PRÁCTICA 2

Sistemas de Gestión de Bases de Datos

12 NOVIEMBRE

PRÁCTICA 2 - EFL
Mario Ventura Burgos 43223476-J
Grado en Ingeniería Informática (GIN 3)
CURSO 2023-2024

ÍNDICE

1. PREPARACIÓN DE LA BASE DE DATOS.

En esta práctica se pide explícitamente que la base de datos tenga un mínimo de 100 registros por tabla. Para crearlos, se ha desarrollado un algoritmo en java que permite la creación de una cantidad determinada de INSERTS.

A continuación, se mostrará el código que se ha usado para crear estos INSERTS y también se adjuntará en la entrega un documento .txt en el cual se encuentran todos los INSERTS obtenidos por el algoritmo e introducidos en la base de datos (el algoritmo es relativamente simple, por tanto, no se comentará nada al respecto. Únicamente hay que añadir que, en nuestro caso, hemos optado por hacer 1500 registros por cada tabla, pero podrían hacerse tantos como se quieran).

ALGORITMO:

```
public class autoInserts {
    // ---> ATRIBUTOS Y CONSTANTES
    private static final int NUM_INSERTS = 1500; // Numero de iteraciones que se haran
    private static final int NUM_INICIO = 1;    // Numero del cual se parte en las
iteraciones
    private static int randomInteger;           // Para generar aleatorios
    private static final int MAX_ALEATORIO = 100;   // Numero maximo aleatorio


    // ---> METODOS
    // Metodo main
    public static void main (String[] args) {
        // Llamadas a los metodos de generacion de inserts
        generate_inserts_aliment();
        System.out.println("");
        generate_inserts_proveidor();
        System.out.println("");
        generate_inserts_albara();
        System.out.println("");
        generate_inserts_linia_albara();
        System.out.println("");
        generate_inserts_venda();
    }

    // Generacion de inserts para aliment
    public static void generate_inserts_aliment() {
        System.out.println("INSERT INTO ALIMENT (REFERENCIA, NOM, PREU) VALUES");
        Random r = new Random();
        int i;

        // Generacion de los inserts
        for(i = NUM_INICIO; i<NUM_INSERTS; i++) {
```

```
            // Precio aleatorio
            randomInteger = r.nextInt(MAX_ALEATORIO)+1;
            // Impresion del insert
            System.out.println("('REF"+i+"', 'ALI"+i+"', "+randomInteger+"),");
        }
        // Precio aleatorio
        randomInteger = r.nextInt(MAX_ALEATORIO)+1;
        // Impresion del insert
        System.out.println("('REF"+i+"', 'ALI"+i+"', "+randomInteger+");");
    }

    // Para proveidor
    public static void generate_inserts_proveidor() {
        System.out.println("INSERT INTO PROVEIDOR (NIF, NOM) VALUES");
        int i;

        // Generacion de los inserts
        for(i = NUM_INICIO; i<NUM_INSERTS; i++) {
            // Impresion del insert
            System.out.println("('NIF"+i+"', 'Proveedor"+i+"'),");
        }
        // Impresion del insert
        System.out.println("('NIF"+i+"', 'Proveedor"+i+"');");
    }

    // Para albara
    public static void generate_inserts_albara() {
        System.out.println("INSERT INTO ALBARA (CODI, NIF_PRO, DATA, FACTURAT) VAL-
UES");
        Random r = new Random();
        int i,randomDay,randomMonth;
        String facturat;

        // Generacion de los inserts
        for(i = NUM_INICIO; i<NUM_INSERTS; i++) {
            // random date
            randomDay = r.nextInt(28)+1;
            randomMonth = r.nextInt(12)+1;
            //facturat
            randomInteger = r.nextInt(100);
            if (randomInteger%2 == 0) {
                facturat = "S";
            } else {
                facturat = "N";
            }
            // Impresion del insert
            System.out.println("("+i+", 'NIF"+i+"', '2023-"+randomMonth+"-"+ran-
domDay+"', '"+facturat+"'),");
        }
```

```
        // Precio aleatorio
        randomDay = r.nextInt(28)+1;
        randomMonth = r.nextInt(12)+1;
        //facturat
        randomInteger = r.nextInt(100);
        if (randomInteger%2 == 0) {
            facturat = "S";
        } else {
            facturat = "N";
        }
        // Impresion del insert
        System.out.println("("+i+", 'NIF"+i+"', '2023-"+randomMonth+"-"+randomDay+"',
'"+facturat+"');");
    }

    // Para linia_albara
    public static void generate_inserts_linia_albara() {
        System.out.println("INSERT INTO LINIA_ALBARA (CODI_ALB, REFERENCIA, QUILO-
GRAMS, PREU) VALUES");
        Random r = new Random();
        int i,randomKG,randomPrice;

        for(i = NUM_INICIO; i<NUM_INSERTS; i++) {
            // Random KG
            randomKG = r.nextInt(100)+1;
            // Random price
            randomPrice = r.nextInt(50)+1;
            // Show insert
            System.out.println("("+i+", 'REF"+i+"', "+randomKG+", "+randomPrice+"),");
        }
        // Random KG
        randomKG = r.nextInt(100)+1;
        // Random price
        randomPrice = r.nextInt(50)+1;
        // Show insert
        System.out.println("(" + i + ", 'REF" + i + "', " + randomKG + ", " + random-
Price + ");");
    }

    // Para venda
    public static void generate_inserts_venda() {
        System.out.println("INSERT INTO VENDA (CODI, REFERENCIA, DATA, QUILOGRAMS,
PREU) VALUES");
        Random r = new Random();
        int i,randomDay,randomMonth,randomKG,randomPrice;

        // Generacion de los inserts
        for(i = NUM_INICIO; i<NUM_INSERTS; i++) {
            // Random date
```

```
            randomDay = r.nextInt(28)+1;
            randomMonth = r.nextInt(12)+1;
            // Random KG
            randomKG = r.nextInt(100)+1;
            // Random price
            randomPrice = r.nextInt(50)+1;
            // Impresion del insert
            System.out.println("("+i+", 'REF"+i+"', '2023-"+randomMonth+"-"+ran-
domDay+"', "+randomKG+", "+randomPrice+"),");
        }

        // Random date
        randomDay = r.nextInt(28) + 1;
        randomMonth = r.nextInt(12) + 1;
        // Random KG
        randomKG = r.nextInt(100) + 1;
        // Random price
        randomPrice = r.nextInt(50) + 1;
        // Impresion del insert
        System.out.println("(" + i + ", 'REF" + i + "', '2023-" + randomMonth + "-" +
randomDay + "', " + randomKG + ", " + randomPrice + ");");
    }
}
```

