# Titulación: Grado en Ingeniería Informática, Ingeniería en

**Sistemas de Información e InfoAde** **Curso: 2024-2025. Convocatoria Ordinaria de Mayo** **Asignatura: Bases de Datos Avanzadas – Laboratorio**

Practica 1: Arquitectura PostgreSQL y

almacenamiento físico

**ALUMNO 1:**

## Nombre y Apellidos:

Álvaro Paniagua Cortijo

**DNI:**

06593675N

**ALUMNO 2:**

## Nombre y Apellidos:

Cristina Martínez Toledo

**DNI:**

09109126E

## Fecha:

04/02/2025

**Profesor Responsable:** Óscar Gutierrez

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 27 enero (GISI), 3 febrero, 10 febrero, 17 febrero, 24 de febrero y 3 marzo (GII/INFOADE)

Entrega de práctica: día 9 de marzo. Aula Virtual

Documento a entregar: un fichero con formato ZIP con las respuestas a las cuestiones planteadas en este fichero pdf (prohibido modificar el fichero o entregar otro documento diferente con respuestas y/o imágenes), así como los ficheros de log de postgresql relacionados con la resolución de la práctica y el fichero con el código de generación de los datos de la cuestión 1. 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 (17.2 la última). 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. Hay que resolver la práctica con consultas SQL.

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?

Se hicieron ajustes en el archivo de configuración postgresql.conf.

Primero, se activó el registro de sentencias DDL y DML configurando el parámetro log\_statement a “mod”.

Luego, para personalizar el prefijo de las líneas del log, se configuró el parámetro log\_line\_prefix de la siguiente manera: log\_line\_prefix = '%m [%p] 09109126E 06593675N %r '.

El %m servirá para mostrar el instante de tiempo en el que se genera el registro en el log, el %p para mostrar el id del proceso que lo generó, y el %r muestra el nombre del host y el puerto desde el que se ha realizado la conexión al servidor de base de datos.

## Organización de Archivos en PostgreSQL

Cuestión 1. Crear una nueva Base de Datos que se llame **PL1**. Después crear una tabla

**productos** con los siguientes campos:

* producto\_id: que debe ser el identificador del producto comenzando por 1.
* nombre: guarda el nombre del producto.
* stock: guarda la cantidad de cada producto en el almacén.
* precio: guarda el precio de cada producto.

Crear un programa que permita generar 25 millones de registros en un fichero de texto que pueda ser cargado en la tabla (preferiblemente en Python) con las siguientes propiedades para los siguientes campos, cuyos valores se deben generar aleatoriamente.

* stock: deben ser valores aleatorios entre 0 y 20000.
* precio: deben ser valores aleatorios entre 10 € y 5000 €.

Cargar los datos en la tabla y localizar los ficheros relacionados con la tabla. ¿cómo se localizan? ¿Cuánto ocupan? ¿por qué?

Para buscar el dio de nuestra tabla utilizaremos el comando:

SELECT oid, datname FROM pg\_database WHERE datname = 'pl1bbdda';

después, en C:\Program Files\PostgreSQL\16\data\base\oid\_de\_tabla encontraremos los archivos correspondientes a la tabla

La carpeta correspondiente tiene un tamaño del 1,9GB, al contar con muchos archivos de texto, donde algunos ocupan entre 50Mb o 10Mb, a excepcion de alguno que supera los 150MB

Cuestión 2. Calcular teóricamente el tamaño en bloques que ocupa la relación **productos** tal y como se realiza en clase de teoría. ¿Concuerda con el tamaño en bloques que nos proporciona PostgreSQL? ¿Cuál es el factor de bloque medio real de la tabla **productos**? ¿Por qué? Realizar una consulta SQL que obtenga ese valor y comparar con el factor de bloque teórico.

Tamaño de variables:

int = 4Bytes.

varchar(18) = caracteres ascii(1Byte por carácter). cabecera de registro = 23Bytes

Longitud de un registro = 4\*3+18+23= 53 Bytes

Con show block\_size, se obtiene una cantidad de 8192 Bytes de longitud de bloque.

El grado de ocupación es de 100% por defecto, por lo que se ocupará la longitud de bloque Como se usan 24 bytes de control por bloque, de los 8192 B , se usan 8168 B para datos (Butil). El factor de bloque será: 8168/53 (redondea hacia abajo) = 154 registros/bloque

El Nº de bloques totales será: 25M/154 (redondeo hacia arriba) = 162.338 bloques

Mediante consultas de PostgresSQL, se obtienen los siguientes datos:

Con esta consulta:

SELECT relnameAS table\_name, relpages AS bloques FROM pg\_class WHERE relname='productos';

se obtiene un total de 183872 bloques. Con la consulta:

SELECT 8192 / avg(pg\_column\_size(t)) AS block\_factor FROM productos t; se obtiene un factor de bloque de 146

El resultado de un factor de bloque menor que el teórico (y por tanto un número total de bloques distinto) podría indicar que PostgreSQL, en la práctica, utiliza menos espacio por registro en comparación con el cálculo teórico, probablemente debido a la eficiencia en el almacenamiento de punteros y el uso de almacenamiento adicional que optimiza el uso del espacio, por lo que el tamaño de bloque útil podría ser menor, o el uso por parte de PostgreSQL de más información de control en los registros como punteros u otros datos (como 4B de metadatos de cada tupla…).

