**Titulación: Grado en Ingeniería Informática y Sistemas de Información**

**Curso: 2019-2020. Convocatoria Ordinaria de Junio**

**Asignatura: Bases de Datos Avanzadas – Laboratorio**

**Practica 1: Arquitectura PostgreSQL y almacenamiento físico**

**ALUMNO 1:**

**Nombre y Apellidos: ISABEL MARTÍNEZ GÓMEZ**

**DNI: 06027983M**

**ALUMNO 2:**

**Nombre y Apellidos: JAVIER GARCÍA JIMÉNEZ**

**DNI: 09099503J**

**Fecha:** 3 de Marzo de 2020

**Profesor Responsable:** Iván González Diego

Mediante la entrega de este fichero los alumnos aseguran que cumplen con la normativa de autoría de trabajos de la Universidad de Alcalá, y declaran éste como un trabajo original y propio.

En caso de ser detectada copia, se calificará la asignatura como Suspensa – Cero.

**Plazos**

Trabajo de Laboratorio: Semana 27 Enero, 3 Febrero, 10 Febrero, 17 Febrero y 24 de Febrero.

Entrega de práctica: Día 3 de Marzo. Aula Virtual

Documento a entregar: Este mismo fichero con las respuestas a las cuestiones planteadas. Si se entrega en formato electrónico el fichero se deberá llamar: **DNIdelosAlumnos\_PECL1.doc**

**AMBOS ALUMNOS DEBEN ENTREGAR EL FICHERO EN LA PLATAFORMA.**

**Introducción**

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

**Almacenamiento Físico en PostgreSQL**

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

Todos los ficheros usados por PostgreSQL se encuentran en el directorio que

hayamos definido como directorio de datos en nuestro sistema (*data\_directory*).

Para buscar este directorio usaremos el siguiente comando:





Finalmente, nos devuelve el directorio donde están almacenados todos los ficheros en el disco duro. En este caso “C:/Program Files/PostgreSQL/12/data”.

Sin embargo, ahí estarán todos los ficheros relacionados con todas las bases de datos. Para ver en qué directorio está exactamente lo referente a nuestra base de datos, miraremos cuál es su identificador. Para ver el identificador de la base de datos creada usaremos el siguiente comando:

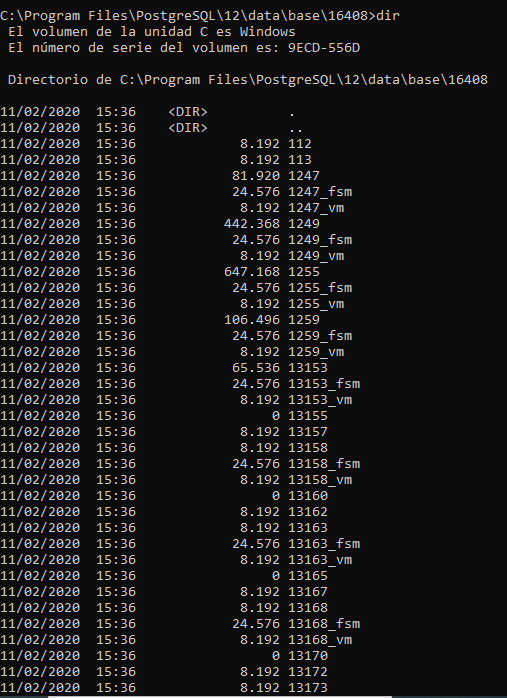




De esta forma vemos que a *‘MiBaseDatos’* se le ha asignado el identificador 16408.

Por tanto, los ficheros referentes a *‘MiBaseDatos’* se encuentran en *“C:/Program Files/PostgreSQL/12/data/base/16408”.*

Por otra parte, para ver los ficheros que se han creado al crear la base de datos nos dirigimos al *data\_directory* donde están todos los ficheros usados por PostgreSQL. Al introducir el comando *dir* estos son algunos de los ficheros que se han creado para inicializar la base de datos:



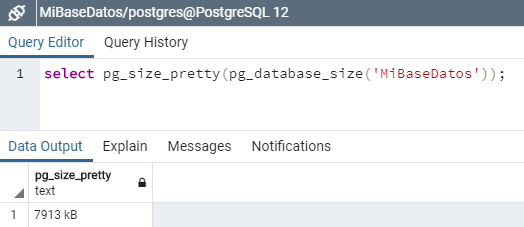
Por último, para saber el tamaño de la base de datos lo calcularemos mediante el

comando *pg\_database\_size* que nos devolverá el tamaño de la base de datos

indicada en bytes. Si introducimos el comando *pg\_size\_pretty* nos sacará el tamaño de

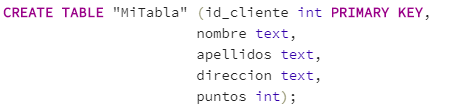
forma más visual.





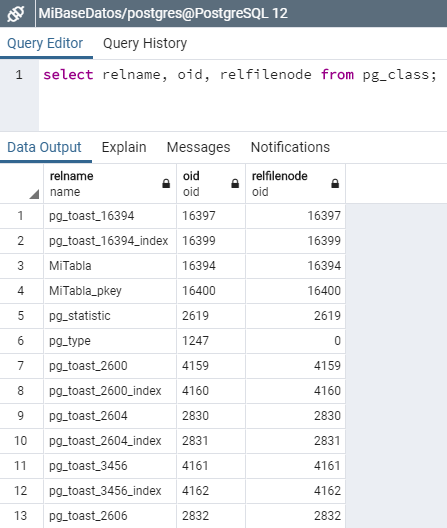
Como podemos observar, el tamaño inicial de ‘MiBaseDatos’ es de 7.913 kB, aunque no hayamos insertado nada aún. Esto es debido a que la base de datos no está vacía. Cuando se crea una base de datos, PostgreSQL inicializa la base de datos creando una serie ficheros iniciales como hemos visto anteriormente.

Cuestión 2. Crear una nueva tabla que se llame **MiTabla** que contenga un campo que se llame id\_cliente de tipo integer que sea la Primary Key, otro campo que se llame nombre de tipo text, otro que se llame apellidos de tipo text, otro dirección de tipo text y otro puntos que sea de tipo integer. ¿Qué ficheros se han creado en esta operación? ¿Qué guarda cada uno de ellos? ¿Cuánto ocupan? ¿Por qué?



Una vez creada la tabla, para poder ver los ficheros que se han creado ejecutaremos el siguiente comando:





De todo el resultado, en especial nos fijaremos en las cuatro primeras relaciones: *pg\_toast\_16934, pg\_toast\_16934\_index, MiTabla y MiTabla\_pkey*.

**pg\_toast\_16603:** PostgreSQL usa un mecanismo TOAST para evitar que las filas de datos físicos excedan el tamaño de un bloque de datos (generalmente 8KB). Con esto se consigue que la tabla tenga filas con más tamaño de lo permitido. Gracias al mecanismo TOAST cuando una fila excede el tamaño permitido (2 KB por defecto), el mecanismo TOAST intenta comprimir los valores y en caso de no poder, almacena los datos “fuera de línea” en una tabla TOAST única y asociada a dicha tabla. Los datos en la tabla de usuario se sustituyen por unos pequeños punteros para acceder a los datos de la fila almacenados en la tabla TOAST mientras que fuera de línea, en la tabla TOAST, se almacena un identificador (OID), un número de secuencia y un campo de datos para el fragmento. El identificador y el número de secuencia de la tabla TOAST proporciona una rápida recuperación de los valores.

En cuanto a tamaño, inicialmente ocupa 0 KB ya que aún no ha habido ningún registro en ‘MiTabla’ que exceda el tamaño de un bloque y haya tenido que ser almacenado “fuera de línea” en esta tabla.

***pg\_toast\_16603\_index:*** sirve para proporcionar índices a la tabla TOAST y encontrar los registros en dicha tabla de forma más eficiente que con una lectura secuencial. Inicialmente ocupa 8 KB.

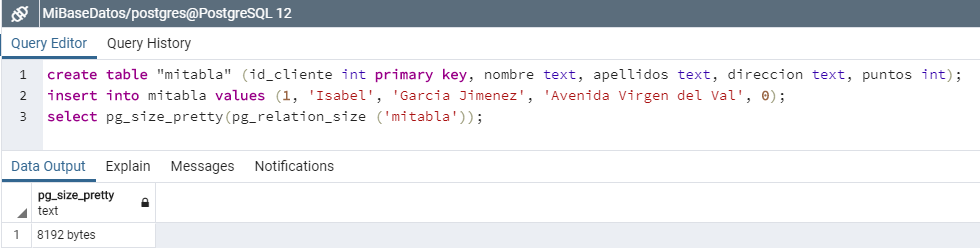
***MiTabla*:** tabla creada con los atributos *id\_cliente*, *nombre*, *apellidos*, *dirección* y *puntos*. Tiene identificador 16603. Inicialmente la tabla ocupa 0 KB, esto es debido a que aún no contiene ningún registro por lo que no ha hecho falta asignar bloques.

***MiTabla\_pkey*:** al crear una clave primaria en la tabla, automáticamente se crea un índice que referencia a la clave primaria de la tabla “MiTabla”. Este índice es “MiTabla\_pkey” e inicialmente ocupa 8 KB.

Cuestión 3. Insertar una tupla en la tabla. ¿Cuánto ocupa la tabla? ¿Se ha producido

alguna actualización más? ¿Por qué?





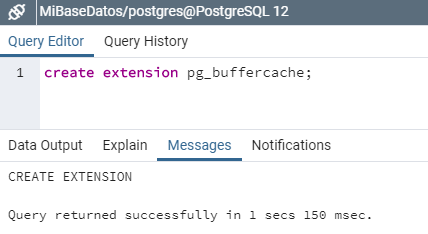
El tamaño que ocupa la tabla tras introducir la tupla es de 8 KB. Una tabla recién creada, ocupa en disco 0 bytes, ya que en su interior no contiene nada y no se ha tenido que crear ningún bloque de datos. Sin embargo, cuando se añade la primera tupla, PostgreSQL añade un bloque de datos para almacenar la tupla que ocupa 8 KB. Por tanto, el tamaño de la tabla ahora se actualiza a 8KB.

