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: _____

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?

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.

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= 51bloques

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.

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

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 relname AS 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.

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.

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.

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.

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.

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.

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 Nivel2 (hojas) = 21315 bloques
Con casi 89% de ocupación en hojas (11% libre). Sumando los valores de los bloques, obtenemos 21422 bloques en total, mismo valor que da si ejecutamos la consulta SELECT relname, relpages FROM pg_class WHERE relname = 'idb_id_precio';. Cada bloque de cada nivel tendrá unas 154 (factor de bloque) tuplas.

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.

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.

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.

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.

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.

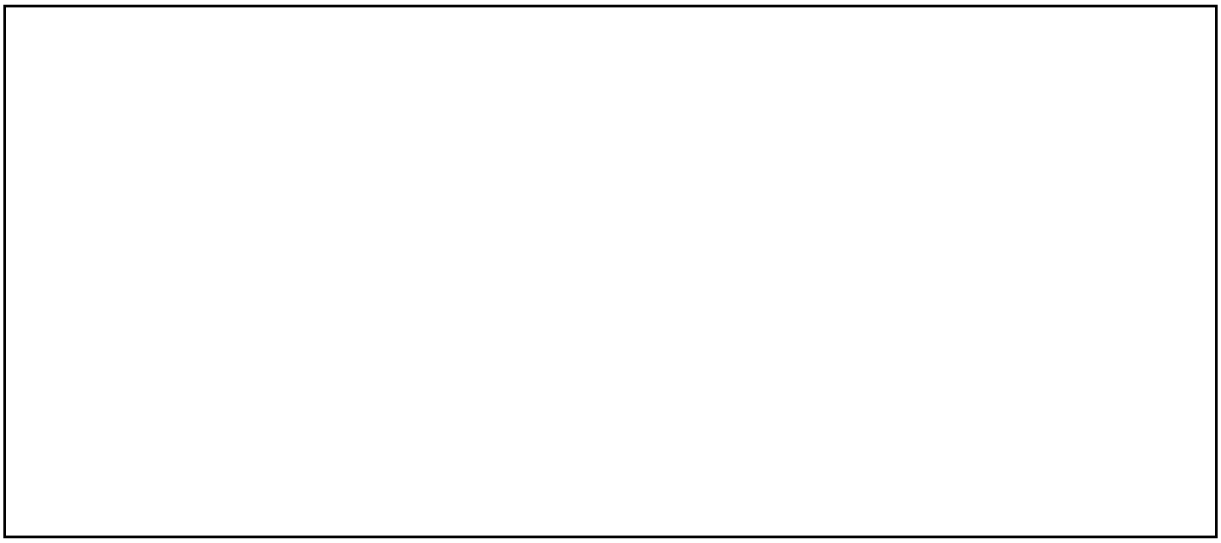
Cuestión 22. Crear un índice primario tipo árbol sobre el campo precio. También crear un índice hash sobre el campo producto_id y otro sobre precio. ¿Cuál ha sido el proceso seguido para crear cada tipo de índice? Incluir el código SQL utilizado para ello.

```
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);
```