Cuestión 3. Realizar una consulta que muestre los productos que tengan un precio de 3000 €. ¿Cuántas tuplas se obtienen y cuántos bloques se leen del disco por Postgres?

¿Por qué? Comparar con los resultados obtenidos al aplicar el método visto en teoría. (usar funciones de “ The Cumulative Statistics System”)

Como precio se trata de un campo no clave, el valor V(precio) = 4990, que son todos los valores distintos que puede tomar el campo. 5000 (máx)-10(min) = 4990.

Teniendo en cuanta la equiprobabilidad de todos los valores del campo precio. El número de registros será de 25.000.000/4990 = 5010 registro de media.

Mientras que como se trata de un campo no clave, el cual no está ordenado se deberán leer todos los 25.000.00 de registros para comprobar todos y cada uno de ellos de manera secuencial.

Para obtener los resultados, utilizaremos las siguientes consultas:

SELECT pg\_stat\_reset();

SELECT count(\*) FROM productos WHERE precio = 3000;

SELECT heap\_blks\_read, heap\_blks\_hit FROM pg\_statio\_user\_tables WHERE relname =’productos’;

Con el count, se obtiene un total de 4965 registros con precio de 3000, un valor ligeramente más bajo que el teórico, lo que se debe a la aleatoriedad de la creación del csv con los datos (no hay equiprobabilidad)

Al ser la primera vez que se lee la tabla, de todos los bloques se han leído:

183872 bloques en memoria (el total)

0 bloques de caché

Cuestión 4. Volver a realizar la consulta de la cuestión 3 de nuevo. ¿Cuántas tuplas se obtienen y cuántos bloques se leen del disco por Postgres ahora? ¿Por qué? Comparar con la cuestión anterior

Si se vuelve a ejecutar la misma consulta, se observará que como los datos no cambiaron, se obtiene el mismo número de registros con precio 3000, aunque esta vez, de todos los bloques se han leído: 183392 bloques en memoria

480 bloques de caché

Si sumas los bloques de memoria y caché, salen en total 183872 bloques leídos. Esto coincide con una búsqueda secuencial y con el número de bloques del apartado anterior de bloques leídos (de nuevo, debido a que la tabla sigue siendo igual, se leerán los mismos bloques)

Cuando se realizan consultas repetidas, PostgreSQL intenta optimizar el rendimiento al evitar lecturas redundantes desde el disco, prefiriendo acceder a los datos ya almacenados en el cache de memoria.

Por esto, cada vez que se ejecute la misma consulta generalmente aumentará el número de bloques leídos de caché, ya que PostgreSQL no necesitará leer los bloques desde el disco nuevamente.

Si se reinicia PostgreSQL y se ejecuta otra vez la consulta, el número de registros leídos desde caché volverá a 0 ya que se vacía el cache de memoria, lo que significa que todos los datos deben leerse nuevamente desde el disco

Cuestión 5. Crear una tabla **productos2** cuyas tuplas estén ordenadas físicamente por el campo precio de menor a mayor y que tenga la misma información. Cargar el mismo fichero de datos creado en la cuestión1. Indicar el proceso de generación de dicha tabla ordenada. ¿Cuántos bloques ocupa la tabla ahora? ¿Hay algún cambio? ¿Por qué?

Se ha creado la tabla productos2 y se han cargado los datos en ella ordenándolos de menor a mayor mediante las siguientes consultas:

create table if not exists productos2( id\_producto1 int,

nombre1 varchar(18), stock1 int,

precio1 int,

constraint ID1 primary key(id\_producto1)

);

insert into productos2(id\_producto1, nombre1, stock1, precio1) select \* from productos order by precio asc;

Al realizar una consulta como la de la cuestión 2, el número de bloques es de 183824, al ser la misma cantidad de información, pero con un orden distinto, por lo que hay bloques que contendrán alguna tupla más que otros, por lo que hay menos bloques en la tabla que en la original.

Cuestión 6. Repetir la cuestión 3 sobre la tabla **productos2** y comparar los resultados obtenidos indicando las conclusiones obtenidas. Relacionarlo con lo visto en teoría.

Como precio se trata de un campo no clave, el valor V(precio) = 4990, que son todos los valores distintos que puede tomar el campo. 5000 (máx)-10(min) = 4990.

Teniendo en cuanta la equiprobabilidad de todos los valores del campo precio. El número de registros será de 25.000.000/4990 = 5010 registro de media.

Mientras que como se trata de un campo no clave, el cual está ordenado, se puede hacer búsqueda secuencial o búsqueda binaria:

Secuencial: leer los 182824 bloques

Binaria: log2(182824)+ (nºregistros a recuperar/factor de bloque) = 18 + (5010/154)

BB= 51 bloques

Se realizan las siguientes consultas, como las de la cuestión 3:

SELECT pg\_stat\_reset();

SELECT count(precio1) FROM productos2 WHERE precio1 = 3000;

SELECT heap\_blks\_read, heap\_blks\_hit FROM pg\_statio\_user\_tables WHERE relname = 'productos2';

Con esto, se obtiene un total de 4965 registros con precio = 3000, la misma cantidad que en la cuestión 3 ya que los datos son los mismos.

