *Imagen 80*.

Han sido estos pocos porque como hemos explicado antes así es el funcionamiento de los índices multiclave.

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

Respuesta

La información que podemos obtener al realizar esta consulta es que ha requerido de muchos accesos a bloques para recuperar unas 20000 tuplas. También se ha empleado búsqueda secuencial en vez del índice.

En este caso el propio PostgreSQL ha optado por usar búsqueda secuencial en vez del índice que en caso de haberse usado hubiera disparado el coste de la búsqueda. Por lo que se ha recorrido por completo toda la tabla porque a pesar

de estar ordenado por referencia podría haber algún nombre repetido como se muestra en la *Imagen 90*.

MiBaseDeDatos on postgres@PostgreSQL 10

```
EXPLAIN ANALYZE SELECT codigo, nombre, descripcion, referencia
    FROM public.mitabla
    WHERE referencia = 200 OR nombre = 'producto6300031';
```

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Gather (cost=1000.00..190517.80 rows=23788 width=43) (actual time=2.448..749.036 rows=23856 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on mitabla (cost=0.00..187139.00 rows=9912 width=43) (actual time=0.758..705.921 rows=7952 loops=3)
5	Filter: ((referencia = 200) OR (nombre = 'producto6300031'::text))
6	Rows Removed by Filter: 3992048
7	Planning time: 0.070 ms
8	Execution time: 749.750 ms

Imagen 90. EXPLAIN de la consulta del multiclave.

El número de bloques que ha leído han sido 96411 de disco y 15917 en memoria que se corresponde con el total.

MiBaseDeDatos on postgres@PostgreSQL 10

```
SELECT blks_read, blks_hit, tup_returned
FROM pg_stat_database
WHERE datname='MiBaseDeDatos'
```

Data Output | Explain | Messages | Query History

	blks_read bigint	blks_hit bigint	tup_returned bigint
1	96411	15917	12000120

Imagen 91. Bloques leídos en la consulta.

Esto se debe a que ha realizado una lectura secuencial de la tabla porque a pesar de tener el campo referencia ordenado el campo nombre podría presentar duplicidades por lo que para ahorrar eligió la búsqueda secuencial leyendo así toda la tabla.

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

Respuesta

En este caso la información que podemos obtener al realizar esta consulta es que ha requerido de pocos accesos para encontrar las tuplas que lo cumplían. En cuanto al índice no se ha empleado sino que se ha buscado por la clave primaria que es código y usando de filtro el nombre .

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 EXPLAIN SELECT codigo, nombre, descripcion, referencia
2 FROM public.mitabla
3 WHERE codigo=6000 AND nombre='producto6300031';
```

Data Output | Explain | Messages | Query History

	QUERY PLAN text
1	Index Scan using mitabla_pkey on mitabla (cost=0.44..8.46 rows=1 width=43)
2	Index Cond: (codigo = 6000)
3	Filter: (nombre = 'producto6300031'::text)

Imagen 92. EXPLAIN de la consulta.

La secuencia de ejecución que ha realizado PostgreSQL ha sido comenzar con la búsqueda de la clave primaria código para continuar y aplicar el filtro del nombre para comprobarlo, sin usar así el índice.

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 SELECT blks_read, blks_hit, tup_returned
2 FROM pg_stat_database
3 WHERE datname='MiBaseDeDatos'
```

Data Output | Explain | Messages | Query History

	blks_read bigint	blks_hit bigint	tup_returned bigint
1	0	9	4

Imagen 93. Bloques accedidos durante la query.

Los bloques a los que ha tenido que acceder para realizar la consulta han sido 8 bloques(1 menos porque lo usa la otra query) que ya estaban en nuestra memoria. Aunque no devuelve ninguna tupla al no coincidir. Ha empleado estos bloques porque al hacer la búsqueda por clave primaria simplemente ha tenido que buscar el campo código adecuado para después aplicar el filtro del nombre mientras que el índice no hubiera funcionado para este caso ya que requiere de dos campos del que uno no estaba disponible.

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

Respuesta

La información obtenemos al realizarla es que va a devolver solo dos columnas aunque va a tener que recorrer secuencialmente toda la tabla porque nuestro campo nombre se puede repetir al no ser único.

En este caso el propio PostgreSQL ha optado por usar búsqueda secuencial en paralelo porque el índice multiclave no sirve al igual que en el caso anterior. Por ello ha recorrido toda la tabla obteniendo las dos filas como resultado.

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 EXPLAIN SELECT codigo, nombre, descripcion, referencia
2 FROM public.mitabla
3 WHERE codigo=6000 OR nombre='producto6300031';
```

Data Output | Explain | Messages | Query History

	QUERY PLAN text
1	Gather (cost=1000.00..188139.20 rows=2 width=43)
2	Workers Planned: 2
3	-> Parallel Seq Scan on mitabla (cost=0.00..187139.00 rows=1 width=43)
4	Filter: ((codigo = 6000) OR (nombre = 'producto6300031'::text))

Imagen 94. EXPLAIN de la query.

Los bloques que ha usado han sido 96183 bloques en disco más 16145 bloques ya en memoria, cuya suma da el total de bloques.

Esto se debe a que nuestro campo nombre no es único porque en ese caso una vez se encontrará la búsqueda cesaría. Sin embargo, aquí continua haciendo que tenga mayor costo.

MiBaseDeDatos on postgres@PostgreSQL 10

```
1 SELECT blks_read, blks_hit, tup_returned
2 FROM pg_stat_database
3 WHERE datname='MiBaseDeDatos'
```

Data Output | Explain | Messages | Query History

	blks_read bigint	blks_hit bigint	tup_returned bigint
1	96183	16145	12000120

Imagen 95. Bloques leídos por la query.

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.

El acceso que hace PostgreSQL depende de una gran cantidad de factores que su algoritmo trata. Este algoritmo le permite hacer la consulta de la forma más eficiente posible con los recursos que dispone como índeces, tablas y claves primarias. En nuestro caso es muy útil que PostgreSQL haga esto porque permite ahorrar muchos recursos que de no ser por su capacidad de elección se malgastarían. Además de que se nos permite ver con detalles como PostgreSQL realiza esto.

En resumidas cuentas, el acceso de PostgreSQL a los datos es el más eficiente de entre todos los posibles caminos que selecciona su algoritmo con gran eficacia. Como

se ha podido comprobar durante todas estas consultas y con todos estos tipos de índices.

Bibliografía

- Capítulo 1: Getting Started.
- Capítulo 5: 5.4 System Columns.
- Capítulo 11: Indexes.
- Capítulo 19: Server Configuration.
- Capítulo 24: Routine Database Maintenance Tasks.
- Capítulo 28: Monitoring Database Activity.
- Capítulo 29: Monitoring Disk Usage.
- Capítulo VI.II: PostgresSQL Client Applications.
- Capítulo VI.III: PostgresSQL Server Applications.
- Capítulo 50: System Catalogs.
- Capítulo 65: Database Physical Storage.
- Apéndice F: Additional Supplied Modules.
- Apéndice G: Additional Supplied Programs.