Ahora que la base de datos está preparada y cuenta con las condiciones establecidas, empecemos a realizar los ejercicios.

2. MOSTRAR EL TAMAÑO DEL BLOQUE.

En este primer apartado se pide mostrar el tamaño del bloque. Para ello, en PostgreSQL, se ejecuta el siguiente comando:

```
-- 1. TAMAÑO BLOQUE
SELECT current_setting('block_size') AS tamano_bloque;
```

Que nos permite obtener el siguiente resultado:

```
postgres=# \c practica2;
Ahora está conectado a la base de datos «practica2» con el usuario «postgres».
practica2=# -- 1. TAMAÑO BLOQUE
practica2=# SELECT current_setting('block_size') AS tamano_bloque;
 tamano_bloque
---------------
 8192
(1 fila)
```

El tamaño del bloque es de vital importancia para el almacenamiento y rendimiento de la base de datos. En nuestro caso se puede ver que el tamaño es de 8192, que viene dado en bytes. Esto indica que el tamaño es, por tanto, de 8KB, que es el tamaño predeterminado en PostgreSQL. Sin embargo, pese a que este tamaño suele ser correcto en la mayoría de los casos, también es de vital importancia conocer nuestro gestor de base de datos para poder cambiar este tamaño, ya que en algunos casos 8KB no es lo óptimo.

3. ANALIZAR LA TABLA ALIMENT Y EXTRAER LA CANTIDAD DE REGISTROS QUE CABEN EN CADA BLOQUE.

Procedemos ahora a analizar la tabla aliment para conocer cuántos registros caben dentro de cada bloque. Recordemos que cada tabla tiene 1500 registros y que cada bloque tiene el tamaño exacto de 8192 bytes.

Para saber cuántos registros caben en cada bloque, lo que haremos será ver en qué bloque se guardan los 1500 registros de la tabla aliment en nuestra base de datos.
Para saber en qué bloque se guarda cada uno de los registros se ejecutará la siguiente consulta:

```
-- 2. REGISTROS POR CADA BLOQUE EN ALIMENT
SELECT (ctid::text::point)[0]::bigint AS numero_bloque, count(*) AS numero_registros
FROM aliment
GROUP BY numero_bloque
ORDER BY numero_bloque;
```

Donde ***(ctid::text::point)*** representa la ubicación física de una fila en el disco, que se convertirá a formato texto y luego a point (representación de un punto en un espacio de 2 dimensiones). Por otro lado, el ***[0]*** representa el primer elemento de un array, ya que esto se tratará como tal y la consulta selecciona el primer elemento de esa lista o array. El resultado que se obtenga se pasa a ***bigint*** y recibe el nombre de numero_bloque, que evidentemente, representa el numero de bloque del disco en el que se almacena el registro.