Por lo mismo que sucedía en la cuestión 3, de todos los bloques se han leído:

183824 bloques en memoria

0 bloques de caché

Cuestión 7. Borrar 4000000 tuplas de la tabla **productos** de manera aleatoria usando el valor del campo producto\_id. ¿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.

PostgreSQL no elimina físicamente las filas de la tabla inmediatamente al ejecutar el comando DELETE. En lugar de eso, marca las filas como eliminadas y las deja en el almacenamiento. Estas tuplas eliminadas permanecen en el disco hasta que el proceso de vacuuming las elimina físicamente. Estas tuplas “muertas” son filas eliminadas que ya no son accesibles desde las consultas, pero no se han eliminado físicamente de la tabla. Esto se debe a que PostgreSQL utiliza un modelo de “multiversion concurrency control” (MVCC), donde mantiene versiones anteriores de las filas para proteger las transacciones.

Debido a esto, el tamaño de la tabla no varía y se mantienen los 183872 bloques de datos. Se han borrado gracias a este código:

DELETE FROM productos

WHERE id\_producto

IN (SELECT id\_producto FROM productos ORDER BY random() LIMIT 4000000);

Cuestión 8. Insertar un nuevo producto en la tabla **productos**. ¿dónde se inserta físicamente la nueva tupla? ¿Por qué? (Mirar apartado System Columns)

Mediante la consulta:

insert into productos(id\_producto,nombre,stock,precio)

values(25000001,'producto\_25000001',250,20);

insertamos el nuevo dato, el cual se inserta en el primer espacio libre de la tabla, el cual con la consulta:

SELECT ctid, \* FROM productos WHERE id\_producto = 25000001;

esta, nos indica que se encuentra en (183823,60), es decir, se ha colocado en el bloque número 183823 y en

la posición 60 dentro del bloque.

Esto podría deberse a que el espacio libre que dejó la tupla en la posición (183823,60), que había sido

borrada en el apartado anterior, ha sido utilizado por la nueva, ya que además el número del bloque es

menor al número de bloques totales que había en la tabla, por lo que se descarta que se haya creado un

bloque nuevo o que se haya añadido al final en el último bloque porque hubiera espacio libre desde un inicio.

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

La operación VACUUM puede servir para recuperar el espacio liberado por las tuplas eliminadas. También se puede usar VACUUM FULL. Esta operación no solo limpia las tuplas muertas, sino que también reorganiza físicamente los bloques de la tabla, reduciendo el tamaño de la tabla en disco, por lo

que se optimizará el rendimiento aún más. Por ello si se hace de nuevo la consulta:

SELECT relnameAS table\_name, relpages AS bloques FROM pg\_class WHERE relname='productos';

se obtiene un total de bloques 154412 que es menor que los 183872 anteriores.

Así, físicamente se han reorganizado y reescrito todas y cada una de las tuplas de la tabla. Con este método, el siguiente dato a insertar, se localiza al final de las tuplas anteriores.

Cuestión 10. Crear una nueva tabla denominada **productos3** con los mismos campos que la cuestión 1 y que esté particionada por el campo precio definido en rangos de 500

€. Insertar los datos del fichero de datos generado en la cuestión 1. 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.

Se crea la tabla como las dos tablas anteriores, pero se usa como PK (id\_producto,precio) y al final se

pone: PARTITION BY RANGE (precio3);

Para hacer las particiones se utiliza:

CREATE TABLE productos3\_10\_510 PARTITION OF productos3 FOR VALUES FROM (10) TO (510);

cambiando el campo valores desde 10 hasta 5001, porque el valor que esta después del TO es excluyente.

Por eso el último número es el 4510-5001.

Gracias a este proceso se crea una tabla principal, formada por 10 subtablas donde los valores se

comprenden dentro del rango establecido. La partición será útil, cuando haya una gran cantidad de

valores dispersos, porque el coste de buscar se reduce a buscar en las subtablas.

Estando formado por 10 particiones, y teniendo en cuenta la equiprobabilidad de los datos,

el nº reg/partición = 25.000.000/10 =2.500.000 de registros por partición.

Teniendo en cuenta que el factor de bloque = 154 calculado con anterioridad. El número de bloques por

partición será de 2.5M/154 = 16.234 bloques en cada partición.

Para calcular el n.º de bloques que usa PostgreSQL, se hace un VACUUM FULL, para reorganizar la tabla

y se hace la siguiente consulta:

SELECT relname AS table\_name, relpages AS bloques FROM pg\_class WHERE relname=' ';

donde en relname se pone el nombre de cada una de las particiones:

Nombre Subtabla N.º de Bloques Nombre Subtabla N.º de Bloques

Productos3\_10\_510 18470 Productos3\_510\_1010 18409

Productos3\_1010\_1510 18427 Productos3\_1510\_2010 18408

Productos3\_2010\_2510 18412 Productos3\_2510\_3010 18433

Productos3\_3010\_3510 18423 Productos3\_3510\_4010 18406

Productos3\_4010\_4510 18409 Productos3\_4510\_5011 18095

Cuestión 11. Repetir la cuestión 3 sobre la tabla **productos3** y comparar los resultados obtenidos con lo visto anteriormente en las tablas **productos** y **productos2** obteniendo conclusiones sobre el método de partición.