Cuestión 4. Aplicar el módulo pg\_buffercache a la base de datos **MiBaseDatos.** ¿Es

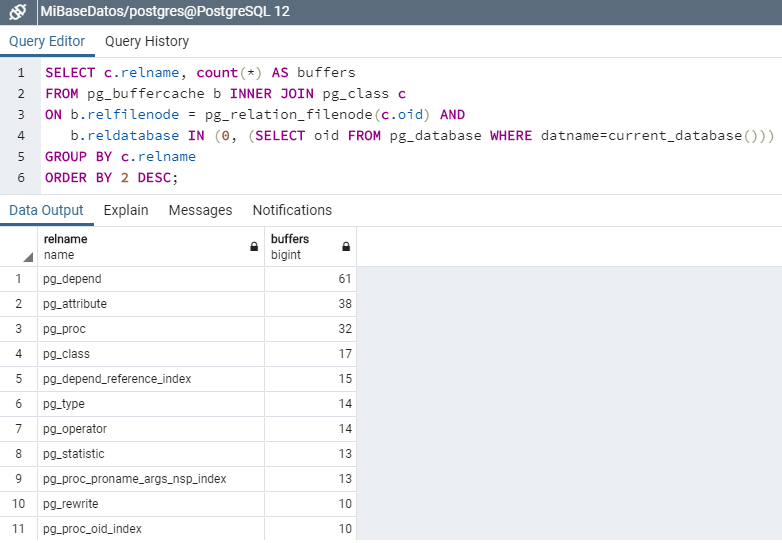
lógico lo que se muestra referido a la base de datos anterior? ¿Por qué?

El módulo *pg\_buffercache* proporciona un medio para examinar lo que está sucediendo en la memoria caché del búfer compartido en tiempo real.

Para instalar el módulo *pg\_buffercache* debemos ejecutar el siguiente comando:



Una vez que tenemos el módulo instalado, haremos la siguiente consulta:



Esta consulta está ordenada de forma que salen primero las relaciones con más bloques subidos del disco a memoria caché. Sin embargo, vamos a ver en qué posición está *‘MiTabla’.*



Vemos que el número de buffers dedicados en ‘*MiTabla’* la cual, hasta el momento, solamente dispone de una tupla, es solamente uno. Esto es obvio ya que PostgreSQL únicamente ha tenido que leer una vez la tabla y subirla a memoria caché para insertar la tupla. Al sólo existir una tupla en la tabla, no se ha necesitado más de un buffer dedicado en memoria caché para almacenar los datos de la misma.

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

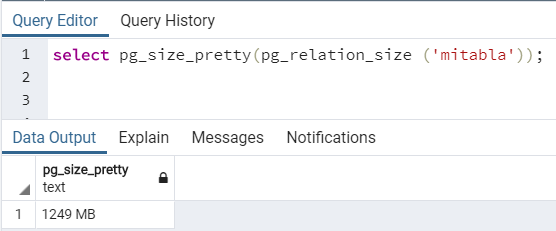
PostgreSQL? ¿Por qué?

Para borrar la tabla, usaremos el comando *drop table MiTabla* y para volver a crearla usaremos el comando *create table MiTabla (id\_cliente int primary key, nombre text, apellidos text, direccion text, puntos int);*

Para importar los datos a la tabla, nos iremos a Import/Export buscamos en el directorio donde esté el archivo .txt y marcamos el limitador como ‘;’.

El archivo original tiene un tamaño de 899,5 MB. Miremos cuánto ocupa ahora la tabla:





Una vez insertados los datos en la tabla, esta pasa a tener un tamaño de 1.249 MB, por lo que vemos que ha aumentado considerablemente el tamaño con respecto a lo que ocupaba el archivo txt.

Este cambio de tamaño se debe principalmente a que PostgreSQL asigna a la cabecera una información de control. Aparte, podríamos decir que también se debe a que existen huecos vacíos entre los diferentes bloques, ya que el número de registros que hay por bloque, no completa al 100% la capacidad del mismo.

Por estos motivos, el tamaño que ocupa la tabla es mayor que el tamaño del archivo original del que se han sacado los datos.

**Cálculo teórico**

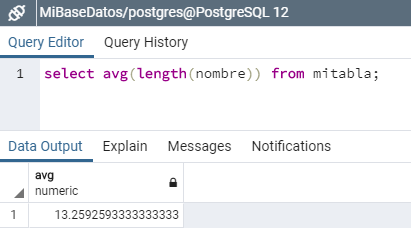
Vamos a calcular cuánto debería de ocupar la tabla en función de datos teóricos.

En la tabla tenemos los atributos:

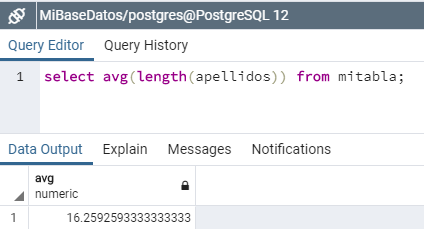
* **id\_cliente y puntos:** de tipo *int*
* **nombre, apellidos y dirección:** de tipo *text*

Un *int* en PostgreSQL supone 4 bytes y un campo de tipo *text* supone 1 byte por carácter. Para considerar la longitud de nombre, apellidos y dirección hemos hecho tres consultas para saber cuál es la media de la longitud de estos campos en la tabla. Por tanto, la media del campo nombre es 13 bytes, de apellidos es 16 y de dirección es 16.

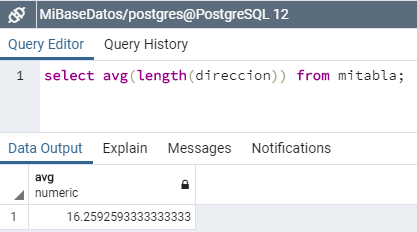












Longitud registro = Σ Lcampo = 4\*2 + 13 + 16\*2 = 53 bytes/registro

El factor de bloque viene dado por el tamaño de un bloque entre la longitud del registro. El tamaño de un bloque en PostgreSQL por defecto es de 8.192 bytes. Por tanto:

fr = B/Lr = 8.192/53 = 154.566 = 154 registros/bloque

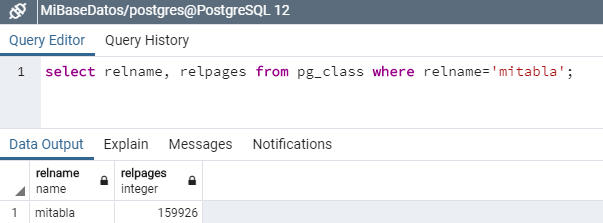
Por último, sacaremos el número de bloques en total que necesitamos si tenemos 15.000.000 de registros:

br = nr/fr = 15.000.000/154 = 97.402,597 bloques = **97.403 bloques en total**

**Tamaño real de bloques en MiTabla**

El tamaño real en bloques que nos proporciona PostgreSQL lo podremos ver con el siguiente comando:



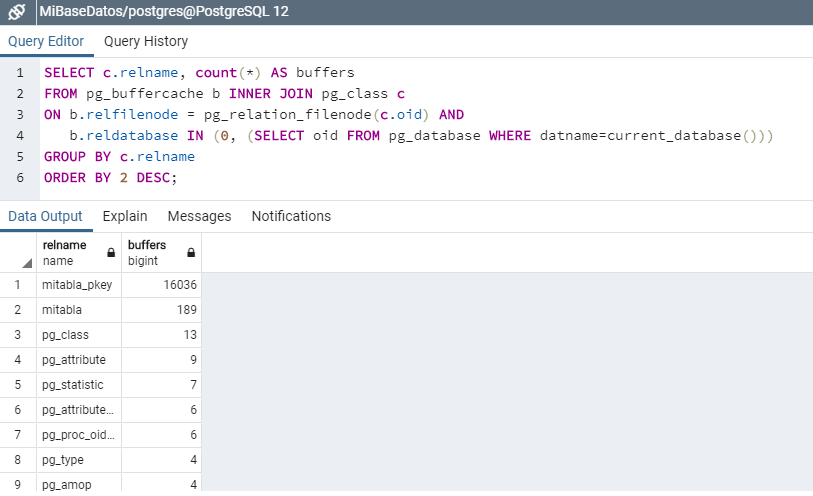


Por tanto, tenemos que el total de bloques real de *MiTabla* es de 159.926 bloques.

Calculando el tamaño en bloques de *MiTabla* en base a la teoría nos salían 97.403 bloques por lo que es una diferencia muy grande. Con 159.926 bloques en total, el factor de bloque sería de 94 registros/bloque. Esto es mucho menos que lo calculado anteriormente y es debido a que PostgreSQL asigna a cada bloque información de control.

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?

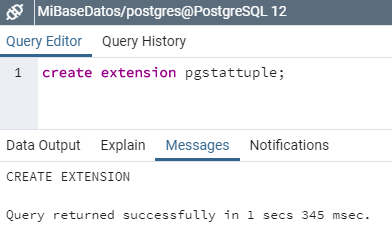
Tras aplicar de nuevo el módulo *pg\_buffercache* a *MiBaseDatos* nos sale lo siguiente:



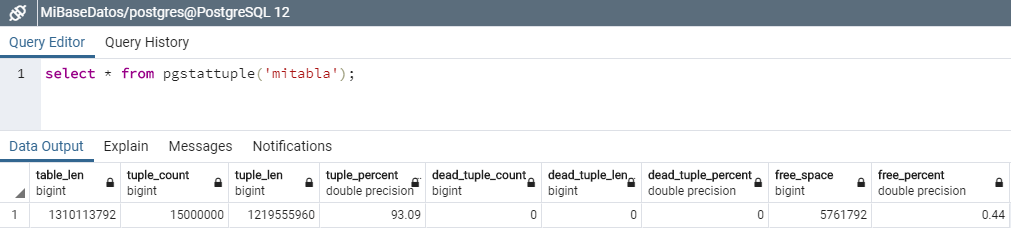
Vemos que *MiTabla* ha pasado de acceder únicamente una vez a la memoria caché del buffer a acceder 189 veces y *mitabla\_pkey* de acceder 2 a 16.036 veces. Al introducir una gran cantidad de datos en *MiTabla,* PostgreSQL necesita leer los datos de *MiTabla* y *MiTabla\_pkey* del disco para poder insertar las tuplas y por ello sube bloques de datos desde el disco a la memoria compartida del buffer caché.

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?

Para instalar el módulo pgstattuple debemos ejecutar el siguiente comando:



Realizamos la consulta para seleccionar todas las estadísticas relacionadas con las tuplas de *MiTabla*.



