Titulación: Grado en Ingeniería Informática e Ingeniería en Sistemas de Información
Curso: 2021-2022. Convocatoria Ordinaria de Junio
Asignatura: Bases de Datos Avanzadas – Laboratorio

Practica 1: Arquitectura PostgreSQL y almacenamiento físico

ALUMNO 1:
Nombre y Apellidos: Victor Herguedas Martín
DNI: 03201939V

ALUMNO 2:
Nombre y Apellidos: Andrés Delgado Blas
DNI: 09127332N

Fecha: 16/03/20022

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.

Es obligatorio proporcionar una explicación a lo que está ocurriendo en PostgreSQL cuando así se indica en la cuestión. No solo vale poner un pantallazo. La ausencia de una explicación hará que sea invalidada esa cuestión.

Plazos

Trabajo de Laboratorio: Semana 7 Febrero, 14 Febrero, 21 Febrero, 28 Febrero y 7 de Marzo.

Entrega de práctica: Día 14 de Marzo. Aula Virtual

Documento a entregar: Un fichero con formato ZIP con las respuestas a las cuestiones planteadas, así como los ficheros de log de postgresql relacionados con la resolución de la práctica y el fichero history del cliente pgcli. El fichero se deberá llamar: `DNIdelosAlumnos_PL1.zip`

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 14. 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. Antes de comenzar es obligatorio configurar lo que se comenta en la cuestión 0. En esta práctica no se permite el uso de pgAdmin u otra herramienta administrativa gráfica. Hay que resolver la práctica con el cliente pgcli de Python.

Cuestión 0. Configurar el fichero de Error Reporting and Logging de PostgreSQL para que aparezcan recogidas las sentencias SQL DDL (Lenguaje de Definición de Datos) + DML (Lenguaje de Manipulación de Datos) generadas en dicho fichero. No se pide activar todas las sentencias. No activar la duración de la consulta. También se debe de configurar el log para que en el comienzo de la línea de registro de la información del log ("line prefix") aparezcan vuestros DNI's y el nombre del host con su puerto. ¿Cómo se ha realizado la configuración?

Para poder configurar este fichero hemos accedido a C:\Program Files\PostgreSQL\14\data y hemos accedido al archivo postgresql.confg y modificado los siguientes campos:
- log_statement = 'mod'
Para activar las sentencias DDL (Lenguaje de definición de datos) y DML (Lenguaje de manipulación de datos)
- log_line_prefix = '03201939V, %r ,%m [%p]'.
A continuación, reiniciamos el servidor para que se aplique la nueva configuración.
Además hemos añadido las extensiones necesarias para esta práctica:
- CREATE EXTENSION pageinspect; - CREATE EXTENSION pgstattuple;

Organización de Archivos en PostgreSQL

Cuestión 1. Crear una nueva Base de Datos que se llame **PL1**. Después crear una tabla **alumno** que tenga un campo denominado carnet de tipo integer, un campo nombre de tipo text, un campo denominado código_de_carrera de tipo integer, un campo edad de tipo integer y un campo denominado puntuación de tipo integer. Notar que no hay primary key en la tabla por el momento. Localizar los ficheros relacionados con la tabla. ¿cómo se localizan? ¿Cuánto ocupan y por qué?

Para crear la base de datos hemos abierto el terminal y hemos ejecutado la siguiente sentencia para crear la base de datos. - createdb -U postgres -p 5432 PL1. Para unirnos a la base sacamos el terminal y lo hacemos a través de PgCli: - pgcli -U postgres -p 5432 PL1. A continuación, metemos la contraseña. Para crear la tabla abrimos el editor y ponemos el comando \e y añadimos el siguiente código: CREATE TABLE alumno (carnet integer, nombre text, codigo_de_carrera integer, edad integer, puntuacion integer); Para localizar los ficheros relacionados con la tabla recurrimos al siguiente comando: - pg_relation_filepath('alumno'). Nos devuelve que está en base/18234/18549 dentro de la carpeta de PostgresSQL en archivos de programa. Estos archivos ocupan 823.536 debido a que tiene los bloques, los archivos de control de los bloques y la información de los alumnos. Si la tabla hubiese ocupado más de un giga se hubiese dividido en varios archivos. Con el comando: - SELECT * FROM heap_page_items(get_raw_page('alumno', 0));

Cuestión 2. Insertar los datos que se entregan en el fichero de texto denominado **datos_alumno.txt**. ¿Cuánto ocupa la información original a insertar? ¿Cuánto ocupa la tabla ahora? ¿Por qué? ¿Cuánto tarda en cargar? Calcular teóricamente el tamaño en bloques que ocupa la relación **Alumno** tal y como se realiza en teoría. ¿Concuerda con el tamaño en bloques que nos proporciona PostgreSQL? ¿Por qué?