Al ejecutar esta consulta en SQL Shell, obtenemos el siguiente resultado:

```
practica2=# -- 2. REGISTROS POR CADA BLOQUE
practica2=# SELECT (ctid::text::point)[0]::bigint AS numero_bloque, count(*) AS numero_registros
practica2-# FROM aliment
practica2-# GROUP BY numero_bloque
practica2-# ORDER BY numero_bloque;
 numero_bloque | numero_registros
---------------+------------------
             0 |              172
             1 |              157
             2 |              157
             3 |              157
             4 |              157
             5 |              157
             6 |              157
             7 |              157
             8 |              157
             9 |               72
(10 filas)
```

(La agrupación (GROUP BY) en la consulta es de vital importancia dada la cantidad de registros en la base de datos, pero la ordenación (ORDER BY) realmente no lo es. Se ha optado por añadirla ya que en un primer momento se hizo sin ordenación y, pese a que el resultado era el mismo, éste se mostraba con una carencia total de orden. De esta forma es más legible el resultado).

Como se puede ver, si sumamos todos los valores obtenidos en la columna numero_registros para cada numero_bloque, el total es de 1500, de forma que todos los registros de la tabla aliment han sido contemplados en la consulta. Además, sabemos en qué bloque se guardan y qué cantidad de registros tiene cada bloque.

Se puede ver, sin embargo, que la cantidad de registros para cada bloque no es siempre la misma. Para responder a la pregunta de "¿Cuántos registros caben en cada bloque?", debemos plantear alguna idea que nos permita por lo menos obtener una cifra aproximada. En nuestro caso, optaremos por hacer una media aritmética. Esto se hace porque en 8 de los 10 bloques la cantidad de registros es la misma: 157 registros. Por tanto, obtener una media aritmética (que saldrá cercana a ese valor) nos permitirá obtener una cifra orientativa con un margen de error ínfimo, debido a que el 80% de los bloques tienen exactamente la misma cantidad de registros. Para calcular esta media hacemos lo siguiente:

$$\frac{172 + 157 + 157 + 157 + 157 + 157 + 157 + 157 + 157 + 72}{10} = \frac{172 + 72 + 8 \cdot (157)}{10} = \mathbf{150}$$

El resultado es bastante evidente: Si hay 1500 registros divididos en 10 bloques, eso supone una media de **150 registros por bloque.**

Tal y como se mencionó con anterioridad, el valor que se esperaba obtener (150) nos permitiría obtener una respuesta a la pregunta de cuántos registros caben por bloque, sin que este resultado presentase una gran diferencia con lo observado (157 registros en el 80% de los bloques).

Como respuesta final, se podría concluir que en un bloque de 8192 bytes en la tabla aliment, caben 150 registros (importante recordad que es un cálculo aproximado).

4. CREAR UN NUEVO TABLESPACE LLAMADO tb1.

Para crear un tablespace es necesario gozar de permisos ya que, de lo contrario, PostgreSQL no nos dejará crearlo.

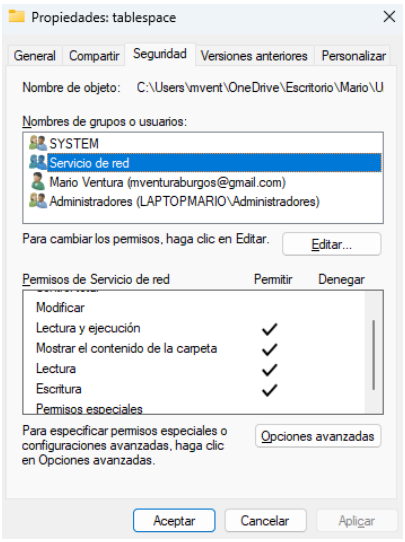
Dado que estamos usando Windows como sistema operativo, la carpeta en la que creemos el tablespace deberá tener permisos para ello, pero también deberá tenerlos la carpeta donde se encuentre esta carpeta. Es decir, debemos asegurarnos de que todas las carpeta que forman la ruta de acceso a la carpeta donde se creará el tablespace tienen permisos para ello.

Hay dos formas de hacerlo:

- Navegando con el sistema operativo y dando permisos de una forma más intuitiva
- Usando comandos ejecutados como administrados en psql.

Nosotros hemos optado por la primera opción.

En la carpeta donde se guarda todo el contenido de esta práctica, se ha creado una carpeta llamada tablespace, que es donde se planea guardar la tablespace tb1.