Realizando las mismas consultas:

SELECT pg\_stat\_reset();

SELECT count(precio3) FROM productos3 WHERE precio3 = 3000; se obtiene un total de 4965 registros con precio = 3000

Teniendo en cuenta que se han creado subtablas, hay que mirar cada una con sus respectivos nombres. Pero como el valor que estamos buscando es del rango [2510-3010), sabemos que ha buscado en la tabla

que forma este rango.

SELECT heap\_blks\_read, heap\_blks\_hit FROM pg\_statio\_user\_tables WHERE relname='productos3\_10\_510';

en todas estas consultas fuera de rango, el número de bloques leídos es 0 SELECT heap\_blks\_read, heap\_blks\_hit FROM pg\_statio\_user\_tables WHERE relname='productos3\_2510\_3010';

De todos los bloques se han leído: 18399 bloques en memoria

97 bloques de caché

## 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 **productos, productos2** y **productos3**. Crear una nueva tabla que se llama **productos** como en la cuestión 1 y que tenga cargados todos los datos del fichero de texto generado.

Se han borrado las tablas con “drop table nombre\_de\_la\_tabla;”

Se ha creado la tabla nueva con la siguiente consulta, igual que en la cuestión 1:

create table if not exists productos(

id\_producto int,

nombre varchar(18),

stock int,

precio int,

constraint ID primary key(id\_producto)

);

Cuestión 13. Crear un índice de tipo árbol para producto\_id. ¿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 con:

CREATE INDEX idb\_id\_producto ON productos(id\_producto);

Insertamos los datos:

\COPY  productos FROM 'PL1.csv' WITH (FORMAT csv, HEADER, DELIMITER E',', NULL 'NULL', ENCODING 'UTF-8');

Para saber dónde está almacenado físicamente, con:

SELECT pg\_relation\_filepath('idb\_id\_producto'); nos indica que en nuestro caso se encuentra en: base/70777/71693

Con: SELECT pg\_size\_pretty(pg\_relation\_size(‘idb\_id\_producto’));

obtenemos un tamaño de 536 MB

Con las consultas:

CREATE EXTENSION IF NOT EXISTS pgstattuple;

SELECT \* FROM pgstatindex('idb\_id\_producto');

obtenemos un total de 3 Niveles repartidos en:

Nivel 0 (raíz )= 1 Nivel 1 (intermedio) = 241 Nivel 2 (hojas) = 68306 bloques

En total, sumando los valores, tendremos 68548 bloques en el índice.

Además, obtenemos aproximadamente un 90% de densidad de hojas (90% del espacio en páginas hoja está ocupado por datos, 10% está libre).

Donde cada bloque de cada bloque de cada nivel tendrá unas 154 (factor de bloque) tuplas.

Cuestión 14. Determinar el tamaño de bloques y el número de niveles que teóricamente tendría de acuerdo con lo visto en teoría. Comparar los resultados obtenidos teóricamente con los resultados obtenidos en la cuestión 13.

Índice B+ sobre campo clave, secundario (no ordenado). Según los datos de la cuestión 2:

Bútil=8168 B. Longitud del campo de producto\_id (Lk)=4 B.

Además, en PostgreSQL, la longitud de puntero a bloque o a registro es de 4 Bytes.

n-> n\*Lpb+(n-1)\*Lk<=Butil -> 4n+4(n-1)<=8168 -> n<=(8168+4)/8 -> n= 1021 ptr

nh-> nh(Lk+Lpr)+Lpb<=Butil -> nh(4+4)+4<=8168 -> nh<= (8168-4)/8 -> nh=1020 reg

Para las hojas, como hay 25M de registros, Br=24510 (25M/1020)

Nvl intermedio 1: 24510/1021=25

Raíz: 1 (25/1021)

Teóricamente tendrá 3 niveles, y ocupará 24536 bloques. Estos valores son bastante menores que los obtenidos realmente. Esto quizás se puede deber a que PostgreSQL no llena completamente los bloques de los índices, o a que la “n” y “nh” reales sean menores, por lo que se necesitarían más bloques para guardar los datos. Además, PostgreSQL reserva espacio adicional en los índices para almacenar otros datos como páginas de metadatos (información sobre la estructura del índice) o páginas de relleno (espacio libre que facilita futuras inserciones sin tener que reequilibrar el árbol inmediatamente).

Cuestión 15. Crear un índice de tipo hash para el campo producto\_id. ¿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.

CREATE INDEX idh\_id\_producto ON productos using hash(id\_producto);

Con:

SELECT pg\_relation\_filepath('idh\_id\_producto'), lo localizamos en: base/70777/71696

Se obtiene con: SELECT pg\_size\_pretty(pg\_relation\_size('idh\_id\_producto')); un tamaño de 675 MB

Con:

CREATE EXTENSION IF NOT EXISTS pgstattuple;

SELECT \* FROM pgstathashindex('idh\_id\_producto ');

Nos da que hay 81920 cajones (bucket\_pages), con 4463 de overflow y casi un 29% de espacio libre

Con: SELECT relname, relpages FROM pg\_class WHERE relname = 'productos'; obtenemos que hay un total de 86385 bloques en el índice.