Para insertar ese fichero en la tabla alumnos hemos usado el comando. copy alumno from 'C:/Clase/Bases de datos avanzadas/PL1/datos_alumno.txt' with delimiter ';' ; La información original a insertar ocupa 494.085 Kb. Ahora la tabla ocupa 823.536Kb Debido a que ahora se ha metido más información de control a los bloques. La tabla tarda en cargar los datos 36.432 segundos. El tamaño de bloques según postgres. Usando el comando select * from pg_class, sacas el tamaño de bloques de la tabla alumno y visualizas en el campo relpages que tiene 102.942 bloques. El tamaño de cada bloque se calcula dividiendo el tamaño de la tabla 823.536Kb/ 102.942 bloques de la tabla = 8Kbytes, que es el tamaño de los bloques. Ahora hacemos el calculo según teoría. En primer lugar calculamos la longitud de los registros que es la siguiente. Longitud del registro = 4 +4 +14 +4 + 4 + cabecera(23)= 53 bytes. 4 bytes cada entero y luego un texto con catorce carácteres que son 14 bytes. El factor de bloque es el siguiente. 8192 - 24(cabecera de bloque) / 53 = 153 bytes (cerca del verdadero el cual son 136). Por lo que el Nº de bloques = 14.000.000/ 153 = 9.1504 bloques. Hay diferencia del número de bloques ya que teóricamente no estamos tomando en cuenta que el bloque útil es menor debido a que tiene más bytes de información de control como itemId o un espacio especial, y para el campo registro puede haber una cabecera de tamaño variable

Cuestión 3 ¿Cuál es el factor de bloque medio real de la tabla **Alumno**? 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 real de la tabla alumno lo sacamos de la siguiente manera. Buscamos la información de los registros con el siguiente comando SELECT * FROM heap_page_items (get_raw_page('alumno', 0)); y vemos que en total tiene 136 registros. Esto también se podría haber hecho con un count directamente. El factor de bloque teórico está calculado en el apartado anterior (cuestión 2)

Cuestión 4. Realizar una consulta que muestre los datos del alumno que tiene el carnet 24567890. ¿Cuántas tuplas se obtienen y cuántos bloques se leen por Postgres? ¿Por qué? Comparar con los resultados obtenidos al aplicar el método visto en teoría.

La consulta es la siguiente. - select ctid,* from alumno where carnet = 24567890. En teoría se leerían todos los bloques, es decir 102.942 ya que se tiene que leer secuencialmente. Aquí no se leen todos esos ya que depende de los bloques que hayan ya cargados en memoria. Con postgres primero reseteamos las tablas de estadísticas - select pg_stat_reset() y acontinuación hacemos la consulta. Por ultimo mostramos las estadísticas -select * from pg_statio_user_tables y vemos en el campo heap_blks_read que se han leído 100349 bloques. Como podemos ver algo menos que en la teoría

Cuestión 5. Realizar una consulta que muestre el número de alumnos que tienen una puntuación de 525. ¿Cuántas tuplas se obtienen y cuántos bloques se leen por Postgres? ¿Por qué? Comparar con los resultados obtenidos al aplicar el método visto en teoría.

Para ver el número de alumnos que tienen una puntuación de 525 hacemos la siguiente consulta. - select count(*) as total from alumno where puntuacion = 525 group by puntuacion. Nos dice que hay un total de 1345 alumnos con esa puntuación (número de tuplas). Para ver el número de bloques leídos por postgres en primer lugar reseteamos las estadísticas. - select pg_stat_reset(). A continuación, volvemos a hacer la consulta anterior y vemos la tabla de estadísticas. - select * from pg_statio_user_tables. Aquí buscamos la tabla alumno y vemos que ha leído 102653 bloques (heap_blks_read). En teoría se leerían todos los bloques, es decir 102.942 ya que se tiene que leer secuencialmente. Aquí no se leen todos esos ya que depende de los bloques que hayan ya cargados en memoria.

Cuestión 6. Crear una tabla **Alumno2** cuyas tuplas estén ordenadas físicamente por el campo puntuación que tenga la misma información que el fichero **datos_alumno.txt**. Indicar el proceso de generación de dicha tabla ordenada.

Primero, creamos la tabla alumno2. - create table alumno2(carnet integer, nombre text, codigo_de_carrera integer, edad integer, puntuacion integer); Ahora, insertamos los datos ordenados en la tabla. - insert into alumno2 (select * from alumno order by puntuacion)

Cuestión 7. Repetir las cuestiones 4 y 5 sobre la tabla **Alumno2** y comparar los resultados obtenidos indicando las conclusiones obtenidas.

Cuestión 4:
Esta según teoría se leerían todos los bloques ya que al no estar ordenado por el campo carnet se tendría que hacer una búsqueda secuencial (102.942 bloques). Ahora, hacemos el proceso de resetear estadísticas, hacer la consulta para buscar ese carnet y mirar la tabla de estadísticas alumno2. -select pg_stat_reset(). - select ctid,carnet from alumno2 where carnet = 24567890. - select * from pg_statio_user_tables. En total ha leído 86937 bloques (heap_blks_read). Hay diferencia de bloques con la teoría debido a que ya había bloques cargados en memoria previamente.

Cuestión 5:
La tabla está ordenada por puntuación por lo que en este caso la búsqueda sería binaria. En teoría se leerían log2(102.942) + 1345/56 -1 = 17 + 25 -1 = 41 bloques. Ahora, hacemos el proceso de resetear estadísticas, hacer la consulta para buscar ese carnet y mirar la tabla de estadísticas alumno2. -select pg_stat_reset(). select count(*) as total from alumno2 where puntuacion = 525 group by puntuacion. - select * from pg_statio_user_tables. Vemos que se han leído

Cuestión 8. Borrar 2.000.000 de tuplas de la tabla **Alumno** de manera aleatoria usando el valor del campo carnet. ¿Qué es lo que ocurre físicamente en la base de datos? ¿Se observa algún cambio en el tamaño de la tabla y estructuras asociadas a ella? ¿Por qué? Adjuntar el código de borrado.