Una vez hecho esto, hacemos click derecho sobre esta carpeta, seleccionamos propiedades, vamos al apartado de seguridad y seleccionamos editar.

Estaremos viendo, entonces, todos los usuarios que tienen acceso a esa carpeta y con qué permisos puede actuar sobre esta. Crearemos un nuevo usuario al que llamaremos “Servicio de red” y le daremos los permisos que aparecen en la imagen.

Ahora seleccionamos aplicar y aceptar.

Ahora ya tenemos una carpeta con permisos sobre la que trabajar. En ella crearemos una tablespace llamada tb1 mediante el siguiente comando en SQL Shell:

```
-- 3. CREAR TABLESPACE tb1
CREATE TABLESPACE tb1 LOCATION 'C:\Users\mvent\OneDrive\Escritorio\Mario\Universidad\4
CARRERA\1 SEMESTRE\SGBD\Practica 2\tablespace';
```

Para saber si tb1 ha sido creado correctamente, podemos ejecutar el comando **\db** en la misma consola, y comprobamos lo siguiente:

```
practica2=# CREATE TABLESPACE tb1 LOCATION 'C:\Users\mvent\OneDrive\Escritorio\Mario\Universidad\4 CARRERA\1 SEMESTRE\SGBD\Practica 2\tablespace';
CREATE TABLESPACE
practica2=# \db
                                        Listado de tablespaces
   Nombre   |  Due±o   |                                          Ubicaci¾n
------------+----------+-----------------------------------------------------------------------------------------------------
 pg_default | postgres |
 pg_global  | postgres |
 tb1        | postgres | C:\Users\mvent\OneDrive\Escritorio\Mario\Universidad\4 CARRERA\1 SEMESTRE\SGBD\Practica 2\tablespace
(3 filas)


practica2=#
```

Como vemos, el tablespace tb1 ha sido creado correctamente en la carpeta mostrada y tratada anteriormente. También pueden verse dos valores: **pg_default** y **pg_global**, que representan el tablespace almacena los datos de los usuarios y el tablespace que almacena los datos globales respectivamente. Estos no han sido creados por nosotros y por eso no tienen ningún valor en la columna de ubicación.

5. MOVER LA BASE DE DATOS AL TABLESPACE

Ahora ya tenemos el tablespace creado y se nos pide que se mueva la base de datos, junto con todos sus objetos, a este nuevo espacio habilitado en el apartado anterior. Esto se puede hacer mediante en SQL Shell mediante el comando siguiente:

```
-- 4. MOVER DATABASE A tb1
ALTER DATABASE practica2 SET TABLESPACE tb1;
```

Cabe destacar que PostgreSQL no nos dejará ejecutar este comando si hay algún usuario conectado a la base de datos. Para poder ejecutarlo, debemos hacer sin estar conectados a ella o sin "estar dentro" por así decirlo. De lo contrario saldrán errores del tipo: ***ERROR: no se puede cambiar el tablespace de la base de datos activa***

Si hacemos todo correctamente, el resultado será el siguiente:

```
postgres=# -- 4. MOVER DATABASE A tb1
postgres=# ALTER DATABASE practica2 SET TABLESPACE tb1;
ALTER DATABASE
postgres=#
```

Si ejecutamos el comando **\db+**, obtendremos una información ampliada acerca de cada tablespace que hay en nuestra base de datos. De toda la información que se muestra, nos interesa la columna 'tamaño', ya que, si ejecutamos \db+ antes y después de mover la base de datos a la tablespace tb1, deberíamos ver un cambio notable en el tamaño de tb1 como consecuencia de haber movido la información. En nuestro caso, hemos hecho esto y se aprecia el siguiente cambio:

Antes de mover la bd a tb1

```
practica2=# \db+
                                                                 Listado de tablespaces
   Nombre    |  Due±o   |                                  Ubicaci¾n                                  | Privilegios | Opcion
es | Tama±o  | Descripci¾n
-------------+----------+-----------------------------------------------------------------------------+-------------+-------
---+---------+-------------
 pg_default | postgres |                                                                             |             |
   | 87 MB   |
 pg_global  | postgres |                                                                             |             |
   | 591 kB  |
 tb1        | postgres | C:\Users\mvent\OneDrive\Escritorio\Mario\Universidad\4 CARRERA\1 SEMESTRE\SGBD\Practica 2\tablespace |             |
   | 0 bytes |
(3 filas)
```

Después de mover la bd a tb1

```
postgres=# \db+
                                                                 Listado de tablespaces
   Nombre    |  Due±o   |                                  Ubicaci¾n                                  | Privilegios | Opcion
es | Tama±o  | Descripci¾n
-------------+----------+-----------------------------------------------------------------------------+-------------+-------
---+---------+-------------
 pg_default | postgres |                                                                             |             |
   | 78 MB   |
 pg_global  | postgres |                                                                             |             |
   | 591 kB  |
 tb1        | postgres | C:\Users\mvent\OneDrive\Escritorio\Mario\Universidad\4 CARRERA\1 SEMESTRE\SGBD\Practica 2\tablespace |             |
   | 8900 kB |
(3 filas)
```

Podemos apreciar que el tamaño de tb1 ha pasado de ser de 0 bytes a ser de 8900kb. Esto nos indica que tb1 no está vacío y que, por tanto, ahora contiene una información que antes no contenía.