Teniendo en cuenta que son 25.000.000 de registros, y hay 81920 cajones, habrá 305 registros por cajón.

Cuestión 16. Determinar el tamaño de bloques y el número de cajones que teóricamente tendría de acuerdo con lo visto en teoría. Comparar los resultados obtenidos teóricamente con los resultados obtenidos en la cuestión 15.

Índice hash sobre campo clave, secundario (no ordenado). Según los datos de la cuestión 2:

Bútil=8168 B. Longitud del campo de producto\_id (Lk)=4 B.

LPB=LPR= 4 Bytes.

Según los datos reales obtenidos en la cuestión 15 (ya que no se da la función hash, solo podemos utilizar los datos obtenidos), hay 81920 cajones/particiones hash.

Longitud registro índice Lri= 4 (LPR)+4(Lk)=8B

Factor de bloque índice Fri=8168/8=1021 reg/bloque

Si hay 25M de registros, en cada cajón habrá 25M/81920=305 reg/cajón

Bri=305/1021=1 bloque/cajón

En total, teóricamente habría 81920 bloques (un bloque por cajón)

Hay una pequeña diferencia entre los datos teóricos y los reales, lo que puede deberse en parte a ese porcentaje de espacio libre y otros datos que PostgreSQL guarde en el índice.

Cuestión 17. Crear un índice de tipo árbol para el campo precio. ¿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.

CREATE INDEX idb\_id\_precio ON productos(precio);

Para saber dónde está almacenado físicamente, con:

SELECT pg\_relation\_filepath('idb\_id\_precio'); nos indica que se encuentra en:

base/70777/71797

Con: SELECT pg\_size\_pretty(pg\_relation\_size(‘idb\_id\_precio’)); obtenemos un tamaño de 176 MB

Con las consultas:

CREATE EXTENSION IF NOT EXISTS pgstattuple;

SELECT \* FROM pgstatindex('idb\_id\_precio');

obtenemos un total de 3 niveles repartidos en:

Nivel 0 (raíz )= 1 Nivel 1 (intermedio) = 106 Nivel 2 (hojas) = 21315 bloques

Con casi 89% de ocupación en hojas (11% libre). Sumando los valores de los bloques en las hojas, obtenemos 21422 bloques en total.

Donde cada bloque de cada bloque de cada nivel tendrá unas 154 (factor de bloque) tuplas.

CAJONES DE PTR???

Cuestión 18. Determinar el tamaño de bloques y número de niveles que teóricamente tendría de acuerdo con lo visto en teoría. Comparar los resultados obtenidos teóricamente con los resultados obtenidos en la cuestión 17.

Índice B+ sobre campo no clave, secundario (no ordenado): habrá cajones de punteros.

Bútil=8168 B. Longitud del campo de precio (Lk)=4 B. LPR=LPB= 4 Bytes.

n-> n\*Lpb+(n-1)\*Lk<=Butil -> 4n+4(n-1)<=8168 -> n<=(8168+4)/8 -> n= 1021 ptr

nh-> nh(Lk+Lpb)+Lpb<=Butil -> nh(4+4)+4<=8168 -> nh<= (8168-4)/8 -> nh=1020 reg

Como V(precio)=4991 (valores de 10 a 5000), habrá ese mismo número de cajones de punteros.

Para las hojas, como apuntan a los cajones, número de hojas=5 (4991/1020)

Raíz: 1 (5/1021)

Para los cajones de punteros:

Si hay 4991 cajones y 25M de registros, suponiendo la equiprobabilidad de los valores, en cada cajón habrá 5009 registros aproximadamente

Lc= LPR=4 B. Frc=1868/4= 467 reg/bloque. Br= 5009/467=11 bloques/cajón

En total el índice ocupará la suma de los bloques del B+ y de los cajones= (11\*4991)+(4991+1)=59892 bloques

Si lo de los cajones esta bien, por que no cuadra NADA y sale mas pequeño en postgres que aquí-?

Cajones de punteros???? (17 y 18)

Cuestión 19. Crear un índice de tipo hash para el campo precio. ¿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.

CREATE INDEX idh\_id\_precio ON productos using hash(precio);

Con:

SELECT pg\_relation\_filepath('idh\_id\_precio'); lo localizamos en base/70777/71798

Se obtiene con: SELECT pg\_size\_pretty(pg\_relation\_size('idh\_id\_precio')); un tamaño de 1106 MB

Con:

SELECT relpages FROM pg\_class WHERE relname = 'idh\_id\_precio';

obtenemos 141627 bloques en el índice.

Para poder ver el número de cajones tiene el índice de la tabla, se usa la consulta:

SELECT \* FROM pgstathashindex('idh\_id\_precio ');

Donde se obtienen 81920 cajones y 59704 de overflow, con casi un 57% de espacio libre.

Como hay cajones de punteros al ser un índice sobre campo no clave y secundario, las particiones hash “redireccionarán” el número de cajones de punteros que haya, que será V(precio)=4991, por lo que habrá 1 bloque por partición.

Cuestión 20. Determinar el tamaño de bloques y número de cajones que teóricamente tendría de acuerdo con lo visto en teoría. Comparar los resultados obtenidos teóricamente con los resultados obtenidos en la cuestión 17.

Índice hash sobre campo no clave, secundario (no ordenado): habrá cajones de punteros.