Para borrar 2.000.000 de tuplas usamos esta consulta. - delete from carnet where nombre in (select carnet from alumno2 order by RANDOM() limit(20000)); Los datos se han borrado lógicamente pero siguen estando en memoria como tuplas muertas por lo tanto la tabla y sus estructuras siguen ocupando lo mismo. - select * from pgstattuple(´alumno´). (843300864 bytes)

Cuestión 9. En la situación anterior, ¿Qué operaciones se pueden aplicar a la base de datos **PL1** para optimizar el rendimiento de esta? Aplicarla a la base de datos **Alumno** de tal manera que se recupere el mayor espacio posible. Comentar cuál es el resultado final y qué es lo que ocurre físicamente.

Podemos usar vaccum que borra definitivamente las tuplas muertas pero no libera espacio en el disco. En cambio, vacuum full si libera los espacios ya que borra las tuplas muertas y luego compacta las tuplas. -VACUUM full alumno; Antes de ese código podíamos ver las tuplas muertas con el código. - select * from pgstattuple(´alumno´). El tamaño anterior era 843300864 bytes y el tamaño actual es 722829312 bytes. Estos datos usando pgsttatuple antes y después.

Cuestión 10. Crear una nueva tabla denominada **Alumno3** con los mismos campos que la cuestión 1 y que esté particionada por medio de una función HASH que devuelva 30 valores sobre el campo carnet. Insertar los datos del fichero **datos_alumno.txt** Explicar el proceso seguido y comentar qué es lo que ha ocurrido físicamente en la base de datos. ¿Cuándo será útil el particionamiento? ¿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.

Primero creamos la tabla alumno3 particionada. - create table alumno3(carnet integer, nombre text,codigo_de_carrera integer, edad integer, puntuacion integer)partition by hash(carnet) ; Ahora añadimos los cajones de esa tabla. CREATE TABLE alumno3_p1 PARTITION OF alumno3 FOR VALUES WITH (MODULUS 30, REMAINDER 0); y así siguiendo una sucesión hasta este último cajón.CREATE TABLE alumno3_p30 PARTITION OF alumno3 FOR VALUES WITH (MODULUS 30, REMAINDER 29); Ahora, insertamos los datos. - copy alumno3 from 'C:/Clase/Bases de datos avanzadas/PL1/datos_alumno.txt' with delimiter ';' . El particionamiento será útil para cuando se hagan consultas por el campo carnet, ya que solo habrá que leerse el cajón donde se encuentre un carnet específico. Ahora buscamos en las particiones con el pg_class para ver cuantos bloques ocupa cada una de ellas. - select relpages, oid, relname from pg_class where relname like ('alumno%') order by relpages. Vemos que cada partición tiene 3442 bloques (relpages). Ahora lo calculamos por teoría. 14.000.000 registros / 30 cajones = 466.667 registros por cajón. Nº de bloques por cajón = 466.667registros/8192 bytes = 57 bloques. No nos da el mismo resultado puesto que el bloque útil que usa postgres es menor debido a que tiene información de control. Además, los registros tienen cabeceras aquí no consideradas.

Cuestión 11. Repetir las cuestiones 4 y 5 sobre la tabla **Alumno3** y comparar los resultados obtenidos con lo visto anteriormente en las tablas **Alumno** y **Alumno2** obteniendo conclusiones sobre el método de partición.

Cuestión 4:
Según la teoría se tendría que leer un cajón entero, es decir 3442 bloques. Ahora, hacemos el proceso de resetear estadísticas, hacer la consulta para buscar ese carnet y mirar la tabla de estadísticas dealumno3. - select pg_stat_reset(). - select ctid,carnet from alumno2 where carnet = 24567890. - select * from pg_statio_user_tables. En este caso ha leído 0 bloques debido a que hemos tenido suerte ya que estaban cargados en memoria. Además, vemos que en la partición 4 a hecho un hit en memoria de 3434 bloques por lo que averiguamos que está en ese cajón el registro.

Cuestión 5:
Según la teoría se tendría que leer todos los blques de forma secuencial debido a que la partición no está preparada para el campo puntuación (102.942 bloques). En este caso se leen diferentes bloques de cada cajón por lo que hacemos un - select sum(heap_blks_read) from pg_statio_user_tables. Así sumamos los bloques leídos en cada partición y suman un total de 102954. En este caso hay más bloques que en teoría debido a que está leyendo todos los bloques de todos los cajones y al ser estructura de cajones y no un archivo secuencial hay mayor fragmentación e información de control de las particiones.

Indexación de PostgreSQL

PostgreSQL soporta indexación definida por el usuario para ayudar a acelerar ciertas consultas. Entre otros tipos de índices soporta árboles y hash. En este apartado se va a trabajar sobre ambos tipos de índices, pudiendo observar cómo se organizan internamente y su funcionamiento.

