

Titulación: Grado en Ingeniería Informática y Sistemas de Información

Curso: 2019-2020. Convocatoria Ordinaria de Junio

Asignatura: Bases de Datos Avanzadas – Laboratorio

Practica 3: Seguridad, Usuarios y Transacciones.

ALUMNO 1:

Nombre y Apellidos: Adina Murg

DNI:

ALUMNO 2:

Nombre y Apellidos: Victoria Lorena Ordenes Orbegozo

DNI:

Fecha: 1/06/2020_____

Profesor Responsable: _____ Iván González _____

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 puntuará **TODA** la asignatura como **Suspenso – Cero**.

Plazos

Tarea online: Semana 13 de Abril, Semana 20 de Abril y semana 27 de Abril.

Entrega de práctica: **Día 18 de Mayo (provisional).** Aula Virtual

Documento a entregar: Este mismo fichero con las respuestas a las cuestiones planteadas, con el código SQL utilizado en cada uno de los aparatos. Si se entrega en formato electrónico se entregará en un ZIP comprimido: **DNI ' sdelosAlumnos_PECL3.zip**

AMBOS ALUMNOS DEBEN ENTREGAR EL FICHERO EN LA PLATAFORMA.

Introducción

El contenido de esta práctica versa sobre el manejo de las transacciones en sistemas de bases de datos, así como el control de la concurrencia y la recuperación de la base de datos frente a una caída del sistema. Las transacciones se definen como una unidad lógica de procesamiento compuesta por una serie de operaciones simples que se ejecutan como una sola operación. Entre las etiquetas BEGIN y COMMIT del lenguaje SQL se insertan las operaciones simples a realizar en una transacción. La sentencia ROLLBACK sirve para deshacer todos los cambios involucrados en una transacción y devolver a la base de datos al estado consistente en el que estaba antes de procesar la transacción. También se verá el registro diario o registro histórico del sistema de la base de datos (en PostgreSQL se denomina WAL: Write Ahead Loggin) donde se reflejan todas las operaciones sobre la base de datos y que sirve para recuperar ésta a un estado consistente si se produjera un error lógico o de hardware. La versión de postgres a utilizar deberá ser la versión 12.

Actividades y Cuestiones

En esta parte la base de datos **TIENDA** deberá de ser nueva y no contener datos. Además, consta de 5 actividades:

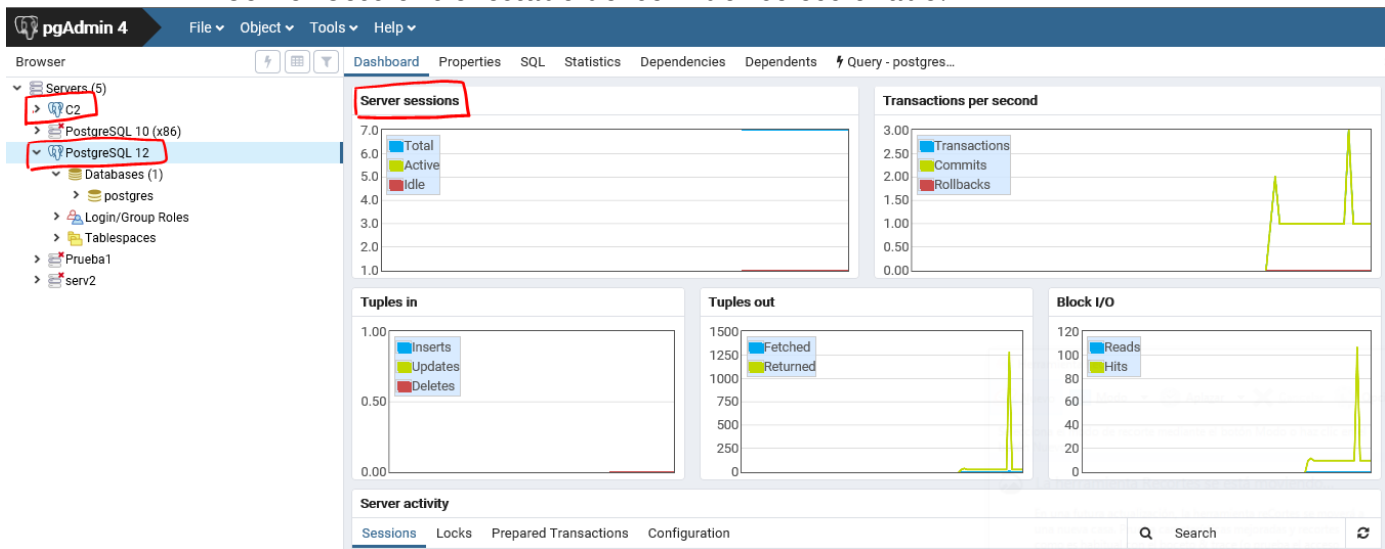
- Conceptos generales.
- Manejo de transacciones.
- Concurrencia.
- Registro histórico.
- Backup y Recuperación