Tras la aplicación del módulo *psgtattuple* a *MiTabla*, obtenemos lo siguiente:

* *table\_len*: la tabla ocupa 1.249 MB.
* *tuple\_count*: existen 15.000.000 tuplas
* *tuple\_len*: la longitud total de las tuplas existentes es 1.163 MB.
* *tuple\_percent*: el porcentaje de tuplas existentes es de 93.09%
* *dead\_tuple\_count*: el número de tuplas “muertas” es 0.
* *dead\_tuple\_len*: la longitud de las tuplas “muertas” es 0
* *dead\_tuple\_percent*: el porcentaje de tuplas “muertas” es 0.
* *free\_space*: el espacio libre total es de 5.49 MB.
* *free\_percent*: el porcentaje de espacio libre es de 0.44%

El espacio que queda libre son 5.49 MB que es lo mismo que el 0.44%. Por tanto, el grado de ocupación de los bloques es de 100 - 0.44 = 99.56%

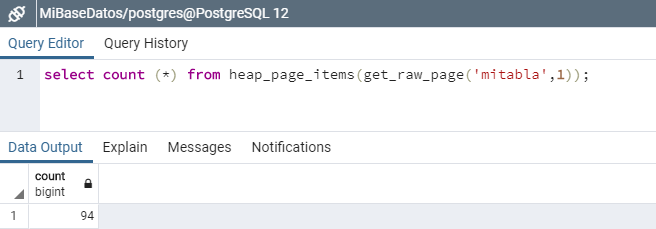
El motivo por el que hay espacios libres es porque los bloques de datos no se llenan al 100% de su capacidad. Esto se debe a que un registro no puede estar dividido en varios bloques, un registro obligatoriamente debe de estar completo en un bloque de datos. Por tanto, es inevitable tener algo de espacio libre en cada bloque.

Cuestión 8 ¿Cuál es el factor de bloque medio real de la tabla? Realizar una consulta SQL que obtenga ese valor y comparar con el factor de bloque teórico siguiendo el procedimiento visto en teoría.

El factor de bloque es el número de registros que caben en un bloque, por tanto, podremos sacar cuál es el factor de bloque con la siguiente consulta ya que todos los bloques, excepto el último, van a tener el máximo número de registros que pueden caber en un bloque.

Entonces, vamos a mirar en el primer bloque para saber el número de registros que hay y, por tanto, sacar el factor de bloque:





Hay 94 registros en el bloque, lo que quiere decir que tenemos un factor de bloque de 94 registros/bloque.

Anteriormente, con los cálculos teóricos obtuvimos un factor de bloque de 154 registros/bloque, sin embargo, vemos que el factor de bloque real es de 94. Esto es por lo que hemos dicho anteriormente, en los cálculos teóricos no hemos considerado la longitud de control ni que los bloques no están ocupados al 100% de su capacidad.

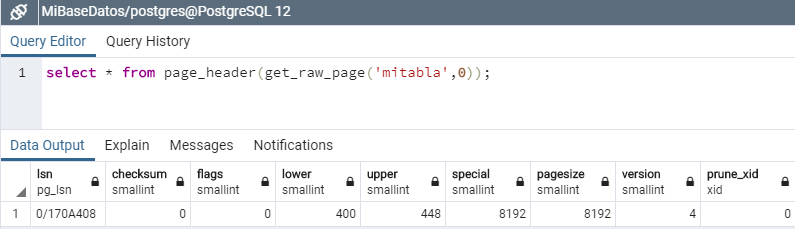
Cuestión 9 Con el módulo pageinspect, analizar la cabecera y elementos de la página del primer bloque, del bloque situado en la mitad del archivo y el último bloque de la tabla **MiTabla**. ¿Qué diferencias se aprecian entre ellos? ¿Por qué?

Para este ejercicio será necesario incluir el módulo *pageinspect*



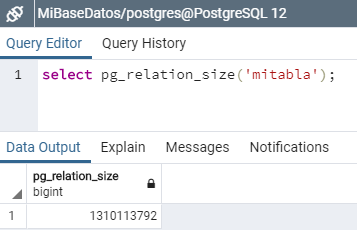
**Primer bloque**

El primer bloque es el bloque 0. Como podemos ver, nos salen varios campos de información de cabecera donde:

* El campo *lower* marca un valor de 400, éste especifica el primer espacio libre donde podemos insertar un dato.
* El campo *upper* marca un valor de 448 que especifica el final del espacio libre.

Para saber cuál es el bloque situado en la mitad del archivo y el último, necesitaremos saber el tamaño de la tabla.



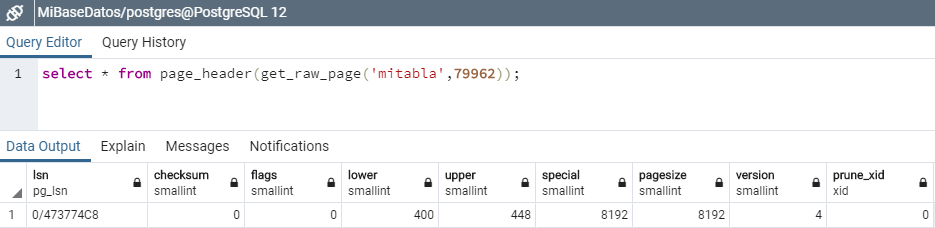


El tamaño de la tabla es de 1.310.113.792 bytes y el tamaño de bloque e de 8.192 bytes. Por tanto, la tabla ocupa 1.310.113.792/8.192 = 159.926 bloques.

**Bloque en la mitad del archivo**

Como hay 159.926 bloques en total en la tabla, el bloque que está en la mitad del archivo será el 79.963.

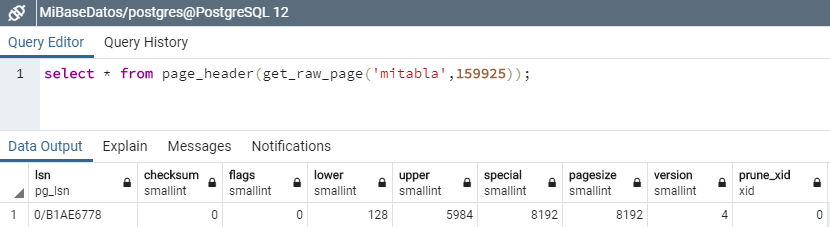
Hay una diferencia de 48 entre el lower y el upper, esto se refiere al espacio libre que hay en este bloque.



**Último bloque del archivo**

El último bloque de la tabla, por tanto, será el bloque 159.925

En este bloque en comparación con los demás, vemos que hay mucha diferencia entre los campos *lower* y *upper*, esto significa que hay mucho espacio libre que es debido a que como es el último bloque no está lleno.



Por último, podemos observar que para todos los bloques coinciden el tamaño de bloque de 8 KB y la versión 4.

Cuestión 10. Crear un índice de tipo árbol para el campo puntos. ¿Dónde se almacena físicamente ese índice? ¿Qué tamaño tiene? ¿Cuántos bloques tiene? ¿Cuántos niveles tiene? ¿Cuántos bloques tiene por nivel? ¿Cuántas tuplas tiene un bloque de cada nivel?

Para crear un índice de tipo árbol para el campo puntos, crearemos el índice con el comando *create index* al cual le llamamos *indice\_arbol\_puntos* y elegimos el campo sobre el cual queremos hacer el índice, en nuestro caso sobre el atributo *puntos*.



Como ya vimos en las primeras preguntas, los ficheros que usa PostgreSQL se guardan en el data\_directory definido. En nuestro caso se almacenará en “C:\ProgramFiles\PostgreSQL\12\data\base\16408”.

Como podemos observar al índice creado se le ha asignado el identificador 16469.





Con la siguiente consulta sabremos el tamaño del índice, que es 321.78 MB. Por tanto, al ocupar 321.78 MB, hay 337.412.096/8192 = 41.188 bloques.

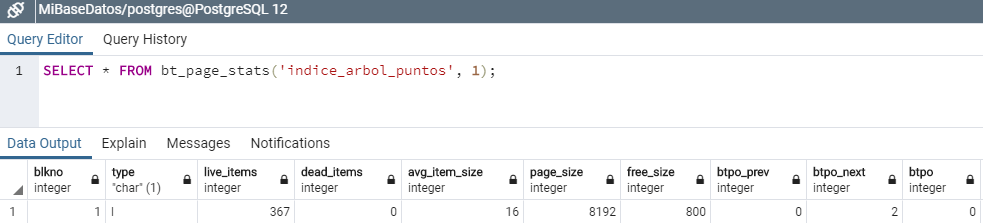


La raíz del árbol B está en el nivel 2, considerando un nivel de hojas, un nivel de nodos intermedios y el nivel de la raíz.

Los bloques que tiene el nivel de las hojas son 40.984 bloques, el nivel intermedio 1 tiene 203 bloques y la raíz tiene 1 bloque. En total, el índice ocupa 41.188 bloques como lo calculado anteriormente.

Para saber el número de tuplas que tiene un bloque en cada nivel, usaremos el módulo *bt\_page\_stats*.

**En el nivel de las hojas**, hay 367 registros en un nodo hoja y cada registro ocupa 16 bytes de media. Vemos en el campo *type* que sale una l, esto significa *leaf.*

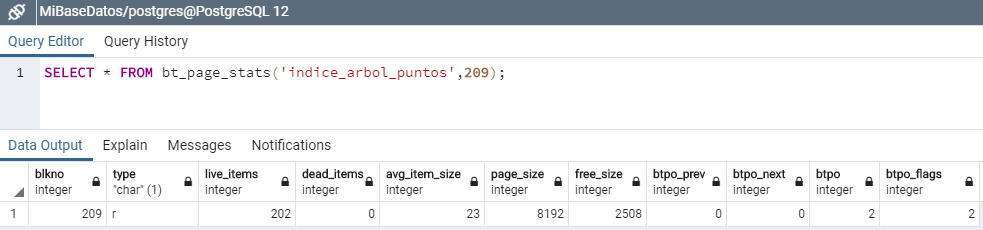


**En el nivel de los nodos intermedios,** hay 204 registros en un nodo intermedio. Cada registro ocupa una media de 23 bytes. Vemos que en el campo *type* que sale una i, esto significa *intermediate.*



**En el nivel raíz** hay 202 registros que ocupan de media cada uno 23 bytes.

Vemos que ahora en el campo *type* sale una r, esto significa *root.*



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.