Cuestión 12. Borrar todas las tablas **Alumno** y **Alumno2**. Crear una nueva tabla que se llama **Alumno** con los mismos campos que la cuestión 2 pero el campo carnet será la Primary Key (PK). ¿Qué ocurre físicamente? ¿Qué ficheros se crean y por qué? ¿Por qué tienen ese tamaño?

Comenzamos borrando las tablas. - Drop table alumno. - Drop table alumno2. Ahora creamos la nueva tabla. - CREATE TABLE alumno (carnet integer, nombre text, codigo_de_carrera integer, edad integer, puntuacion integer, primary key(carnet)); Físicamente se ha creado un índice para primary key llamado alumno_pkey el cual hemos descubierto usando el comando. - select relname from pg_class where relname like('%alumno%'). Para ver el tamaño del índice - select * from pgstattuple ('alumno_pkey'). De aquí hemos tomado el campo table_len. Esta tabla ocupa 8192 bytes. Hacemos el mismo procedimiento para la tabla alumno donde se encuentran los registros de datos y obtenemos que ocupa 0bytes. Alumno ocupa 0 bytes debido a que no hay ningún registro introducido todavía por lo que no hay bloques y prueba_pkey ocupa 1 bloque para almacenar indices aunque está todavía vacío.

Cuestión 13. Cargar el fichero de datos **datos_alumno.txt**. ¿Cuánto tarda ahora en cargar? Comparar con la cuestión 2.

Para cargar el archivo usamos el comando. - copy alumno from 'C:/Clase/Bases de datos avanzadas/PL1/datos_alumno.txt' with delimiter ';' ; y ha tardado 2 minutos 4 segundos. Ha tardado considerablemente más que la 2 (36.432 segundos) debido a que ha ido creando el índice mientras se insertaban los datos.

Cuestión 14. Crear un índice de tipo árbol para el campo carnet. ¿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? Indicar el procedimiento seguido e incluir el código SQL utilizado.

Creamos el índice. CREATE INDEX alumno_btree_carnet ON alumno (carnet); Por defecto en postgresSQL cuando creas un índice se crea del tipo btree. Este índice se crea como si fuese una tabla más. En este caso usamos el siguiente comando para averiguarlo. - select pg_relation_filepath ('alumno_btree_puntuacion'). Vemos que se ha almacenado en la siguiente ubicación data/base/18234/18558. Para ver el número de bloques usamos el siguiente comando. - select relname, relpages from pg_class where relname like('alumno_btree_carnet'). Vemos que el índice tiene 38.389 bloques. Para ver más información del índice usamos el comando. - select * from pgstatindex('alumno_btree_carnet'); Aquí observamos que hay tres niveles (tree_level =2) (0,1,2). También de aquí sacamos que el nivel hoja tiene 38252 bloques (leaf pages), el nivel intermedio tiene 136 bloques (internal pages) y sabemos que la raíz tiene 1 bloque. Ahora buscamos los registros por bloque que tiene el nivel 2 (raíz). SELECT * FROM bt_page_stats('alumno_btree_carnet', 290); Este tiene 135 registros por bloque. Ahora buscamos los registros por bloque que tiene el nivel 1 (intermedio). - SELECT * FROM bt_page_stats('alumno_btree_carnet', 2000); Este tiene 285 registros por bloque. Finalmente, hacemos lo mismo con el bloque 280 que pertenece al nivel hoja y vemos que este nivel tiene 367 tuplas por bloque (live_item).

Cuestión 15. 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 14.

En primer lugar hemos sacado la longitud de las tuplas con el comando. - SELECT * FROM bt_page_stats('alumno_btree_carnet', 215); De aquí obtenemos el tamaño de longitud a bloque ya que obtenemos que un registro ocupa 16 bytes y le restamos los 4 bytes del campo = 12 bytes. Hacemos lo mismo con un bloque de la hoja y obtenemos que el tamaño del puntero a registro es el mismo (12 bytes).
A continuación sacamos el bloque útil del bloque hoja:
Butil=8192 bytes - 24(cabecera) - 4*n(matriz de identificadores 4 bytes*elemento) - 812 bytes (espacio libre) - 16 bytes (special)=7340 - 4*n
Ahora obtenemos el bloque útil del nivel intermedio
Butil=8192 bytes - 24(cabecera) - 4*n(matriz de identificadores 4 bytes*elemento) - 2460 bytes (espacio libre) - 16 bytes (special)=5692 - 4*n
El espacio libre lo hemos obtenido de ver la diferencia entre upper y lower con el siguiente comando. ELECT * FROM page_header(get_raw_page('alumno_btree_carnet', 291)); y lo mismo para el otro tipo de bloque.
Ahora sacamos el número de punteros a bloque por bloque del nodo raíz e intermedio.
n*Lploque + (n-1)*Lcarnet <= 5692- 4n ---> n*12+(n-1)*4<=5692 - 4n ---> n<=5696/20= 284 registros por bloque intermedio/raíz
Ahora sacamos el número de registros por bloque del nodo hoja.
nh*(Lcarnet + Lpreg)+ Lpbloque <= 7340 - 4nh --> nh*(4+12)+12<=7340 - 4nh --> nh<=7328/20=366 registros por bloque hoja
Nos hemos acercado bastante con la teoría a la realidad. Nos ha variado en uno el número de registros por bloque que suponemos que ha sido al no tener en cuenta algún dato de control en el bloque útil. En cambio en la raíz hay mas diferencia ya que los datos que hemos obtenido sería en caso de estar llena

y la raíz no lo está ya que todabía puede incluir registros en ese bloque.