Bútil=8168 B. Longitud del campo de precio (Lk)=4 B. LPR=LPB= 4 Bytes.

Los cajones de punteros serán iguales que en la cuestión 18.

Índice Hash:

Longitud registro índice Lri= 4 (LPB)+4(Lk)=8B

Factor de bloque índice Fri=8168/8=1021 reg/bloque

Si hay 4991 cajones, en cada partición hash habrá 4991/81919=1 reg/partición

Bri=1/1021=1 bloque/partición

En total, teóricamente habría 81919 (un bloque por cajón de 1 reg cada uno)+(11\*4991)=136820 bloques

Hay más bloques reales que teóricos probablemente en parte debido a que por ese 57% de espacio libre se necesitan más bloques para almacenar los mismos datos.

Cuestión 21. ¿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.

Teniendo en cuenta que el índice que PostgreSQL tiene implementado como predeterminado es el índice de tipo árbol, que por lo visto en teoría en algunos casos tiene un coste menor que el índice Hash.

Aquí pese a que desconocemos el valor la función de asociación, sabemos que en total se crean 81920 cajones, que son bastantes, por lo que, pese a que reduce mucho la búsqueda al tener unos 305 registros por cajón, eleva el coste de almacenamiento de PostgreSQL.

Dentro de los incides hash, como no hay una función establecida y los valores del indice en la BBDD son aleatorios al haberlos creado con un random, los valores no se guardan de manera equiprovable. Se da el caso de que hay particiones que necesitan de espacio extra al establecido, overflow, originando que estas nuevas particiones extras no se llenen del todo, esto provoca que se creen huecos vacíos dentro de los indices al no tener información suficiente para llenarlo.

Hacer cálculos teóricos de cuánto costaría leer de cada cosa

Mientras tanto el árbol que en este caso solo tiene 2 niveles (raíz e intermedio), más los nodos hojas, permite un almacenaje más eficiente.

Por estas dos razones se entiende que para estas cuestiones el índice en formato árbol en estos ejercicios mejoran la búsqueda de datos para PostgreSQL.

ᕦ⊙෴⊙ᕤ

## 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 pueden usar para monitorizar los bloques leídos (tipo statio de la sección The Cumulative Statistics System) 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 (Table 28.2).

Para ello, borrar todas las tablas creadas y volver a crear la tabla **productos** como en la cuestión 1. Cargar los datos que se encuentran originalmente en el fichero generado en la cuestión 1.

Cuestión 22. Crear un índice primario tipo árbol sobre el campo precio. También crear un índice hash sobre el campo producto\_id y otro sobre precio. ¿Cuál ha sido el proceso seguido para crear cada tipo de índice? Incluir el código SQL utilizado para ello.

create table if not exists productos1(

id\_producto int,

nombre varchar(18),

stock int,

precio int,

constraint ID primary key(id\_producto)

);

CREATE INDEX idb\_precio ON productos1(precio);

CREATE INDEX idh\_producto\_id ON productos1 using hash(id\_producto);

CREATE INDEX idh\_precio ON productos1 using hash(precio);

Cuestión 23. 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? Comparar con lo visto en teoría. 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 (usar funciones de “The Cumulative Statistics System”).

1. Mostrar la información de las tuplas con 2000 €.

En todas las consultas se seguirá la misma estructura:

1º Con:

SELECT pg\_stat\_reset(); se reinicializaran los datos.

2º Con la consulta en cuestion: Select id\_producto, nombre, stock from productos1 where precio=2000;

Y 3º con:

SELECT i.indexrelname, i.idx\_scan, i.idx\_tup\_read, i.idx\_tup\_fetch, s.heap\_blks\_read, s.heap\_blks\_hit

FROM pg\_stat\_user\_indexes i JOIN pg\_statio\_user\_tables ON i.relname = s.relname

WHERE i.relname = 'productos1';

Obtendremos la información necesaria de las consultas:

índice que ha usado PostgreSQL para la consulta, veces usado, filas leídas y recuperadas y bloques leídos de memoria y cache.

Se ha utilizado el índice B+ con el campo precio, se han leído 5014 tuplas y se han leído 4947 bloques de memoria.

1. Mostrar la información de la tupla con producto\_id igual a 60000.

Consulta: select nombre, precio, stock from productos1 where id\_producto = 60000;

Se ha utilizado el índice hash sobre el campo id\_producto, se ha leído una sola tupla y se han leído un único bloque exclusivamente en la memoria caché, pese a realizar el SELECT pg\_stat\_reset(); varias veces

1. Mostrar los datos de los productos que tienen un producto\_id entre 180000 y

200000.

Con la consulta:

Select nombre, precio, stock from productos1 where id\_producto>=180000 and id\_producto<=200000;

Se obtiene que no se ha usado ningún índice establecido. El índice que ha usado hasta 3 veces (idx\_scan=3) PostgreSQL es el que crea por defecto, leyendo 20003 tuplas, 148 bloques de memoria y 2 bloques del caché. Este índice es un índice B+ sobre el campo id\_producto (ya que se definió como clave primaria)

1. Contar el número de productos que valen más de 4000 €.

Con la consulta: Select count(id\_producto) from productos1 where precio>4000;

no ha usado ningún índice, por lo que suponemos que ha hecho una lectura secuencial de todos los datos, lo que tiene sentido ya que se están leyendo una gran cantidad de datos, por lo que es menos costoso hacer una lectura secuencial que no varias lecturas de cualquiera de los dos índices.