Otra forma de comprobar el cambio (de hecho, probablemente sea la mejor forma de comprobarlo) de ubicación de la base de datos es preguntando directamente a PostgreSQL dónde está almacenada nuestra base de datos. Esto lo haremos ejecutando el comando \l+ practica2, que nos mostrará una información detallada sobre la base de datos especificada en el comando (practica2).

```
\l+ practica2;
```

```
postgres=# \l+ practica2;
                                                                    Listado de base de datos
  Nombre   |  Due±o   | Codificaci¾n | Proveedor de locale |      Collate       |       Ctype        | configuraci¾n ICU | Reglas ICU: | Privilegios
 | Tama±o  | Tablespace | Descripci¾n
-----------+----------+--------------+---------------------+--------------------+--------------------+-------------------+-------------+------------
-+---------+------------+-------------
 practica2 | postgres | UTF8         | libc                | Spanish_Spain.1252 | Spanish_Spain.1252 |                   |             |
 | 8836 kB | tb1        |
(1 fila)
```

De entre toda la información que muestra el comando, nos interesa la columna ‘Tablespace’, donde podemos ver que el valor es tb1 y que, por tanto, hemos movido correctamente la base de datos al tablespace tb1, creado en el apartado 3.

6. ANALIZAR LA TABLA ALBARA Y EXTRAER LA CANTIDAD DE REGISTROS QUE CABEN EN CADA BLOQUE.

Para analizar la tabla Albara y extraer la cantidad de registros que caben en cada bloque seguiremos el mismo procedimiento seguido en el apartado 2 para la tabla Aliment. Recordemos que cada tabla tiene 1500 registros y que cada bloque tiene el tamaño exacto de 8192 bytes.

Ejecutaremos la misma consulta que en el apartado 2, pero ahora actuaremos sobre la tabla Albara:

```
-- 5. REGISTROS POR CADA BLOQUE EN ALBARA
SELECT (ctid::text::point)[0]::bigint AS numero_bloque, count(*) AS numero_registros
FROM albara
GROUP BY numero_bloque
ORDER BY numero_bloque;
```

Obtenemos el siguiente resultado:

```
practica2=# -- 5. REGISTROS POR CADA BLOQUE EN ALBARA
practica2=# SELECT (ctid::text::point)[0]::bigint AS numero_bloque, count(*) AS numero_registros
practica2-# FROM albara
practica2-# GROUP BY numero_bloque
practica2-# ORDER BY numero_bloque;
 numero_bloque | numero_registros
---------------+------------------
             0 |              157
             1 |              157
             2 |              157
             3 |              157
             4 |              157
             5 |              157
             6 |              157
             7 |              157
             8 |              157
             9 |               87
(10 filas)
```

Puede verse una vez más que los 1500 registros de la tabla están separados en 10 bloques diferentes, 9 de los cuales tienen 157 registros. Dado que el único bloque que no tiene 157 registros es el último, esto nos puede hacer pensar en que quizá, en cada bloque caben exactamente 157 registros y que estos registros se han guardado de manera secuencial en sus respectivos bloques hasta llegar al último, donde solo faltaban 87 para llegar a 1500, y por eso el último bloque es el único que tiene 87 registros.

Sin embargo, esto es solo una suposición. Haremos un cálculo al igual que lo hicimos en el apartado 2 y, previsiblemente, obtendremos que 150 es el número promedio de registros por bloque (recordemos que es un valor aproximado ya que se trata de una media aritmética)

$$\frac{87 + 9 \cdot (157)}{10} = \mathbf{150}\ \boldsymbol{registros\ por\ bloque}$$

7. CAMBIAR EL TAMAÑO DEL BLOQUE A 32K

Para poder cambiar el tamaño del bloque a 32K tendremos que instalar PostgreSQL desde el código fuente, de forma que podamos "reconfigurarlo" desde cero, estableciendo desde un inicio que queremos que el tamaño del bloque sea de 32K. Para hacer esto, hemos descargado y descomprimido el archivo ***tar.gz*** comprimido desde el siguiente enlace:

https://ftp.postgresql.org/pub/source/v15.0/postgresql-15.0.tar.gz

Una vez hecho esto, dado que estamos usando Windows, tendremos que usar **MSYS32 MINGW64** como consola para poder llevar a cabo instrucciones de Linux en Windows (otra solución podría ser usar una partición de disco en la que tengamos instalado Ubuntu y usar su consola).