Cuestión 1: Arrancar el servidor Postgres si no está y determinar si se encuentra activo el diario del sistema. Si no está activo, activarlo. Determinar cuál es el directorio y el archivo/s donde se guarda el diario. ¿Cuál es su tamaño? Al abrir el archivo con un editor de textos, ¿se puede deducir algo de lo que guarda el archivo?

1. Con el servidor de Postgres activado, determinar si está activo el diario del sistema

El server es la conexión con Postgres.

Aquí podemos observar que tenemos 5 servidores, pero sólo 2 de ellos activos, mientras que los otros 3 no están activos. (Se trabajará con Postgres 12, que obviamente está activo). Por otro lado, si miramos en Dashboard podemos ver en Server Sessions el estado del servidor seleccionado.



El WAL (diario del sistema) se encuentra activo por defecto en Postgres. Para comprobarlo, podemos también realizar una consulta simple:

```
SELECT * FROM pg_ls_waldir()
```

Pg_ls_waldir: lista el nombre, tamaño y última modificación de los archivos del directorio WAL (Write Ahead Loggin). Información [aquí](#).

postgres on postgres@PostgreSQL 12

Query Editor Query History

```
1 SELECT * FROM pg_ls_waldir()
```

Data Output Explain Messages Notifications

	name text	size bigint	modification timestamp with time zone
1	0000000100000000000000EE	16777216	2020-04-21 13:46:06+02
2	0000000100000000000000EF	16777216	2020-04-06 22:24:32+02
3	0000000100000000000000F0	16777216	2020-04-06 22:24:38+02
4	0000000100000000000000F1	16777216	2020-04-06 22:24:45+02
5	0000000100000000000000F2	16777216	2020-04-06 22:24:52+02
6	0000000100000000000000F3	16777216	2020-04-06 22:24:58+02
7	0000000100000000000000F4	16777216	2020-04-06 22:25:07+02
8	0000000100000000000000F5	16777216	2020-04-06 22:25:12+02
9	0000000100000000000000F6	16777216	2020-04-06 22:25:18+02
10	0000000100000000000000F7	16777216	2020-04-06 22:25:26+02
11	0000000100000000000000F8	16777216	2020-04-06 22:25:45+02
12	0000000100000000000000F9	16777216	2020-04-06 22:25:56+02
13	0000000100000000000000FA	16777216	2020-04-06 22:26:39+02

Con esto comprobamos que obtenemos resultados, que coincidirán con los almacenados en el directorio y mostraremos a continuación.

El directorio donde se guarda el diario WAL es: **C:\Program Files\PostgreSQL\12\data\pg_wal**

Encontramos muchos archivos de 16 MB todos y con nombres en hexadecimal (0-9, A-F).

00000001000000030000002A	16/04/2020 20:50	Archivo	16.384 KB
00000001000000030000002B	16/04/2020 20:50	Archivo	16.384 KB
00000001000000030000002C	16/04/2020 20:50	Archivo	16.384 KB
00000001000000030000002D	16/04/2020 20:50	Archivo	16.384 KB
00000001000000030000002E	16/04/2020 20:50	Archivo	16.384 KB
00000001000000030000002F	16/04/2020 20:50	Archivo	16.384 KB
00000001000000030000003A	16/04/2020 20:50	Archivo	16.384 KB
00000001000000030000003B	16/04/2020 20:50	Archivo	16.384 KB
00000001000000030000003C	16/04/2020 20:50	Archivo	16.384 KB
00000001000000030000003D	16/04/2020 20:50	Archivo	16.384 KB
00000001000000030000003E	16/04/2020 20:50	Archivo	16.384 KB
00000001000000030000003F	16/04/2020 20:50	Archivo	16.384 KB



