Para determinar el tamaño en bloques que ocupa un índice de árbol B, realizaremos lo siguiente:

* Es un **índice secundario con campo no clave**, por tanto, posee **cajones de punteros**.
* El número de registros a indexar para este índice es el número de valores diferentes que tenga el campo puntos. **NRI = V(Puntos) = 700**
* Como ya sabemos, la longitud del registro índice en la hoja del árbol B es 16 bytes. Estos 16 bytes son la suma del campo puntos más la longitud del puntero a bloque. Por tanto, para saber cuánto ocupa el tamaño de un puntero a bloque, restamos el tamaño del campo. **LPB = 16 - 4 = 12 bytes.**

Tras esto ya podemos calcular cuántos punteros habrá en un nodo hoja. Como se ha comentado anteriormente, al ser un índice secundario con campo no clave, posee cajones de punteros, por tanto, los nodos hoja tienen punteros a bloque en vez de punteros a registro.

**Nodos hoja:**

nH\*(LPB+LPUNTOS) + LPB < = B

nH \* (12 + 4) + 12 < = 8.192

nH \* 16 < = 8.192 - 12

16\* NH < = 8.180; **NH = 511 punteros a bloque o valores de campo.**

**Nodos intermedios/raíz:**

n \* (LPB + LPUNTOS) + (n-1) \* LPUNTOS < = B

n \* 16 + (n-1) \* 4 < = 8.192

n \* 16 + n \* 4 – 4 < = 8.192

n \* 20 < = 8.196; **n = 409 punteros a bloque y 408 valores de campo**

Calculamos los bloques:

NH = 700/511 = 2 bloques

Raíz = 2/409 = 1 bloque

**Cajones de punteros**

Aún no hemos acabado, como bien hemos dicho anteriormente, estamos haciendo un índice secundario sobre campo no clave y hay cajones de punteros. Vamos a calcular el número de bloques que ocupan estos:

LRC = LPR, suponiendo que la longitud del puntero a registro sea igual a la longitud del puntero a bloque:

FRC = B/LRC = 8.192/12 = 682 punteros a registro/bloque

En cada cajón considerando equiprobabilidad habrá:

Si hay un total de 15.000.000 registros y hay 700 valores distintos del campo puntos:

En cada cajón hay 15.000.000/700 = 21.428 punteros a registro/cajón

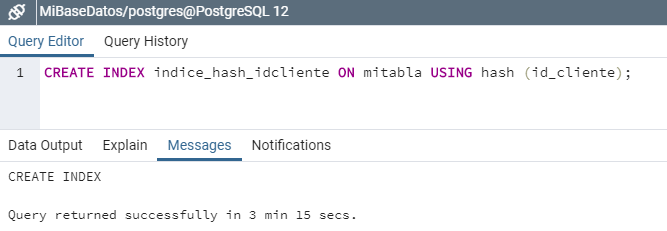
BRC = 21.428/1365 = 16 bloques/cajón

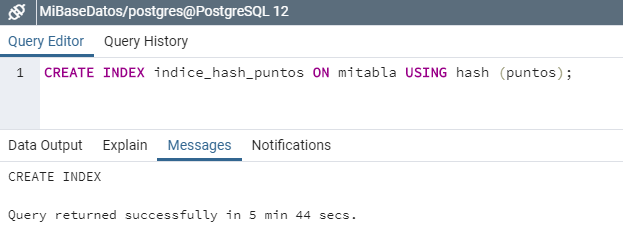
Número bloques totales cajones de punteros = 700\*16 = **11.200 bloques**

**El número total de bloques teórico del árbol B+ es: 1+2+11.200 = 11.203 bloques**

Hay una gran diferencia entre el número de bloques real y el número de bloques calculado teóricamente, una de las principales causas es que PostgreSQL trabaja con árbol B y lo hemos calculado según un árbol B+. También hemos llegado a la conclusión de que PostgreSQL no trabaja con cajones de punteros si no que directamente indexa sobre los 15.000.000 de registros. Por otra parte, el número de bloques calculados es mucho menor ya que no hemos considerado cabeceras de información de control y se ha supuesto que el tamaño del puntero a registro es igual al tamaño del puntero a bloque al ser imposible calcularlo.

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

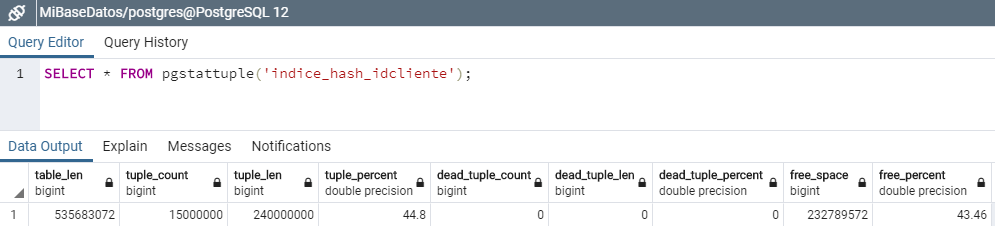




Cuestión 13. A la vista de los resultados obtenidos de aplicar los módulos pgstattuple y pageinspect, ¿Qué conclusiones se puede obtener de los dos índices hash que se han creado? ¿Por qué?

**PGSTATTUPLE**

**indice\_hash\_cliente**



**Índice\_hash\_puntos**



Como se puede observar, el índice realizado sobre el campo *id\_cliente* ocupa mucho menos espacio que el realizado sobre el campo *puntos*. Esto se debe claramente a la categoría que tienen ambos campos.

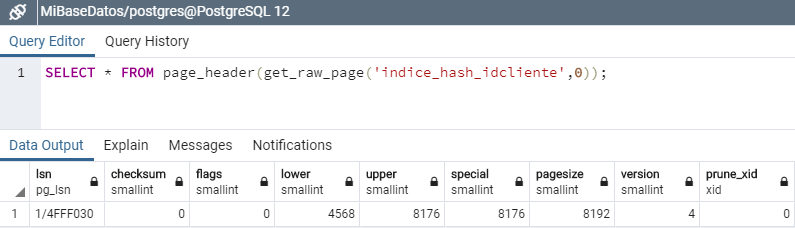
El *id\_cliente* es una *primary key*, lo que significa que no necesita cajones de punteros para ser indexado. Sin embargo, el campo puntos al no ser un campo clave y al no estar ordenado el archivo según este campo, necesita cajones de punteros para ser indexado, por tanto, ocupa mucho más espacio.

**PAGEINSPECT**

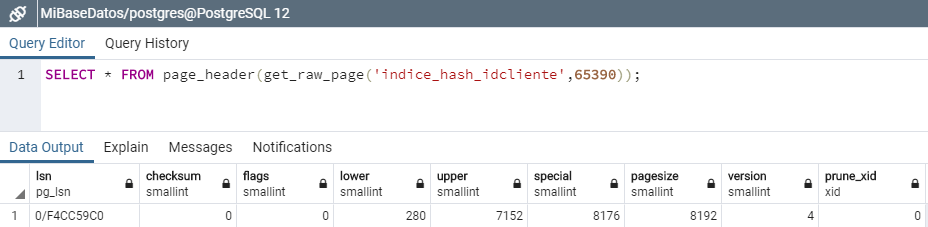
**Indice\_hash\_idcliente**

Como la longitud del índice es 535.683.072 bytes, habrá 535.683.072/8.192 bloques = 65.391 bloques.

**Primer paquete**

****

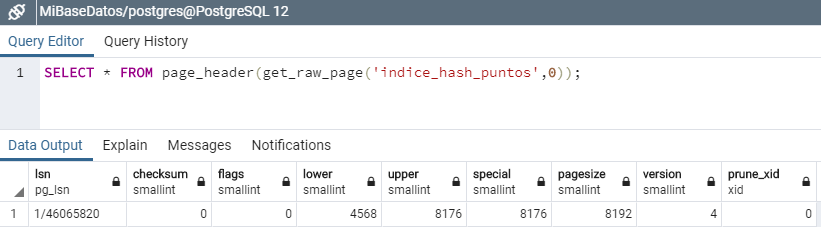
**Último bloque**

****

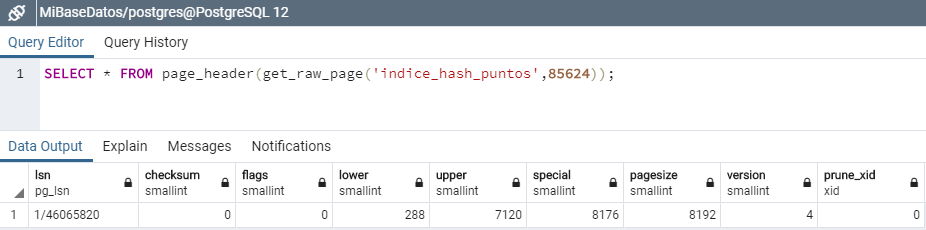
**Indice\_hash\_puntos**

Como el tamaño de *indice\_hash\_puntos* es 701.440.000, el número total de bloques es 701.440.000/8192 = 85.625 bloques

**Primer bloque**

****

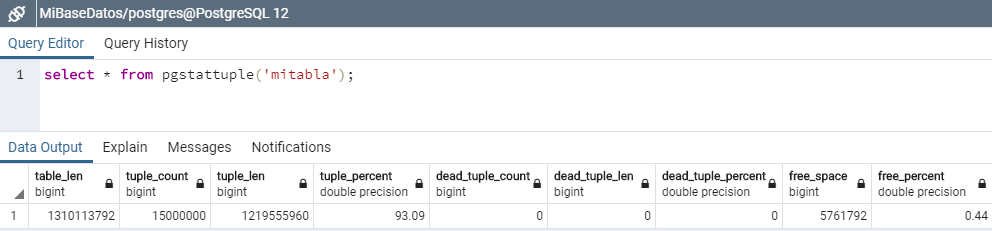
**Último bloque**

****

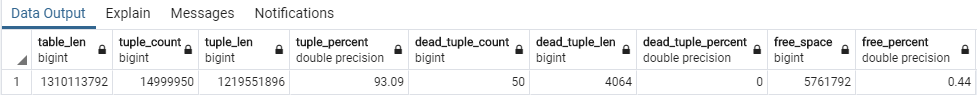
Como se puede observar a través del apartado *lower* en el primero y en el último bloque de nuestro índice, el primer bloque contiene más registros, porque su diferencia entre *lower* y *upper* (que se corresponde con el espacio libre dentro del bloque) es más pequeña.

Cuestión 14. Realice las pruebas que considere de inserción, modificación y borrado para determinar el manejo que realiza PostgreSQL internamente con los registros de datos y las estructuras de los archivos que utiliza. Comentar las conclusiones obtenidas.