Dentro de la consola que nos ofrece MSYS32 MINGW64, accederemos a la carpeta que hemos obtenido tras descomprimir el archivo tar.gz. Haremos esto con el **comando cd** y, cuando estemos dentro de esa carpeta, podremos comenzar a configurar inicialmente el "nuevo" postgres con tamaño de bloque de 32K. Para llevar a cabo esta configuración se hacen los siguientes comandos:

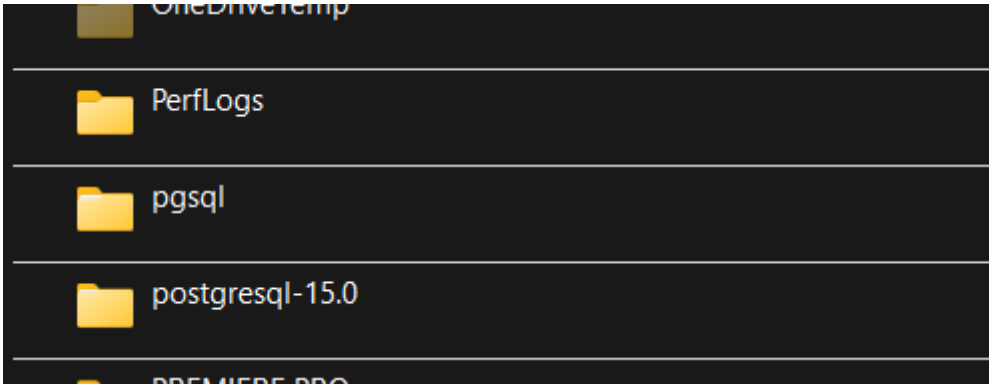
```
pacman -Su
pacman --needed -S git mingw-w64-x86_64-gcc base-devel
```

Estos dos comandos se usan para buscar actualizaciones de todos los paquetes instalados en el sistema y para instalar nuevos paquetes, respectivamente. En este caso, el segundo comando, está instalando los paquetes git, mingw-w64-x86_64-gcc, y base-devel. Por otro lado, --needed instala únicamente los paquetes que aún no están instalados en el sistema.

Ahora, con esto hecho y estando situados dentro de la carpeta previamente mencionada, procedemos con la nueva configuración de tamaño del bloque mediante el siguiente comando:

```
./configure --host=x86_64-w64-mingw32 --prefix=/c/pgsql/ --with-blocksize=32 --with-wal-blocksize=32 && make && make install
```

Tras esto, deberíamos tener una nueva carpeta llamada pgsql, que será donde se alojará el nuevo postgres. Podemos comprobar con el explorador de archivos de Windows si esta carpeta se ha creado, o mediante la propia consola accediendo a la ruta donde esta carpeta debería encontrarse y ejecutando el comando **ls** (o **ls -l** para más información) para ver el contenido de esta.



Podemos comprobar como efectivamente la carpeta pgsql ha sido creada correctamente

Ahora crearemos un nuevo cluster donde, por seguridad, guardaremos una copia de nuestra base de datos como medida preventiva a posibles errores. Esto lo haremos desde el **cmd** del propio Windows (como administrador), donde ejecutaremos el siguiente comando:

pg_dump -U postgres -W -h localhost practica2 > p2.sql

Ahora que ya tenemos la copia de seguridad, en una nueva carpeta de nombre pgdata situada en C:\pgdata, podemos crear el cluster:

initdb -D C:\pgcluster -U postgres -A trust

```
Los archivos de este cluster serán de propiedad del usuario «mvent».
Este usuario también debe ser quien ejecute el proceso servidor.

El cluster será inicializado con configuración regional «Spanish_Spain.1252».
La codificación por omisión ha sido por lo tanto definida a «WIN1252».
La configuración de búsqueda en texto ha sido definida a «spanish».

Las sumas de verificación en páginas de datos han sido desactivadas.

corrigiendo permisos en el directorio existente C:/pgcluster ... hecho
creando subdirectorios ... hecho
seleccionando implementación de memoria compartida dinámica ... windows
seleccionando el valor para max_connections ... 100
seleccionando el valor para shared_buffers ... 128MB
seleccionando el huso horario por omisión ... Europe/Paris
creando archivos de configuración ... hecho
ejecutando script de inicio (bootstrap) ... hecho
realizando inicialización post-bootstrap ... hecho
sincronizando los datos a disco ... hecho

Completado. Ahora puede iniciar el servidor de bases de datos usando:

    pg_ctl -D ^"C^:^\pgcluster^" -l archivo_de_registro start
```