Cuestión 16. Utilizando el procedimiento comentado en teoría para buscar sobre índices de tipo árbol, determinar el "**número de bloque leído**" de cada estructura hasta encontrar el registro de datos que tiene el carnet 26935691. Mostrar el código SQL utilizado. Se suministra el procedimiento almacenado **hex_to_int** que permite convertir el valor hexadecimal del ítem en un valor entero.

En primer lugar creamos la función hex_to_int. - \i C:\Users\victo\Downloads\hex_to_int (2).sql. Ahora buscamos el bloque raíz del árbol para poder comenzar la búsqueda. - select * from pgstatindex (' alumno_btree_carnet '). Vemos que está en elo bloque 290 (root_block_no). Ahora que tenemos el bloque buscamos a ver si está el registro. - select * from bt_page_items('alumno_btree_carnet',290) where hex_to_int(data)=26935691. Vemos que no está en este nivel, por lo que buscamos el valor más cercano sin pasarse. - select ctid, hex_to_int(data) from bt_page_items('alumno_btree_carnet',290) where hex_to_int(data)<26935691 order by hex_to_int(data) desc. El valor más cercano es | (38195,1) | 26928496. Repetimos el paso de buscar en el bloque en este caso el 38195 a ver si está el registro. - select * from bt_page_items('alumno_btree_carnet',38195) where hex_to_int(data) =26935691. Vemos que no se encuentra en este bloque tampoco por lo que volvemos a buscar el valor más cercano sin pasarse. - select ctid, hex_to_int(data) from bt_page_items ('alumno_btree_carnet',38195) where hex_to_int(data)<26935691 order by hex_to_int(data) desc. En este caso es el valor | (38212,1) | 26935450. Ahora buscamos si el registro se encuenta en ese bloque. - select * from bt_page_items('alumno_btree_carnet',38212) where hex_to_int(data) =26935691. Vemos que sí por lo que ahora vemos el puntero y vemos que el htid es 24381 por lo que el resgistro se encuentra en el bloque 24381.

Cuestión 17. Crear un índice de tipo árbol para el campo puntuación. ¿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? Indicar el procedimiento seguido e incluir el código SQL utilizado.

Para crear el índice por el campo puntuación usamos el siguiente comando. - CREATE INDEX alumno_btree_puntuacion ON alumno (puntuacion); Para ver el tamaño del archivo usamos el siguiente comando. - select * from pgstattuple('alumno_btree_puntuacion'). Vemos que ocupa 101146624 bytes. - para ver el número de bloques usamos. - select relname, relpages, oid from pg_class where relname like('alumno_btree_puntuacion') y obtenemos 12.347 bloques. Para ver los bloques que tiene por nivel usamos. - select * from pgstatindex('alumno_btree_puntuacion'); Hemos obtenido que el bloque tiene 12286 bloques raíz, 60 bloques intermedios y 1 bloque raíz. Ahora para buscar las tuplas que tiene cada nivel hemos usado este comando hasta que hemos encontrado 3 bloques 1 de cada nivel. - SELECT * FROM bt_page_stats('alumno_btree_puntuacion', 215);
El 215 es un bloque de nivel 2 y tiene 58 tuplas. El bloque 3 es del nivel 1 y tiene 210 tuplas y el bloque 2000 es de nivel 0 y tiene 10 tuplas.

Cuestión 18. 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 17.

El índice es secundaria y de campo no clave por lo que tenemos cajones de punteros a registro. Este arbol es denso por lo que hay un puntero por cada valor. El número de valores diferentes lo sacamos utilizando el siguiente comando. - select count(distinct puntuacion) from alumno. Vemos que hay 10.000 valores diferentes por el campo puntuación.
El bloque útil de los nodos hoja es el siguiente:
Butil=8192 bytes - 24(cabecera) - 4*n(matriz de identificadores 4 bytes*elemento) - 816 bytes (espacio libre)- 16 bytes (special) =7336 - 4*n
El bloque útil de los nodos intermedios es el siguiente:
Butil=8192 bytes - 24(cabecera) - 4*n(matriz de identificadores 4 bytes*elemento) - 1128 bytes (espacio libre) - 16 bytes (special)=7024 - 4*n
Los espacios libres los hemos buscado con el siguiente comando y buscando la diferencia entre upper y lower. - SELECT * FROM page_header(get_raw_page('alumno_btree_puntuacion', 2));
El nº de registros por nodo hoja es el siguiente:
nh*(Lcarnet + Lpbloque)+ Lpbloque <=7336 - 4*nh ---> nh*(4+12)+12<=7336 - 4nh ---> nh<=7317/24=304 bloques por nodo.
El nº de punteros a bloque por bloque del nodo raíz e intermedios son los siguientes:
n*Lploque + (n-1)*Lcarnet <= 7024- 4n ---> n*12+(n-1)*4<=7024 - 4n ---> n<=7028/= 292 punteros por bloque.
Según hemos calculado en teoría el número de cajones es de 10.000. Ahora vemos el nº de registros por cajón 14.000.000(registros)/10.000(valores diferentes) = 1400 reg por cajón. La longitud de puntero a registro es 12 igual que la de puntero a bloque. El factor de bloque del indice b+ de los cajones es el siguiente: 8192/12 = 682 reg/bloq.