Antes de realizar cualquier operación, las estadísticas sobre la tabla indican lo siguiente:

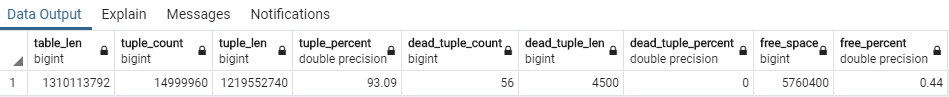


Veamos qué pasa si **borramos 50 tuplas** de la tabla de forma aleatoria. Una vez borradas, volvemos a mirar las estadísticas:



Tras ver los resultados, vemos que se ha modificado el campo dead\_tuple\_count de 0 a 50, esto es porque hemos eliminado 50 tuplas de forma aleatoria de la tabla. Sin embargo, el campo *table\_len* no se ve afectado tras borrar estas 50 tuplas, esto es debido a que PostgreSQL no elimina las tuplas si no que se marca esas tuplas como “muertas”.

Vamos a **insertar ahora 6 tuplas que ya existen** **y** **10 que no existen** para ver qué pasa:



Como vemos, en *tuple\_count* se insertan 10 tuplas nuevas que son las que hemos insertado que no existen. Sin embargo, incrementa en 6 el campo *dead\_tuple\_count,* esto es porque hemos añadido 6 tuplas que ya existen en la base de datos.

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



Cuando borramos datos en PostgreSQL, no se modifica el tamaño de la tabla, aunque se hayan borrado los datos, ¿por qué? Como hemos explicado anteriormente, esto es por el modo de borrar datos que tiene PostgreSQL. Cuando borramos tuplas de una tabla, el tamaño de la tabla no se modifica a no ser que apliquemos un VACUUM para reutilizar ese espacio.

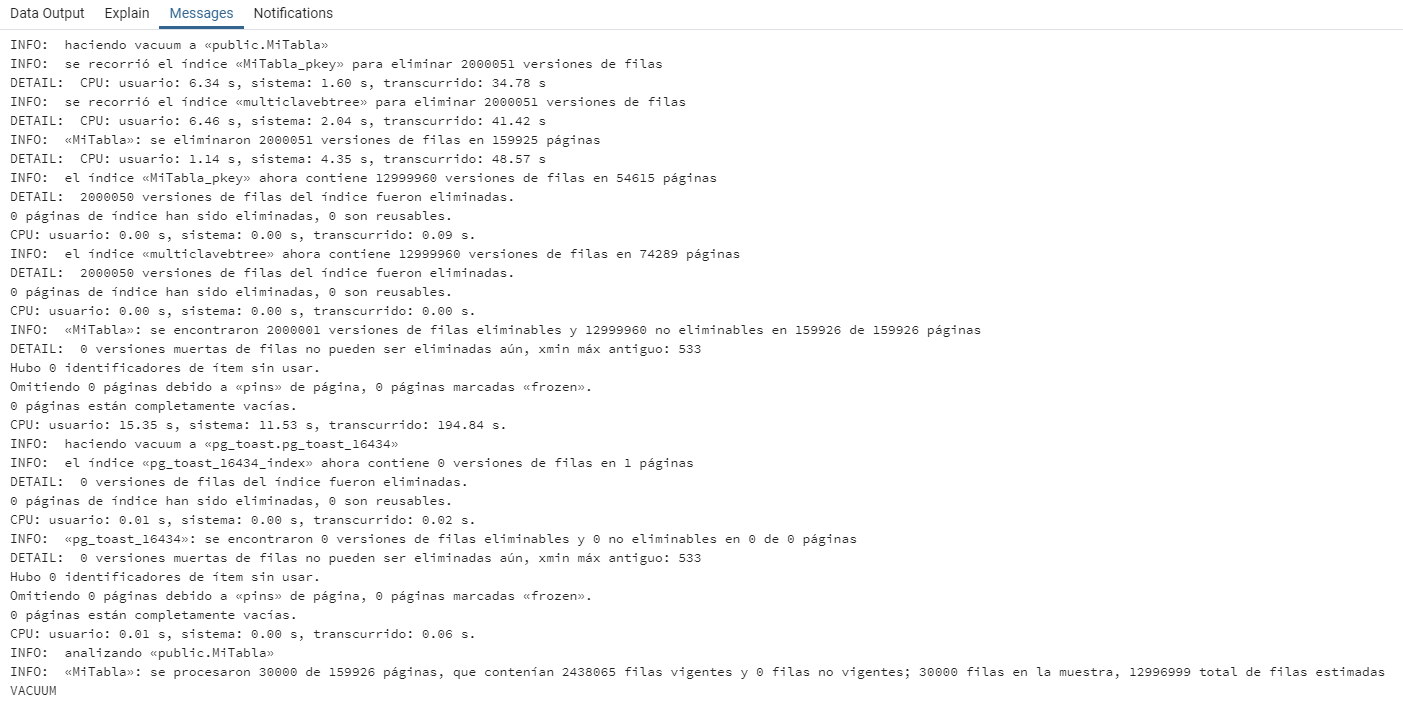
PostgreSQL simplemente marca esas tuplas como “muertas”, es decir, son tuplas que para el programa están eliminadas, de hecho, si realizamos una búsqueda de alguno de los valores borrados, nos va a devolver que esa tupla no existe en la tabla. Sin embargo, el tamaño no se modifica, y eso es por lo mencionado sobre las tuplas “muertas”. Esto se puede comprobar fácilmente observando el número de tuplas muertas existentes en la tabla antes y después de realizar un borrado.

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

Para optimizar el rendimiento de la base de datos, podemos ejecutaremos *VACUUM* sobre las relaciones existentes en la base de datos como *MiTabla*.

Al aplicar *VACUUM* sobre *MiTabla*, se va a reasignar el espacio eliminando las tuplas que había marcado como muertas y utilizando ese espacio para información realmente necesaria. Esto es muy útil especialmente en tablas que se actualizan con gran frecuencia al tener una gran cantidad de tuplas “muertas”. Para ver el resultado usaremos *VERBOSE* y *ANALIZE.*



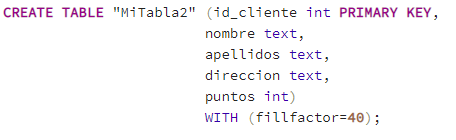


Una vez realizado el *VACUUM* sobre *MiTabla*, aplicaremos el módulo *pgstattuple* y vemos que ya no hay tuplas muertas.



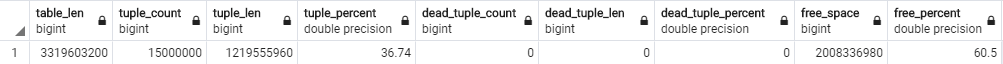
Cuestión 17. Crear una tabla denominada **MiTabla2** de tal manera que tenga un factor de llenado de tuplas que sea un 40% que el de la tabla **MiTabla** y cargar el archivo de datos anterior Explicar el proceso seguido y qué es lo que ocurre físicamente.

Para crear una tabla con un factor de llenado de tuplas del 40% usaremos *fillfactor.* Al no usar *fillfactor* al crear la tabla *“MiTabla”* hemos considerado que el factor de llenado de los bloques es del 100% ya que Postgres asigna esto por defecto.



Cuando cargamos de nuevo los 15.000.000 de datos del archivo de datos en esta nueva tabla y aplicamos el módulo *pgstattuple*:

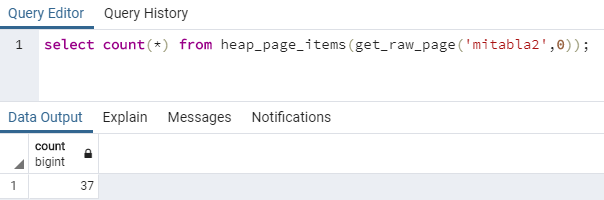




Podemos observar que la tabla ocupa mucho más espacio que antes. Esto tiene fácil explicación, cuando ponemos que las tablas han de estar ocupadas al 40% lo que conseguimos es que haya menos registros por bloque. Como consecuencia, se necesitarán muchos más bloques para almacenar el mismo número de registros, por tanto, nuestra tabla ocupará una mayor cantidad de espacio que la primera tabla creada.

Antes, el factor de bloque veíamos que era 94. Veamos por curiosidad cuál es el factor de bloque en mitabla2. Aplicando la siguiente consulta:



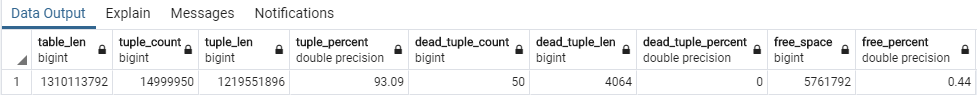


Vemos finalmente que el factor de bloque ahora es de 37 registros por bloque. Esto nos confirma lo dicho anteriormente.

Cuestión 18. Realizar las mismas pruebas que la cuestión 14 en la tabla **MiTabla2**. Comparar los resultados obtenidos con los de la cuestión 14 y explicar las diferencias encontradas.

**Prueba de borrado de 50 elementos aleatorios**

* MiTabla



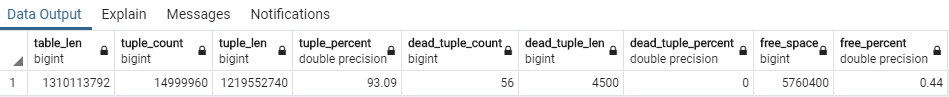
* MiTabla2



En la eliminación de elementos aleatorios de la tabla no hay ninguna diferencia. De nuevo, volvemos a ver como el número de tuplas muertas es de 50.

**Prueba de inserción de 10 elementos válidos y 6 no válidos**

* MiTabla



* MiTabla2



En la inserción de datos no hay ninguna diferencia tampoco, podemos comprobar que se han añadido 10 valores en *tuple\_count* y que el número de tuplas muertas se han incrementado en 6 al intentar introducir 6 valores ya existentes.

Las mayores diferencias son las del tamaño de la tabla y el espacio libre que hay ya que en *MiTabla2* hay una gran cantidad de espacio libre y ocupa mucho más en cuanto a tamaño.

Cuestión 19. Las versiones 11 y 12 de PostgreSQL permite trabajar con particionamiento de tablas. ¿Para qué sirve? ¿Qué tipos de particionamientos se pueden utilizar? ¿Cuándo será útil el particionamiento?