Hecho esto, cambiaremos el puerto 5432 a 5433 en el archivo de configuración ***postgresql.conf***, para no usar el mismo puerto que en el cluster original.

```
#------------------------------------------------------------------------------
# CONNECTIONS AND AUTHENTICATION
#------------------------------------------------------------------------------

# - Connection Settings -

#listen_addresses = 'localhost'         # what IP address(es) to listen on;
                                        # comma-separated list of addresses;
                                        # defaults to 'localhost'; use '*' for all
                                        # (change requires restart)
port = 5433                             # (change requires restart)
max_connections = 100                   # (change requires restart)
#reserved_connections = 0               # (change requires restart)
#superuser_reserved_connections = 3     # (change requires restart)
#unix_socket_directories = ''           # comma-separated list of directories
```

Hecho esto, accedemos al cluster y comprobamos que se está ejecutando con los siguientes comandos:

pg_ctl -D C:\pgcluster start
pg_ctl -D C:\cluster1 status

Ahora accedemos al cluster nuevo y le pedimos que nos muestre el tamaño del blocksize:

psql -d postgres -p 5433 -U postgres
show block_size

```
C:\pgsql\bin>psql -d postgres -p 5433 -U postgres
psql (15.0)
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=# \c p2
You are now connected to database "p2" as user "postgres".
```

```
p2=# \c postgres
You are now connected to database "postgres" as user "postgres".
postgres=# show block_size;
 block_size
------------
 32768
(1 row)
```

Como puede verse, el tamaño del bloque ahora es de 32768 bytes, que equivale a 32KB.
Por último, vamos a crear una base de datos nueva donde copiaremos toda la información de la base de datos original. Para ello haremos lo siguiente:

psql -h localhost -p 5433 -U postgres -f p2.sql new_p2

```
postgres=# \c p2
You are now connected to database "p2" as user "postgres".
p2=# \dt
           List of relations
 Schema |     Name     | Type  |  Owner
--------+--------------+-------+----------
 public | albara       | table | postgres
 public | aliment      | table | postgres
 public | linia_albara | table | postgres
 public | proveidor    | table | postgres
 public | venda        | table | postgres
(5 rows)
```

Donde p2.sql es el archivo sql que contiene la información con la copia que hicimos de la base de datos original, y new_p2 será el nombre de la base de datos nueva.

Podemos comprobar, por tanto, como el tamaño del bloque ha sido modificado correctamente a 32KB

8. ANALIZAR LAS TABLAS ALIMENT Y ALBARA Y EXTRAER EL NÚMERO DE REGISTROS POR CADA BLOQUE

Para saber cuantos registros caben en cada bloque ahora que el bloque es de 32K, seguiremos el mismo procedimiento usado en los casos anteriores. El resultado debería ser diferente esta vez ya que, al ser 4 veces mayor el tamaño el bloque (32/8 = 4 = bloque con cuádruple de capacidad), la cantidad de registros que caben por bloque debería ser también mayor. Recordemos también, que la cantidad de registros en cada tabla no ha sido modificada y sigue siendo 1500.

Hacemos la misma consulta que en los apartados 2 y 5 y obtenemos lo siguiente:

```
-- 7. ANALISIS CON 32K DE BLOQUE
    -- 7.1 Para aliment
SELECT (ctid::text::point)[0]::bigint AS numero_bloque, count(*) AS numero_registros
FROM aliment
GROUP BY numero_bloque
ORDER BY numero_bloque;

    -- 7.2 Para albara
SELECT (ctid::text::point)[0]::bigint AS numero_bloque, count(*) AS numero_registros
FROM albara
GROUP BY numero_bloque
ORDER BY numero_bloque;
```

```
p2=# SELECT (ctid::text::point)[0]::bigint AS numero_bloque, count(*) AS numero_registros FROM aliment GROUP BY numero_bloque ORDER BY numero_bloque;
 numero_bloque | numero_registros
---------------+-----------------
             0 |             644
             1 |             629
             2 |             227
(3 rows)
```

```
p2=# SELECT (ctid::text::point)[0]::bigint AS numero_bloque, count(*) AS numero_registros FROM albara GROUP BY numero_bloque ORDER BY numero_bloque;
 numero_bloque | numero_registros
---------------+-----------------
             0 |             629
             1 |             629
             2 |             242
(3 rows)
```