Cuestión 19. Utilizando el procedimiento comentado en teoría para buscar sobre índices de tipo árbol visto, determinar el número de bloque leído de cada estructura hasta encontrar el registro de datos que tiene una puntuación de 456. Mostrar el código SQL utilizado

El coste según teoría sería el siguiente:
Índice arbol b+: 1(nodo raíz)+1(nodo intermedio 1)+1(nodo hoja (cajón de punteros a bloque))*3 (bloques) +1400(tuplas)= 1405 bloques.
Ahora reiniciamos las estadísticas para poder ver cuantos bloques se leen. - select pg_stat_reset(). A continuación, hacemos la búsqueda. - select * from alumno where puntuacion = '456'. Ahora miramos las estadísticas. - select * from pg_statio_user_indexes. Se han leído 0 bloques del índice ya que estaban en memoria, pero ha hecho hit de 4. Ahora vemos los bloques leídos fuera del índice. - select * from pg_statio_user_tables. Aquí se han hiteado 1411 bloques y leído 0 ya que estaban en memoria.

Cuestión 20. Crear un índice de tipo hash para el campo carnet. ¿Dónde se almacena físicamente ese índice? ¿Qué tamaño tiene? ¿Cuántos bloques tiene? ¿Cuántos cajones tiene? ¿Cuántas tuplas tiene de media un cajón? Indicar el procedimiento seguido e incluir el código SQL utilizado.

Primero creamos el índice. - create index alumno_hash_carnet on alumno using hash(carnet) ; Ahora para ver el tamaño de los bloques y su ubicación usamos el siguiente comando. - select relname, relpages, oid from pg_class where relname like('alumno_hash_carnet'); Vemos que tiene 62811 bloques (desbordamiento) debido a que hay más bloques que la suma del total de los cajones por bloque multiplicado por los bloques. Ahora para ver la ubicación, - select pg_relation_filepath ('alumno_hash_carnet ' '). La ubicación es base/42592/51086. Si nos vamos a este archivo y lo inspeccionamos vemos que ocupa 502.488 kbytes. (62.811 bloques * 8 kbytes). Para ver las estadísticas del índice usamos el siguiente comando. - select ntuples, maxbucket from hash_metapage_info(get_raw_page('alumno_hash_carnet',0)); Observamos que hay un total de 49151 cajones (maxbucket). Ahora para sacar las tuplas de media de un cajón dividimos el total de tuplas/total de cajones => 14000000 / 49151 = 285 tuplas por cajon de media.

Cuestión 21. Determinar el tamaño de bloques que teóricamente tendría de acuerdo con lo visto en teoría. Comparar los resultados obtenidos teóricamente con los resultados obtenidos en la cuestión 20.

El índice es primario y de campo clave.
Los hash son siempre densos nri = V(campo) = nr = 14.000.000 reg.
Número de cajones N = 49.151 cajones.
Cada cajón lleva de media nc = nri / N = 14.000.000 / 49.151 =285 reg
Lric = Lcarnet + Lpregistro = 4 + 12=16 reg
frc= 8192 /16 bloques = 512 reg / bloq (número de bloques=número cajones)
bc = 285/512= 1 bloq
El índice ocupa: Bri = N* Bc =49.151* 1= 49.151 blq
Hay más bloques que cajones en el apartado 20 porque debe haber desbordamiento de cajones.

Cuestión 22. Crear un índice de tipo hash para el campo puntuación. ¿Dónde se almacena físicamente ese índice? ¿Qué tamaño tiene? ¿Cuántos bloques tiene? ¿Cuántos cajones tiene? ¿Cuántas tuplas tiene de media un cajón? Indicar el procedimiento seguido e incluir el código SQL utilizado.

HOla
asd

Cuestión 23. Determinar el tamaño de bloques que teóricamente tendría de acuerdo con lo visto en teoría. Comparar los resultados obtenidos teóricamente con los resultados obtenidos en la cuestión 22.

Cuestión 24. ¿Qué conclusiones se puede obtener de la gestión y organización de PostgreSQL sobre los dos índices árbol y hash que se han creado y han sido analizados? ¿Por qué? Comparar con lo visto en teoría.

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. En este apartado se pretende mostrar cómo es el acceso a los datos para diferentes tipos de consultas.

PostgreSQL suministra varias vistas estadísticas que se puede usar para monitorizar los bloques leídos (tipo statio) de cada una de las estructuras creadas en la base de datos. En este apartado se deben usar esas vistas y está prohibido el uso de otro comando para este fin.