El particionamiento de tablas es una técnica usada sobre todo con el objetivo de crear una base de datos más eficiente. Esta técnica consiste básicamente en dividir lo que ocuparía una tabla entera en distintas partes más pequeñas. Con esto se consigue por ejemplo que, si las tablas más buscadas dentro de una base de datos se encuentran en la misma partición, su búsqueda sea más rápida ya que simplemente habría que recorrer dicha partición para encontrar los registros.

También, algo muy útil de esta técnica lo encontramos a la hora de realizar borrados y cargas masivas en nuestra base de datos. En lugar de borrar una tabla en concreto de nuestra base de datos, si la tenemos almacenada en una partición, podemos eliminar directamente la partición, algo que es mucho más rápido que realizar la primera operación.

Tipos de particionamientos:

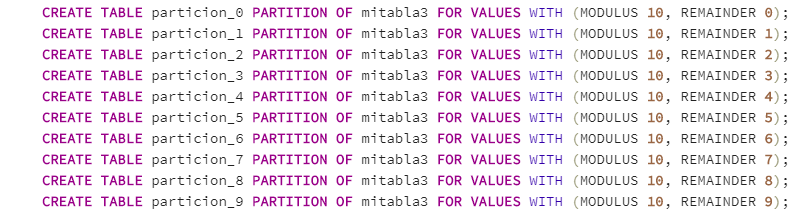
* **Particionamiento de rangos:** como su propio nombre indica, este particionamiento consiste en particionar la tabla dividiéndola en rangos creados a partir de una o varias columnas.
* **Particionamiento de lista:** este particionamiento, muy sencillo, consiste en dividir en listas la tabla definiendo qué valores clave componen cada lista.
* **Particionamiento de hash:** en este tipo de particionamiento la tabla se divide especificando un valor módulo y un resto para cada partición. Dependiendo del valor de resto que resulte al dividir el valor del campo por el módulo definido, el registro irá en una partición o en otra.

Cuestión 20. Crear una nueva tabla denominada **MiTabla3** con los mismos campos que la cuestión 2, pero sin PRIMARY KEY, que esté particionada por medio de una función HASH que devuelva 10 valores sobre el campo puntos. Explicar el proceso seguido y comentar qué es lo que ha ocurrido físicamente en la base de datos.

Para crear una tabla particionada por medio de un hash vamos a usar *PARTITION BY HASH (puntos).* Por otra parte, para hacer que el hash devuelva 10 valores sobre el campo puntos, crearemos 10 tablas con particiones de la tabla original MiTabla3, en la que cada tabla se encarga de hacer el módulo 10 al campo puntos.

Una vez cargados los datos, la función hash evalúa el campo puntos de la tupla y finalmente evalúa la tabla en la que se insertará.





Físicamente, se han creado dentro de la tabla las 10 particiones. Por tanto, cuando cualquier tupla se inserte en esta tabla, se va a evaluar de forma automática a qué partición debe de ir.

Si, por ejemplo, se inserta una tupla cuyo atributo *puntos* es 251, se realizará el módulo 10 sobre este campo. El resto de hacer el módulo será 1, por tanto, se evaluará de forma automática e irá a la tabla *particion\_1*.



Cuestión 21. ¿Cuántos bloques ocupa cada una de las particiones? ¿Por qué? Comparar con el número bloques que se obtendría teóricamente utilizando el procedimiento visto en teoría.

Veamos cuántos bloques ocupa cada una de las particiones:

Para saberlo debemos saber cuál es el tamaño de cada partición, esto lo obtendremos realizando:





El tamaño de la partición 0 es 132.874.240 y el tamaño de un bloque en Postgres es de 8.192 por tanto, la partición ocupará 16.220 bloques.

Calculo cuántos bloques ocupa las demás particiones realizando los cálculos de la misma manera que con la partición 0:

* **Partición 0:** 16.220 bloques.
* **Partición 1:** 127.229.952/8.192 = 15.531 bloques.
* **Partición 2:** 136.658.944/8.192 = 16.682 bloques.
* **Partición 3:** 112.353.280/8.192 = 13.715 bloques.
* **Partición 4:** 157.286.400/8.192 = 19.200 bloques.
* **Partición 5:** 123.650.048/8.192 = 15.094 bloques.
* **Partición 6:** 144.130.048/8.192 = 17.594 bloques.
* **Partición 7:** 119.816.192/8.192 = 14.626 bloques.
* **Partición 8:** 127.139.840/8.192 = 15.520 bloques.
* **Partición 9:** 128.999.424/8.192 = 15.747 bloques.

Por tanto, el tamaño en bloques total de la tabla sería de **159.929 bloques.**

Calculemos ahora el número de bloques teórico que ocuparía esta función hash:

* El número de cajones es 10 por tanto, la función hash nos devuelve 10 valores sobre el campo puntos.
* Como tenemos 15.000.000 de registros en el archivo, cada uno de los cajones tendrá 15.000.000/10 registros (usando equiprobabilidad), esto nos da que cada cajón consta de 1.500.000 de registros.

La longitud de registro calculada anteriormente es LR = 53 bytes

El factor de bloque en este caso sería: fRC = B/LR = 154 registros/bloque

Por tanto, cada cajón ocupa:

bRC = nRC / fRC = 1.500.000/154 = 9.741 bloques cada cajón.

Si tenemos 10 cajones: Tamaño total cajones = 10 \* 9.741 = **97.410 bloques en total.**

Vemos que hay una gran diferencia entre el valor real y el calculado teóricamente esto es debido a que PostgreSQL asigna información de control a los bloques que no se ha considerado en el cálculo teórico y además los bloques no se llenan al 100% de su capacidad.

**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 volvera crear la tabla MiTabla como en la cuestión 2. Cargar los datos que se encuentran originalmente en el fichero datos\_mitabla.txt

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

En PostgreSQL cada tabla tiene un archivo de almacenamiento dinámico donde se almacenan la mayoría de los datos. Si la tabla tiene alguna columna con valores potencialmente anchos, también puede haber un archivo TOAST asociado con la tabla, que se usa para almacenar valores demasiado amplios que no caben en la tabla principal. Habrá un índice válido en la tabla TOAST, si está presente además de índices asociados con la tabla principal.

Cada tabla e índice se almacena en un archivo de disco separado. Postgres por tanto, ofrece varias herramientas para monitorizar la actividad de la base de datos sobre el disco de tres maneras:

* Usando las funciones de SQL enumeradas en la tabla “Database Object Size Functions”.
* Funciones como *pg\_database\_size* para saber el tamaño total que ocupa una base de datos o *pg\_column\_size* que muestra el tamaño usado para guardar un valor.
* Usando el módulo oid2name
* Ayuda a los administradores a examinar la estructura de archivos utilizada por PostgreSQL.
* Usando la inspección manual de los catálogos de inspección del sistema.

Sin embargo, se recomienda usar las funciones SQL al ser las más fáciles. Usando *psql* en una base de datos recientemente “vacuumed” o analizada, se pueden emitir consultas para ver el uso del disco de cualquier tabla.

Por tanto, las estructuras sobre las que se puede recopilar información son sobre bases de datos, tablas, índices y atributos.

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



El proceso seguido consiste en crear un índice a través de la query mostrada en la imagen anterior.

Al pedirnos el enunciado un índice primario sobre el campo puntos es necesario primero ordenar los datos del archivo según el campo puntos.

Creamos el índice usando un árbol B y realizando una ordenación ascendente sobre puntos para poder realizar correctamente el índice primario.

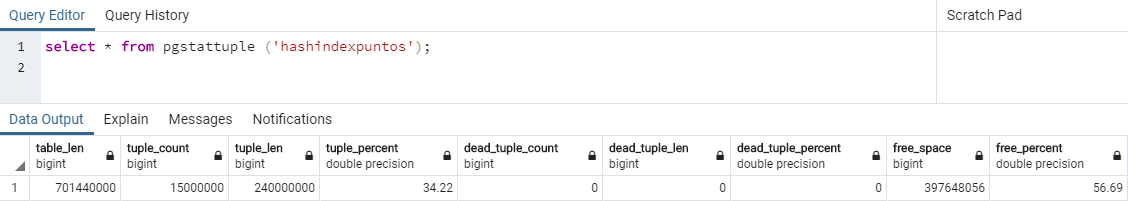
Cuestión 24. Crear un índice hash sobre el campo puntos y otro sobre id\_cliente.





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







Como se puede observar en los tamaños, el índice más pequeño es el índice primario sobre el campo puntos. Esto tiene una fácil explicación, ya que, al ser un **índice primario + campo no clave**, no posee cajones de punteros.

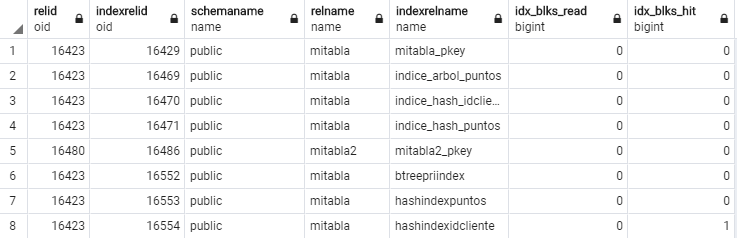
El segundo que menos ocupa es el índice hash sobre el campo id\_cliente, ya que es un **índice secundario + campo clave** y no posee cajones de punteros.

Sin embargo, si nos fijamos en el índice hash sobre el campo puntos el tamaño aumenta considerablemente, ya que al ser **secundario sobre campo no clave**, los cajones de punteros son necesarios. Esto hace que el tamaño sea considerablemente mayor.

Cuestión 26. Para cada una de las consultas que se muestran a continuación, ¿Qué información se puede obtener de los datos monitorizados por la base de datos al realizar la consulta? ¿Comentar cómo se ha realizado la resolución de la consulta? ¿Cuántos bloques se han leído? ¿Por qué? Importante, reinicializar los datos recolectados de la actividad de la base de datos antes de lanzar cada consulta:

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





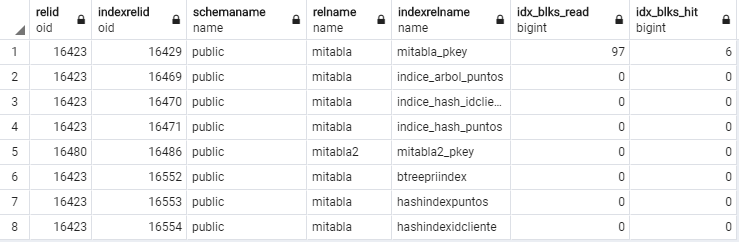
Al aplicar el módulo *pg\_statio\_user\_indexes* obtenemos los índices que están en la base de datos en las tablas usuario. Vemos que al realizar la consulta de seleccionar todo dónde el *id\_cliente = 8101000* se incrementa en uno el campo *idx\_scan\_hit* en el índice de MiTabla denominado *hashindexcliente.* Esto sucede ya que Postgres evalúa el coste para realizar esa consulta mediante búsqueda secuencial o mediante los índices disponibles. Finalmente ve que realizando esa consulta mediante el índice *hashindexcliente* va a ser más eficiente y va a tener un coste menor que si lo hace mediante otro índice o búsqueda.