Por lo tanto, no ha usado ningún índice establecido ni predeterminado. Ha leído 0 tuplas con el índice.

En total ha leído 180087 bloques de memoria y 3785 bloques de cache, en total :183872 bloques, que es el número total de bloques de la tabla como se observa en la pregunta 2. Con este valor se reafirma que PostgreSQL ha realizado una lectura secuencial de todos los 25M de registros de la tabla.

1. Mostrar el número total de unidades de los productos que tienen el mismo precio.

Para la consulta:

SELECT precio, sum(stock) AS total FROM productos1 GROUP BY precio;

PostgreSQL en total ha leído 179895 bloques de memoria y 3977 bloques de cache,

Como en la consulta anterior, no ha usado ningún índice en la lectura, sino que la hizo secuencial.

En total se han leído los 183872 bloques, que es el número total de bloques de la tabla como se observa en la pregunta 2 de este apartado. Con este valor se reafirma de nuevo que PostgreSQL ha realizado para esta consulta una lectura secuencial en los 25M de registros de la tabla, ya que al tener que recuperar un número muy elevado de registros, es menos costoso hacerlo secuencialmente.

1. Insertar un nuevo producto en la tabla productos con un precio de 1000 €.

Para asegurarnos de que la PK no coincide con otra, vamos a usar el id\_producto 25.000.001 y valores stock 250 y precio 1000 con:

insert into productos1(id\_producto,nombre,stock,precio)

values(25000001,'producto\_25000001',250,1000);

Se han leído 2 bloques de memoria cache para insertar el nuevo producto, el cual se localiza mediante la consulta:

SELECT ctid, \* FROM productos1 WHERE id\_producto = 25000001;

(183823,61)

En el bloque 183823 en la posición 61, que es el siguiente espacio libre en la tabla.

1. Actualizar el precio del producto insertado anteriormente para cambiar de 1000

€ a 2000 €.

Para:

UPDATE productos1 SET precio = 2000 WHERE id\_producto = 25000001;

Se ha modificado el producto insertado en el apartado anterior, y se ha localizado usando el índice hash 1 vez sobre el campo id\_producto, leyendo en total 4 bloques de caché.

Cuestión 24. Borrar los índices creados en la cuestión 20. Crear un índice multiclave tipo árbol sobre los campos stock y precio. Incluir el código SQL utilizado para ello.

Los índices se borraron con consultas del tipo: ”DROP INDEX nombre\_indice;”

Índice creado con la consulta:

Create index idmb\_precio\_stock on productos1 using btree(precio,stock);

donde se especifica que es un índice tipo árbol sobre los campos precio y stock

Cuestión 25. 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 productos que tienen un stock de 100 unidades y un precio de 1000

€.

Como en la cuestión 23.1 se va a seguir la misma estructura de consultas:

1. SELECT pg\_stat\_reset(); se reinicializaran los datos.
2. La consulta en cuestión: select id\_producto, nombre from productos1 where stock=100 and precio=1000;
3. SELECT i.indexrelname, i.idx\_scan, i.idx\_tup\_read, i.idx\_tup\_fetch, s.heap\_blks\_read, s.heap\_blks\_hit FROM pg\_stat\_user\_indexes i JOIN pg\_statio\_user\_tables s ON i.relname = s.relname WHERE i.relname = 'productos1';

Dada a la aleatoriedad de los datos, se ha dado el caso en el que ningún solo producto tenga los valores indicados. Por eso, pese a que utiliza el índice creado, no lee ningún solo bloque ni de memoria ni de caché.

1. Mostrar los productos que tienen un stock de 100 unidades o un precio de 1000

€.

Con la consulta:

select id\_producto, nombre from productos1 where stock=100 or precio=1000;

Al realizar esta consulta PostgreSQL, nos indica que no a seguido el indice creado o el propio, y que en total se han leído 179606 bloques de memoria y 4266 de caché, aun haciendo SELECT pg\_stat\_reset();

Que nos da un total de 183872 bloques leídos, por lo que se puede entender que ha realizado una búsqueda secuencial para leer y comprobar todos los registros a las condiciones dadas.

1. Mostrar los productos que tienen un stock de 100 unidades.

Con la consulta:

select id\_producto, nombre from productos1 where stock=100;

Al realizar esta consulta PostgreSQL, nos indica que no a seguido el índice creado o el propio, y que en total se han leído 179510 bloques de memoria y 4362 de caché, aun haciendo SELECT pg\_stat\_reset();

Que nos da un total de 183872 bloques leídos, por lo que se puede entender que ha realizado una búsqueda secuencial para leer y comprobar todos los registros a las condiciones dadas.

1. Mostrar los productos que tienen un precio de 1000 €.

Con la consulta:

select id\_producto, nombre from productos1 where precio=1000;

Aquí se observa un cambio con las consultas anteriores. Ha usado el índice creado al inicio del ejercicio 1 vez, leyendo 4926 tuplas con ese índice. Por lo que ya no ha realizado una búsqueda secuencial.