Para ello, borrar todas las tablas creadas y volver a crear la tabla **Alumno** como en la cuestión 12. Cargar los datos que se encuentran originalmente en el fichero datos_empleado.txt

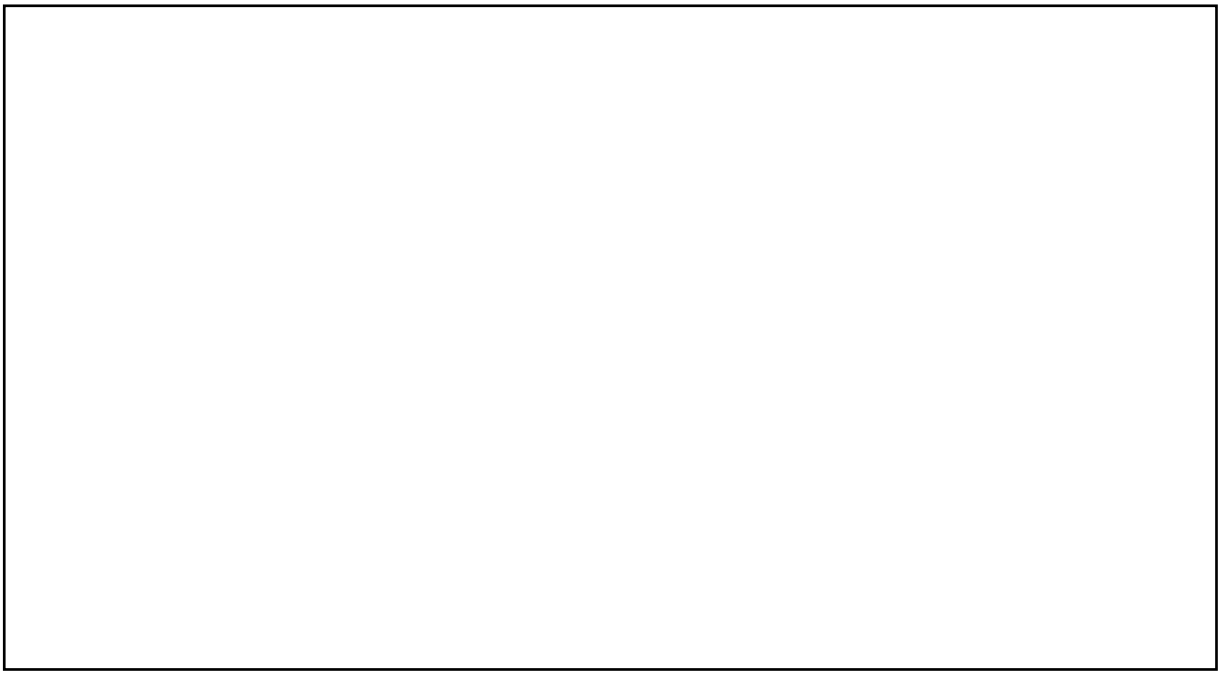
Cuestión 25. Crear un índice hash sobre el campo carnet y otro sobre puntuación. También crear un índice primario tipo árbol sobre el campo puntuación. ¿Cuál ha sido el proceso seguido para crear cada tipo de índice? Incluir el código SQL utilizado para ello.

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 de cada estructura? ¿Por qué? Importante, reinicializar los datos recolectados de la actividad de la base de datos antes de lanzar cada consulta. Se recuerda que solo se pueden usar vistas sobre las estadísticas de la base de datos.

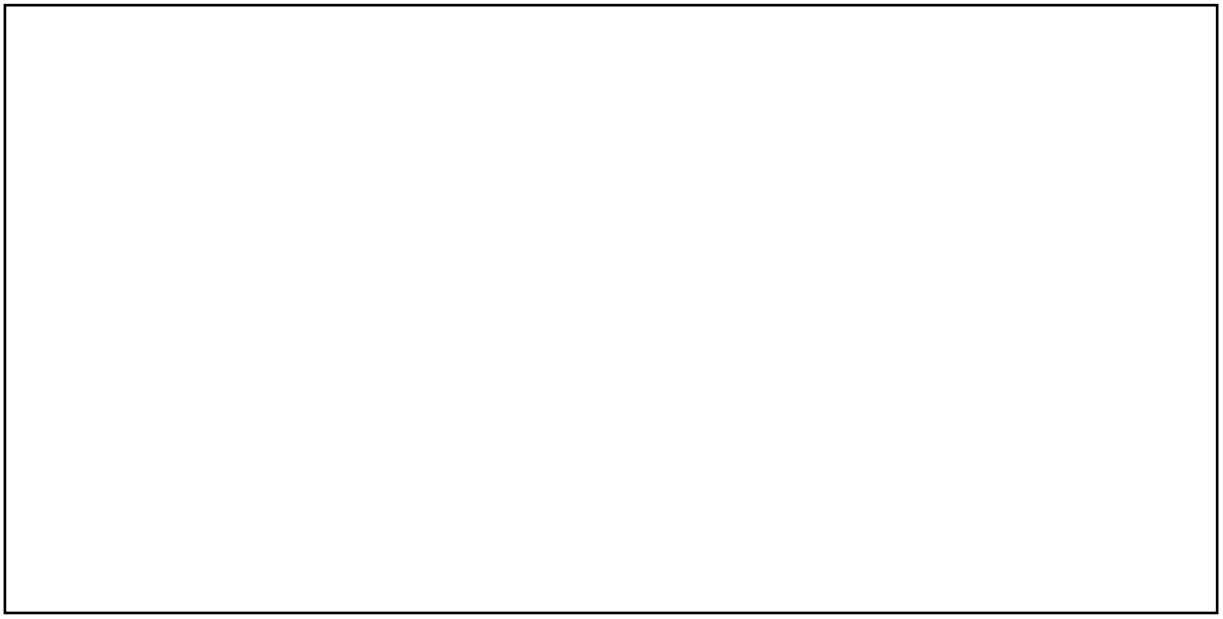
1. Mostar la información de las tuplas con carnet 23234566.