El número de bloques a recuperar ha sido de 1 bloque.

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



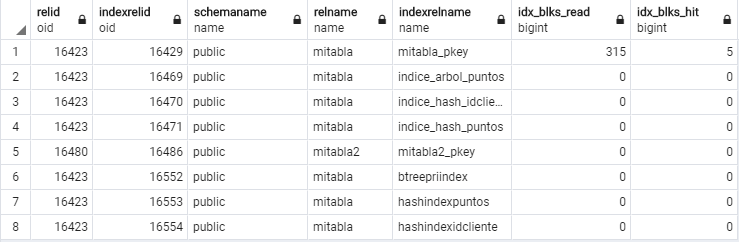




Para buscar todas las tuplas donde el *id\_cliente<30000*, PostgreSQL opta por hacer una búsqueda secuencial sobre el índice *mitabla\_pkey* leyéndose de esta forma 97 bloques y recuperando 6 bloques.

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

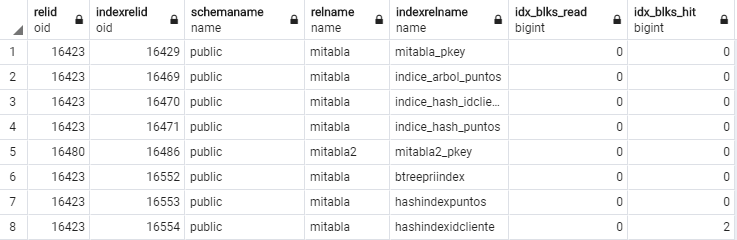




Para esta consulta, PostgreSQL evalúa cuál es la mejor opción y opta por hacer una búsqueda en el índice *mitabla\_pkey*. De esta manera lee 315 bloques y recupera 5 bloques.

1. Mostar la información de las tuplas con id\_cliente=34500 o id\_cliente=30.204.000.



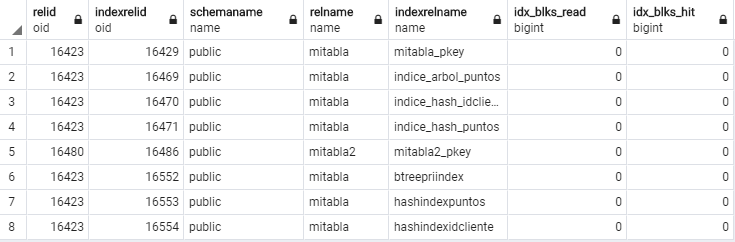


Como es una consulta mediante el campo *id\_cliente* y además es una consulta sobre una igualdad, Postgres sabe que lo más eficiente sería hacerlo mediante el índice *hashindexcliente* ya que evalúa directamente en que cajón está.

El campo *idx\_blks\_hit* en este caso sería 2 debido a que tiene que evaluar y recuperar aquellos bloques que cumplen la condición, es decir, aquellas en las que el id\_cliente es 34.500 y aquellas en las que el id\_cliente es 30.204.000

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



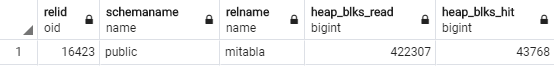


Al ver las estadísticas, vemos que no se ha leído ningún bloque ni se ha recuperado ningún índice de los índices. Y entonces, ¿no se ha leído ningún bloque?

Lo que pasa en esta consulta es que PostgreSQL decide que la mejor opción para resolver esta consulta es mediante una búsqueda secuencial ignorando la tupla donde el id\_cliente sea igual a 3.450.000. Esto es debido a que la consulta necesita recuperar absolutamente todas las tuplas excepto una.

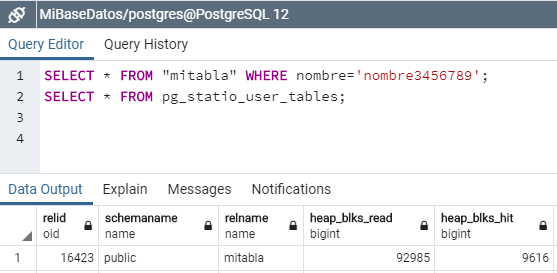
Por ello, la mejor opción es mediante una búsqueda secuencial, vemos las estadísticas de las tablas.

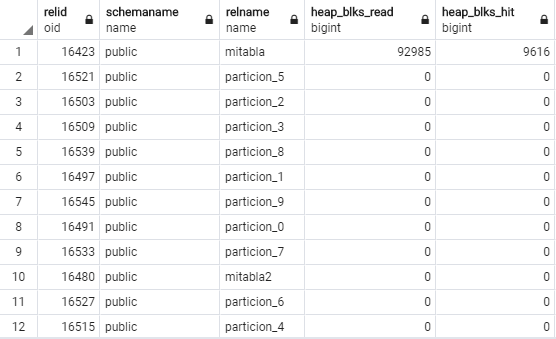




Ahora sí, vemos que cuando aplicamos esta consulta Postgres ha optado por leer la tabla *“MiTabla”* y recuperar los bloques y tuplas de forma secuencial.

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

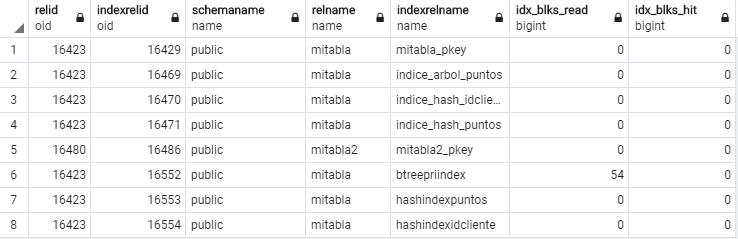




En esta consulta, PostgreSQL vuelve a hacer una búsqueda secuencial. Esto es debido a que no hemos creado ningún índice sobre el campo nombre y por tanto lee los bloques de forma secuencial hasta dar con la tupla.

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

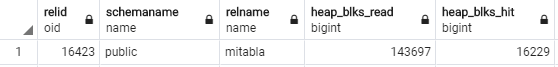




En esta consulta, Postgres decide buscar las tuplas con puntos igual a 650 mediante el índice primario B sobre el campo puntos al ser la mejor opción y la que menor coste va a tener al realizar la consulta.

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

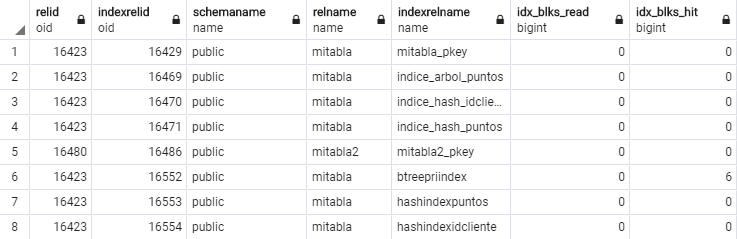




Para realizar esta consulta, PostgreSQL opta por realizar una búsqueda secuencial leyendo 143.697 bloques y recuperando 16.229 bloques donde hay tuplas que cumplen con la condición.

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

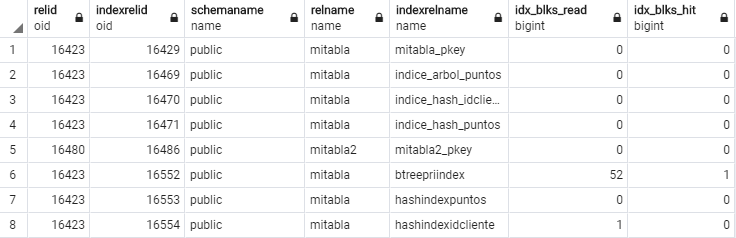




En esta consulta, PostgreSQL realiza la búsqueda mediante el índice *btreepriindex,* que es un árbol B primario ordenado por el campo puntos. Esta es la mejor opción y con menor coste para realizar la consulta.

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



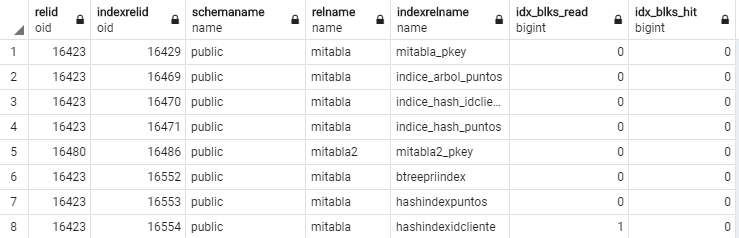


Para realizar esta consulta, PostgreSQL utiliza tanto el índice hash sobre el campo clave *id\_cliente* como el índice primario de árbol B sobre el campo puntos.

Estos índices hacen la consulta más eficiente.

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





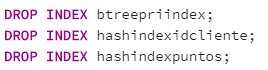
Para realizar esta consulta, PostgreSQL mira el índice hash sobre el campo clave *id\_cliente* ya que sólo habrá una tupla a recuperar al ser el campo clave. Una vez que mira esa tupla, comprueba el campo *puntos* mirando si es 230, en el caso de que lo sea, recupera esa tupla y en caso contrario, no recupera ninguna tupla ya que no existe.

Mirando por el campo clave *id\_cliente* en vez de por el campo *puntos* Postgres aumenta el rendimiento de la consulta al ahorrarse leer más bloques de datos ya que puede haber muchas tuplas con puntos = 230.

Cada vez que hacemos una nueva query hay que ejecutar este comando de modo que podamos ir viendo qué bloques se leen en cada momento.