En total se han leído 4721 bloques de memoria y 124 bloques del caché aún haciendo varias veces el SELECT pg\_stat\_reset();

Con esto se entiende que dependiendo del campo en cuestión en este caso (el valor stock) como el rango de valores es tan alto (20001 valores distintos) PostgreSQL prefiere hacer una búsqueda secuencial sobre ese campo.

Cuestión 26. Crear la tabla **productos3** particionada por medio de una función de asociación h = stock mod 10. 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? Comparar con la teoría. Importante, reinicializar los datos recolectados de la actividad de la base de datos antes de lanzar cada consulta.

1. Mostrar el número de productos con un stock de 200.

Al hacer la siguiente consulta (después de SELECT pg\_stat\_reset(); ):

SELECT COUNT(\*) FROM productos3 WHERE stock= 200;

SELECT relname AS nombre\_tabla, heap\_blks\_read, heap\_blks\_hit FROM pg\_statio\_user\_tables

WHERE relname LIKE 'productos3%';

Da que se han leído 1227 registros de 18820 bloques en la partición 1.

Se puede observar que en lugar de guardarse en la partición 0, se ha guardado en la 1.

1. Mostrar los productos que tienen un stock de 100 o 200 o 300.

Con el mismo primer y último “select” de la cuestión 26.1, y con esta consulta entre medias:

SELECT count(\*) FROM productos3 WHERE stock IN (100, 200, 300);

Se obtiene que se leen 3632 registros de las siguientes particiones:

productos3\_p0 : 18432 bloques leídos. productos3\_p1 : 18880 bloques. productos3\_p9 : 18688 bloques

En total, se leen 56000 bloques.

El archivo organizado por función de asociación teóricamente tendría 10 cajones con (si asumimos equiprobabilidad) 2.5M de registros en cada uno. Si Fr es 154 reg/bloque , br=2.5M/154= 16234 bloques/cajón, un valor algo menor a lo obtenido realmente tanto en esta cuestión como en la 26.1 (alrededor de 18400), ya que quizás se guardan otros datos o PostgreSQL no llena del todo los bloques.

Teóricamente además debería leer solo de la partición p0, pero lee de otras. También, los valores de stock van de 0 a 20000 (V(stock)=20001), por lo que si supusiéramos equiprobabilidad habría 25M/20001= 1250 registros por cada valor, bastante similar a los resultados reales.

1. Mostrar los productos con un stock de más de 100 y menos de 200.

Ahora, la consulta “intermedia” será:

SELECT count(\*)  FROM productos3 WHERE stock > 100 AND stock < 200;

Darían 123038 registros leídos (aproximadamente 100\*1250), con en total 184064 (aproximadamente 18400\*10, lo que sería un coste aproximado de leer todos los bloques de todas las particiones) bloques leídos de las 10 particiones. Teóricamente se leerían 99 valores, por lo que serían 1250\*99=123750 registros leídos de 10 particiones leyendo en total 16234\*10=162340 bloques (todas las particiones con todos sus bloques)

Cuestión 27. 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.

En este apartado se han comprobado distintas estrategias de lectura y uso de índices:

PostgreSQL utiliza diferentes estrategias de acceso a los datos dependiendo de la consulta realizada y la existencia de índices adecuados. Se ha comprobado que, en consultas con filtros sobre campos indexados, PostgreSQL prefiere utilizar estos índices en lugar de realizar un escaneo secuencial. Sin embargo, en algunos casos, cuando la cardinalidad del campo es alta o la consulta requiere recorrer una gran cantidad de registros, se observa que PostgreSQL opta por una búsqueda secuencial, ignorando los índices, para optimizar el acceso.

En la cuestión 26, vimos que a pesar de la función módulo sobre la que se creó, los valores se guardaban en particiones distintas a lo esperado. A priori podríamos pensar que el valor es incorrecto, sin embargo, se debe a que PostgreSQL antes de hacer la función módulo le aplica una función interna “hashint4”, convirtiendo el número en un valor hash pseudoaleatorio, y el módulo 10 lo hace con el valor resultante. Lo hace para evitar particiones descompensadas si los datos no están distribuidos uniformemente , ya que si una partición tiene muchos más datos que otras, las consultas que acceden a esa partición serán más lentas porque PostgreSQL tendrá que leer más bloques de disco.

# Bibliografía (PostgreSQL 16)

* + Capítulo 1: Getting Started.
  + Capítulo 5: 5.6 System Columns.
  + Capítulo 5: 5.12 Table Partitioning.
  + Capítulo 11: Indexes.
  + Capítulo 19: Server Configuration.
  + Capítulo 24: Routine Database Maintenance Tasks.
  + Capítulo 27: Monitoring Database Activity. The statistics Collector
  + Capítulo 27.6: Monitoring Disk Usage. Determining Disk Usage
  + Capítulo 51: System Catalogs
  + Capítulo 64.1: B-Tree Indexes
  + Capítulo 64.6: Hash Indexes
  + Capítulo 65: Database Physical Storage
  + Capítulo VI.II: PostgresSQL Client Applications.
  + Capítulo VI.III: PostgresSQL Server Applications.
  + Apéndice F: Additional Supplied Modules. Pageinspect, pgstatutuple
  + Apéndice G: Additional Supplied Programs. Oid2name