Tal y como vemos, en ambas tablas los registros se dividen en un total de 3 bloques enumerados de 0 a 2. Puede notarse que, en ambas tablas, el último bloque es el que menos registros tiene. Cobra importancia, por tanto, la idea de que en general, se llenan pretende llenar por completo un bloque de datos antes de pasar al siguiente; aunque se sabe que la gestión interna puede ser mucho más compleja. De todas formas, y tal y como mencionamos anteriormente, esto es solo una hipótesis. Si realizamos un cálculo como en el de los apartados 2 y 5, en este caso obtenemos:

$$\frac{644 + 629 + 227}{3} = \frac{629 + 629 + 242}{3} = \mathbf{500}\ \boldsymbol{registros\ por\ bloque}$$

9. ANALIZA Y EXPLICA LAS DIFERENCIAS

Pueden observarse varias diferencias cuando se usan bloques de 8KB en comparación con cuando se usan bloques de 32KB.
La evidente diferencia principal es que, cuanto mayor sea el tamaño del bloque, más registros caben dentro de este bloque y, por tanto, se usan menos bloques para almacenar la misma información. Esto puede ser beneficioso ya que un tamaño de bloque más grande puede ayudar a reducir la fragmentación, ya que cabrían más registros en un solo bloque y, por lo tanto, habría menos posibilidades de que se produzcan fragmentaciones; aunque también es cierto que la fragmentación puede aumentar si el sistema operativo no consigue asignar un tamaño de bloque de 32KB de manera contigua

Sin embargo, usar un tamaño de bloque de 32KB sin que sea estrictamente necesario es bastante inútil e inadecuado. Este tamaño de bloques suele usarse en almacenamiento de datos a gran escala, en almacenamientos de datos columnares, almacenes de datos con grandes cargas de lectura, etc.

Será tarea del gestor de la base de datos escoger qué tamaño de bloque debe usarse en el caso específico de la base de datos que se esté gestionando. Aun así, PostgreSQL ofrece 8KB como tamaño predeterminado del bloque porque, en términos generales, suele ser un tamaño correcto/adecuado en la mayoría de los casos (aunque es importante que el gestor conozca que el tamaño predeterminado es 8KB y cómo puede cambiarse este tamaño si así se desea).

Algunas de las ventajas que supone el uso de un bloque de mayor tamaño son, entre otras tantas:

- Menor sobrecarga de E/S ya que hay menos operaciones
- Menor cantidad de bloques para la misma cantidad de información
- Más datos en cada bloque
- Menos bloqueos a procesos
- Mejora (o no) de la fragmentación si el SO asigna bloques del tamaño dado de manera contigua

En nuestro caso, si hablamos de cifras, puede observarse que hemos pasado de calcular entorno a 150 registros por bloque a obtener 500 registros por bloque. Esto supone un **330% más de capacidad** de contener registros por parte de cada bloque. Pese a que el tamaño del bloque es el cuádruple (32/8 = 4), no apreciamos, al menos en base a los cálculos, un 400% más de capacidad (cuádruple = 4 veces más = 4·100%), ya que hemos obtenido un 330% más de capacidad y no un 400%. Sin embargo, en los apartados 2 y 5 obtuvimos que se podía almacenar unos 150 registros por bloque y, ahora, vemos que hay ciertos bloques con más de 600 registros, siendo 600 el cuádruple de 150 (600/150 = 4). Con lo cual sí que se supera esa capacidad cuadruplicada en algunos casos.

El hecho de que el tercer bloque en las tablas Aliment y Albara con tamaño de bloque 32kb contenga aproximadamente 200 registros hace que la media aritmética baje hasta 330%. Es evidente que, si nuestra base de datos contase con más registros, esta media subiría ya que todavía caben muchos registros en el tercer bloque de estas dos tablas. Esto haría que se alcanzasen las cifras que se esperaba obtener ya que obviamente, si la capacidad del bloque aumenta en un 400%, los cálculos deberían mostrarlo así.

Esto nos demuestra una vez más que este 330% y otros tantos datos obtenidos no son del todo realistas y que solo presentan una cifra orientativa, y es que en los apartados 2 y 5, si bien es cierto que también había bloques con una cantidad notablemente reducida de registros en comparación con el resto de los bloques, también es cierto que el hecho de tener 10 bloques hace que la media no se vea enormemente afectada si uno de ellos presenta una cantidad reducida de registros.

En conclusión, el tamaño del bloque a utilizar en una base de datos es decisión del gestor de esta. Se pueden usar bloques de 2KB hasta 32KB o más, dependiendo del sistema gestor que se utilice. Cada tamaño de bloque presenta características diferentes y por tanto tiene sus propias ventajas, desventajas y casos de uso recomendados