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

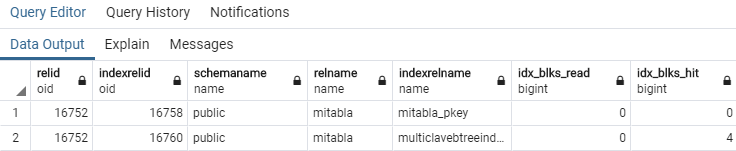




Cuestión 28. Para cada una de las consultas que se muestran a continuación, ¿Qué información se puede obtener de los datos monitorizados por la base de datos al realizar la consulta? ¿Comentar cómo se ha realizado la resolución de la consulta? ¿Cuántos bloques se han leído? ¿Por qué? Importante, reinicializar los datos recolectados de la actividad de la base de datos antes de lanzar cada consulta:

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



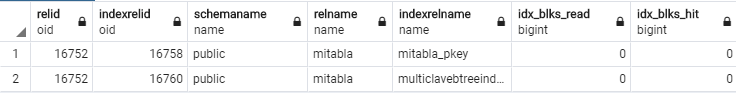


En este caso tenemos una consulta cuyo objetivo es recuperar las tuplas que tienen un valor de puntos igual a 200 y un valor de nombre igual a *“nombre3456789”,* por tanto, para la resolución correcta y eficiente de esta consulta es mirar el índice.

Esto es porque al ser un “and”, sólo es necesario recuperar los registros con el valor de uno de los dos campos. La búsqueda por tanto se realizaría sobre el campo cuya búsqueda tenga menor coste. Por este motivo en esta consulta se realiza una búsqueda a través del índice.

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







En este caso nos encontramos con algunas diferencias respecto al caso anterior.

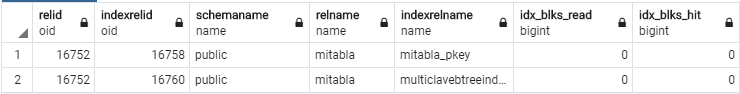
Primero podemos observar que se trata de una consulta en la que se buscan valores a través de un “or”, esto es que sirven tanto las tuplas que tienen un valor 200 en el campo puntos como las tuplas que tienen como nombre “*nombre3456789”*.

Como se observa en las capturas de pantalla sobre la monitorización de la información se observa que al ver si se ha buscado por el índice no salen bloques leídos.

Bien, esto es porque al ser una búsqueda en la que te sirven tanto unos valores como otros, sale más barato realizar una búsqueda secuencial sobre la tabla que buscar en el índice árbol B, ya que, si realizamos la búsqueda en el árbol B, nos tendríamos que leer prácticamente todo el árbol, algo que es menos eficiente que buscar secuencialmente en el archivo.

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



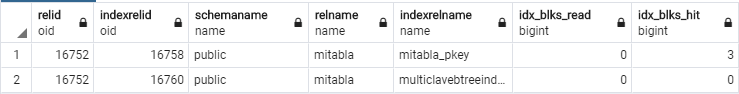




En este caso podemos ver cómo otra vez es una búsqueda con un “or” en su enunciado. Además, aquí nos encontramos con la clave primaria *id\_cliente* como campo de búsqueda. Como se ha mencionado anteriormente, en los casos en los que hay un “or”, es más eficiente realizar una búsqueda secuencial en el archivo que leer casi todo el árbol para encontrar los valores.

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



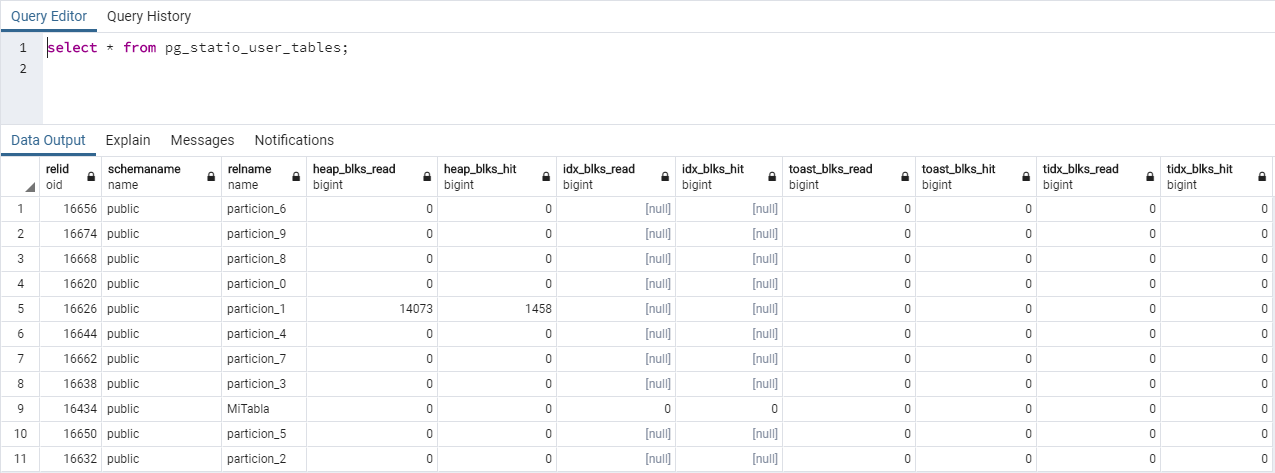


Pasamos a la última consulta y vemos que en este caso es con un “and”. Por tanto, como hemos mencionado anteriormente en este caso sí que sale rentable realizar una búsqueda según nuestro índice. Además, en este caso sabemos que para encontrar un registro con un *id\_cliente*, al ser la primary key y al no tener ningún índice para ese valor y al no estar ordenado, habría que leer secuencialmente el archivo. Esto sería mucho más costoso que buscar en nuestro índice multiclave el valor del nombre propuesto en la consulta. Por este motivo, se realiza la búsqueda sobre el índice.

Cuestión 29. Crear la tabla **MiTabla3** como en la cuestión 20. Para cada una de las consultas que se muestran a continuación, ¿Qué información se puede obtener de los datos monitorizados por la base de datos al realizar la consulta? ¿Comentar cómo se ha realizado la resolución de la consulta? ¿Cuántos bloques se han leído? ¿Por qué? Importante, reinicializar los datos recolectados de la actividad de la base de datos antes de lanzar cada consulta:

1. Mostrar las tuplas cuyos puntos valen 200.





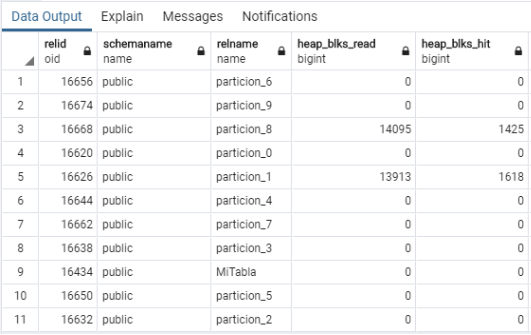
1. Mostrar las tuplas cuyos puntos valen 200 y 300.





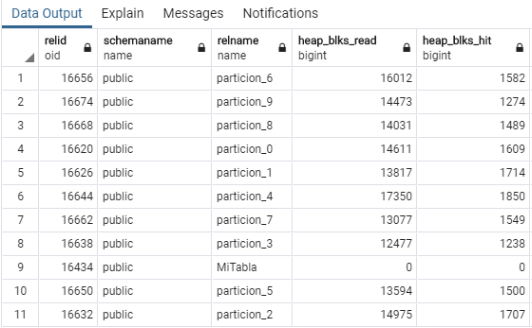
1. Mostrar las tuplas cuyos puntos valen 200 o 202





1. Mostrar las tuplas cuyos puntos son > 500.



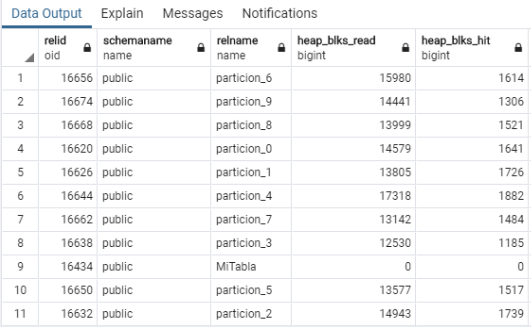


Como se puede observar, los registros que se recuperan en esta consulta son muy variados, por lo que hay que leerse todas las particiones para recuperarlos todos.

Al recuperar todos los puntos que sean mayores de 500, existen valores en módulo 10 que se guardan en todas las particiones, por tanto, es necesario leer todas las particiones para recuperar las tuplas.

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



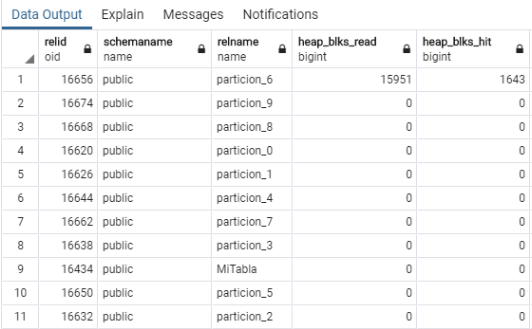


Como se puede observar para recuperar todos estos registros entre ese rango de valores, necesitamos acceder a todas las particiones, ya que hay registros repartidos entre todas ellas.

Por eso el número de bloques es muy elevado, y por eso se han leído bloques en todas las particiones.

6. Mostrar las tuplas cuyos puntos son 800





Cuestión 30. A la vista de los resultados obtenidos de este apartado, comentar las conclusiones que se pueden obtener del acceso de PostgreSQL a los datos almacenados en disco.

Como se puede observar en las distintas consultas realizadas en el último apartado, podemos observar que PostgreSQL escoge la partición del hash en la que debe mirar (la cual viene determinada por la función hash que en este caso es módulo 10 de puntos) para acceder a la tupla o tuplas que se buscan en la consulta.

Conociendo el funcionamiento de una organización hash, sabemos que cuando la consulta es la búsqueda de un dato concreto, es decir, una igualdad, la eficiencia de la búsqueda en la consulta es mucho mayor. Sin embargo, cuando buscamos valores entre rangos, o valores mayores o menores que un cierto límite, el número de bloques leídos aumenta considerablemente.

**Bibliografía (PostgreSQL 12)**

* Capítulo 1: Getting Started.
* Capítulo 5: 5.5 System Columns.
* Capítulo 5: 5.11 Table Partitioning.
* Capítulo 11: Indexes.
* Capítulo 19: Server Configuration.
* Capítulo 24: Routine Database Maintenance Tasks.
* Capítulo 28: Monitoring Database Activity.
* Capítulo 29: Monitoring Disk Usage.
* Capítulo VI.II: PostgresSQL Client Applications.
* Capítulo VI.III: PostgresSQL Server Applications.
* Capítulo 50: System Catalogs.
* Capítulo 68: Database Physical Storage.
* Apéndice F: Additional Supplied Modules.
* Apéndice G: Additional Supplied Programs.