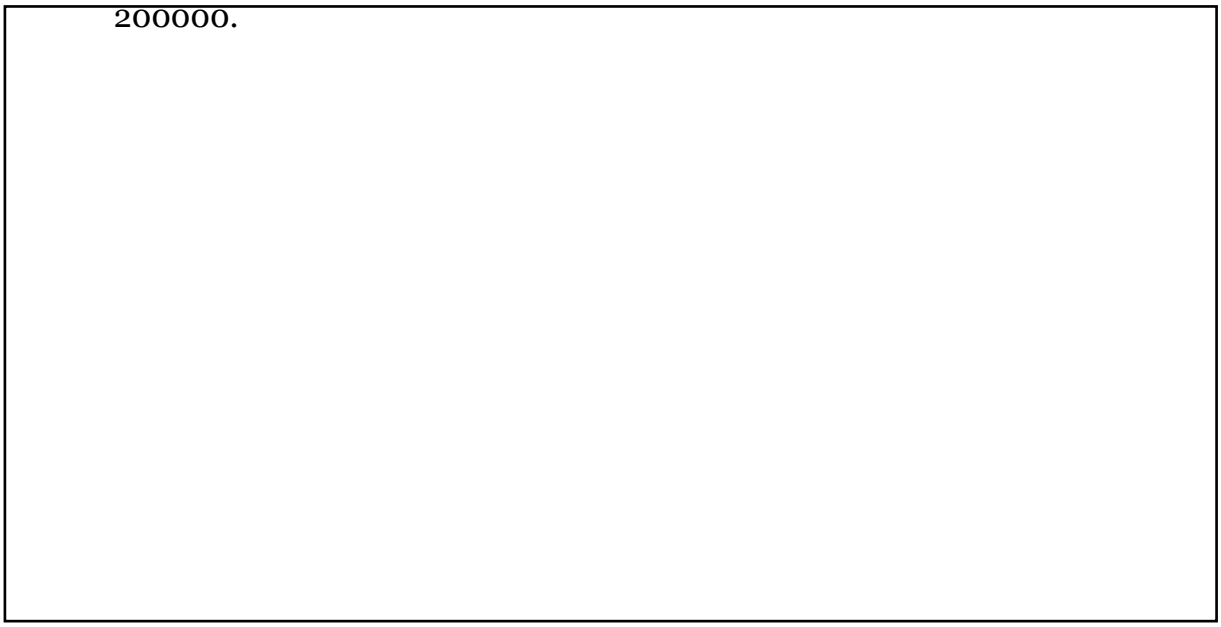
Cuestión 23. Para cada una de las consultas que se muestran a continuación, ¿Qué información se puede obtener de los datos monitorizados por la base de datos al realizar la consulta? Comentar cómo se ha realizado la resolución de la consulta. ¿Cuántos bloques se han leído de cada estructura? ¿Por qué? 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 €.

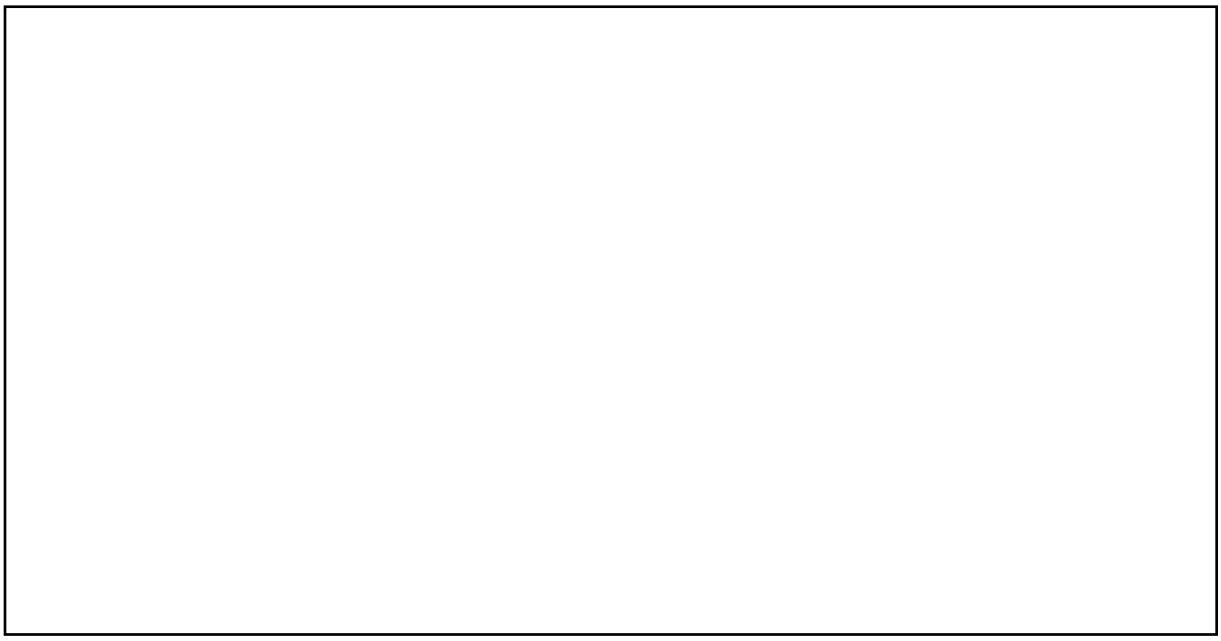
2. Mostrar la información de la tupla con producto_id igual a 60000.



3. Mostrar los datos de los productos que tienen un producto_id entre 180000 y 200000.



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



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

6. 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.

7. Actualizar el precio del producto insertado anteriormente para cambiar de 1000 € a 2000 €.

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.

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 €.

2. 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.

3. 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 indice creado o el propio, y que en total se han leido 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.

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

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.

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

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

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.

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