2. Mostar la información de las tuplas con puntuación de 3550.

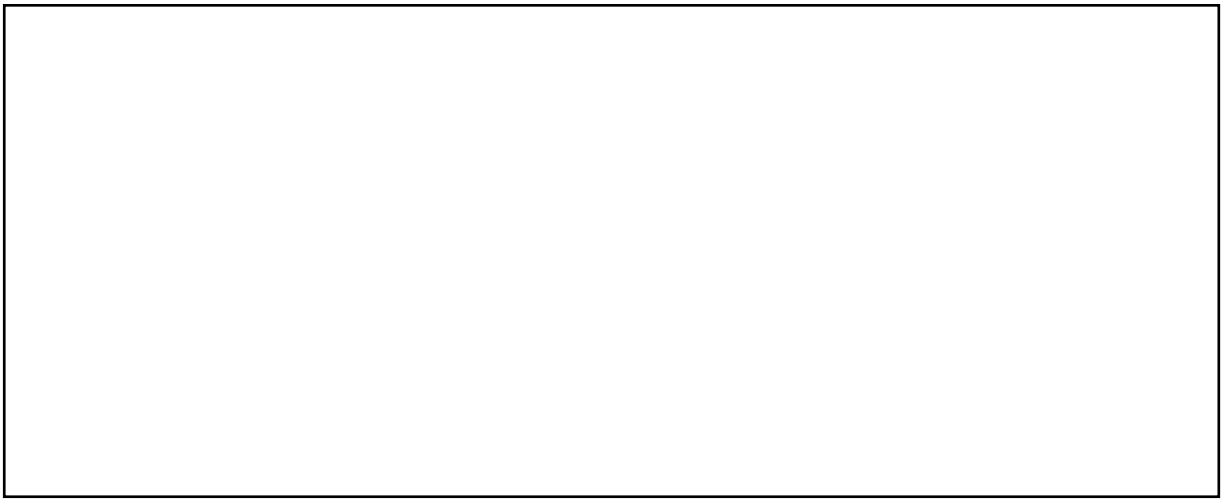
3. Contar el número de alumnos que tienen una puntuación de menos de 300.



4. Mostrar los alumnos que tienen en el nombre los caracteres 135678.



5. Actualizar el nombre del empleado que tiene el carnet 23567890



6. Mostar la información de los alumnos que tienen un carnet igual a 23567890 o 21557891

7. Insertar un nuevo alumno con código de carrera 20.

Cuestión 27. Borrar los índices creados sobre la tabla **alumno** y crear un índice multiclave tipo árbol sobre los campos carnet y puntuación. Incluir el código SQL utilizado para ello.

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 de cada estructura? ¿Por qué? Importante, reinicializar los datos recolectados de la actividad de la base de datos antes de lanzar cada consulta:

1. Mostrar los alumnos cuyo carnet es 13933295 o su nombre es alumno25

Primero reseteamos las estadísticas con el comando .-select pg_stat_reset();-. Después creamos la consulta select * from alumno3 where codigo_de_carrera>12

2. Mostrar las tuplas cuyo carnet es 19886862 y su puntuación es 50

3. Mostrar las tuplas cuyo carnet es 19886862 o su puntuación es 50

4. Mostrar los alumnos cuya puntuación es 60.

Cuestión 29. Crear la tabla **alumno3** como en la cuestión 10 pero sobre el campo código de carrera. 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 de cada estructura? ¿Por qué? Importante, reinicializar los datos recolectados de la actividad de la base de datos antes de lanzar cada consulta.

1. Mostrar las tuplas cuyo código de carrera es mayor que 12

Primero reseteamos las estadísticas con el comando .-select pg_stat_reset();-. Después creamos la consulta select * from alumno3 where codigo_de_carrera>12

2. Mostrar las tuplas cuyo código de carrera es 6 o 12

3. Mostrar las tuplas cuyo código de carrera es mayor que 10 y menor que 15.

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.

Bibliografía (PostgreSQL 14)

- Capítulo 1: Getting Started.
- Capítulo 5: 5.5 System Columns.
- Capítulo 5: 5.11 Table Partitioning.
- Capítulo 11: Indexes.
- Capítulo 20: Server Configuration.
- Capítulo 25: Routine Database Maintenance Tasks.
- Capítulo 28: Monitoring Database Activity. The statistics Collector
- Capítulo 29: Monitoring Disk Usage. Determining Disk Usage
- Capítulo VI.II: PostgresSQL Client Applications.
- Capítulo VI.III: PostgresSQL Server Applications.
- Capítulo 52: System Catalogs.
- Capítulo 64: B-Tree Indexes.
- Capítulo 70: Database Physical Storage.
- Apéndice F: Additional Supplied Modules. Pageinspect, pgstatutuple
- Apéndice G: Additional Supplied Programs. Oid2name