Dentro del archivo podemos comprobar que el documento no es legible con un editor de textos, dado que los archivos WAL son registros binarios.


```
tienda on postgres@PostgreSQL 12
Query Editor  Query History
1  insert into tienda (id_tienda, nombre, ciudad, barrio, provincia)
2  values (645487389, 'tienda 24h','Madrid', 'Centro', 'Madrid' );

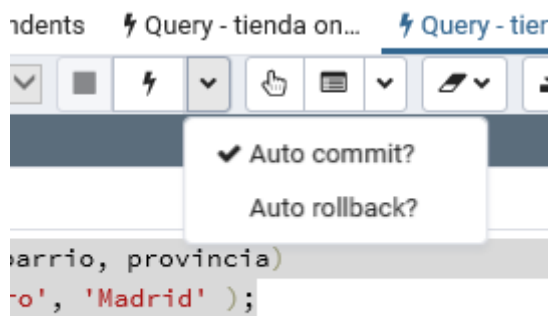
Data Output  Explain  Messages  Notifications
INSERT 0 1

Query returned successfully in 1 secs 460 msec.
```

Después de realizar el insert volvemos a los wal y buscamos el actualizado, el más reciente:

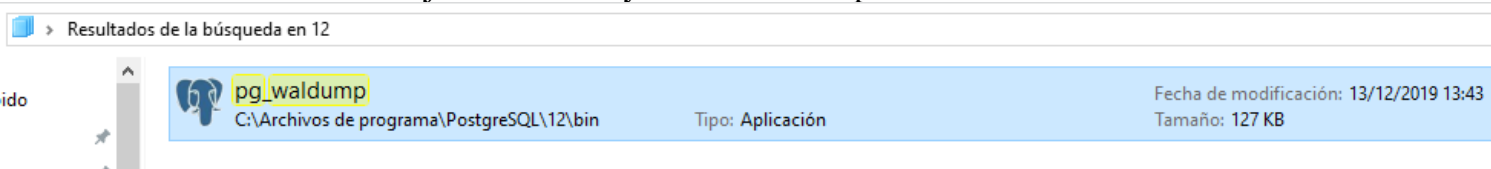
	00000001000000010000000A	21/04/2020 17:58
	00000001000000010000000B	21/04/2020 17:59
	00000001000000010000000C	21/04/2020 17:59
	00000001000000010000000D	21/04/2020 17:59
	00000001000000010000000E	21/04/2020 18:00
	00000001000000010000000F	21/04/2020 18:00
	000000010000000100000005	21/04/2020 18:34
	000000010000000100000006	06/04/2020 22:34
	000000010000000100000007	06/04/2020 22:34
	000000010000000100000008	06/04/2020 22:35
	000000010000000100000009	21/04/2020 17:58
	000000010000000100000010	21/04/2020 18:03
	000000010000000100000011	21/04/2020 18:03
	000000010000000100000012	21/04/2020 18:04
	000000010000000100000013	21/04/2020 18:04
	000000010000000100000014	21/04/2020 18:06
	000000010000000100000015	21/04/2020 18:06
	000000010000000100000016	21/04/2020 18:06

Ahora abrimos el wal y buscamos dentro de él, en este caso buscamos por “tienda” y podemos ver que se ha quedado registrado, además de ser legible.



Cuestión 3: ¿Para qué sirve el comando `pg_waldump.exe`? Aplicarlo al último fichero de WAL que se haya generado. Obtener las estadísticas de ese fichero y comentar qué se está viendo.

Buscamos el ejecutable .exe y localizamos el path:



A continuación, abrimos cmd y ejecutamos el comando, pero nos exige obligatoriamente pasarle algún parámetro. Procedemos a usar `--help` para ver qué opciones tenemos:

```
C:\Users\QuasarPC>cd C:\Program Files\PostgreSQL\12\bin
C:\Program Files\PostgreSQL\12\bin>pg_waldump.exe
pg_waldump: error: no se especificó ningún argumento
Pruebe «pg_waldump --help» para mayor información.

C:\Program Files\PostgreSQL\12\bin>pg_waldump.exe
pg_waldump: error: no se especificó ningún argumento
Pruebe «pg_waldump --help» para mayor información.

C:\Program Files\PostgreSQL\12\bin>pg_waldump.exe --help
pg_waldump decodifica y muestra segmentos de WAL de PostgreSQL para depuración.

Empleo:
  pg_waldump [OPCIÓN]... [SEGINICIAL [SEGFINAL]]

Opciones:
-b, --bkp-details      mostrar información detallada sobre bloques de respaldo
-e, --end=RECPTR       detener la lectura del WAL en la posición RECPTR
-f, --follow           seguir reintentando después de alcanzar el final del WAL
-n, --limit=N          número de registros a mostrar
-p, --path=RUTA        directorio donde buscar los archivos de segmento de WAL
                        o un directorio con un ./pg_wal que contenga tales archivos
                        (por omisión: directorio actual, ./pg_wal, $PGDATA/pg_wal)
-r, --rmgr=GREC        sólo mostrar registros generados por el gestor de
                        recursos GREC; use --rmgr=list para listar nombres válidos
-s, --start=RECPTR     empezar a leer el WAL en la posición RECPTR
-t, --timeline=TLI     timeline del cual leer los registros de WAL
                        (por omisión: 1 o el valor usado en SEGINICIAL)
-V, --version          mostrar información de versión, luego salir
-x, --xid=XID          sólo mostrar registros con el id de transacción XID
-z, --stats[=registro] mostrar estadísticas en lugar de registros
                        (opcionalmente, mostrar estadísticas por registro)
-?, --help            mostrar esta ayuda, luego salir
```


1. Para qué sirve walldump:

El comando de pg_walddump nos ayuda a hacer legibles los archivos WAL. Sólo se puede ejecutar por el usuario que instaló el servidor. Su utilidad principalmente es para depuración o análisis y estudio.

Según la propia descripción del archivo, 'pg_walddump - decode and display WAL.'

Información [aquí](#).

2. Aplicarlo al último registro WAL creado y obtener las estadísticas:

Como bien nos indica help, el comando a emplear será pg_walddump -z seguido del nombre en hexadecimal del archivo que queremos estando en C:\Program Files\PostgreSQL\12\data\pg_wal>.

El nombre del archivo que contiene el último log como podemos observar en el ejercicio anterior es: 000000010000000100000005

```
C:\Program Files\PostgreSQL\12\data\pg_wal>pg_walddump.exe -V
pg_walddump (PostgreSQL) 12.1

C:\Program Files\PostgreSQL\12\data\pg_wal>pg_walddump.exe -z 000000010000000100000005
Type          N      (%)      Record size      (%)      FPI size      (%)      Combined size      (%)
-----
XLOG          3 ( 0,02)      342 ( 0,01)      0 ( 0,00)      342 ( 0,01)
Transaction   3 ( 0,02)      166 ( 0,00)      0 ( 0,00)      166 ( 0,00)
Storage       1 ( 0,01)      46 ( 0,00)      0 ( 0,00)      46 ( 0,00)
CLOG          0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Database      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Tablespace    0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
MultiXact     0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
RelMap        0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Standby       10 ( 0,06)      492 ( 0,01)      0 ( 0,00)      492 ( 0,01)
Heap2         8597 ( 50,78)  2470348 ( 58,22)  8192 ( 33,59)  2478540 ( 58,08)
Heap          3 ( 0,02)      296 ( 0,01)      8708 ( 35,71)  9004 ( 0,21)
Btree         8312 ( 49,10)  1771500 ( 41,75)  7488 ( 30,70)  1778988 ( 41,69)
Hash          0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Gin           0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Gist          0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Sequence     0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
SPGist        0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
BRIN          0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
CommitTs      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
ReplicationOrigin 0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Generic       0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
LogicalMessage 0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
-----
Total          16929      4243190 [99,43%]      24388 [0,57%]      4267578 [100%]
pg_walddump: fatal: error in WAL record at 1/54207C8: invalid record length at 1/5420800: wanted 24, got 0

C:\Program Files\PostgreSQL\12\data\pg_wal>
```

3. Comentar qué estamos viendo:

Estas estadísticas nos muestran varios campos.

La columna **type** nos indica el tipo de funciones que hay almacenadas en ese archivo. **N** es el número de registros de cada tipo y **%** nos indica lo mismo que N, pero medido en porcentajes sobre el 100% total.

Record size nos informa del peso de dichas funciones y además podemos ver los % correspondientes.

FPI es el tamaño de los Full Page Image y además podemos compararlo en %

Combined size nos ofrece una suma del total de las columnas (observamos que tenemos 4.267.578 funciones) y el porcentaje que corresponde a un 100%.

Analizando ahora las funciones, destacamos que los que más peso tienen son los btree y heap/heap2.

Nota: El error que nos aparece: *pg_waldump: fatal: error in WAL record at 1/54207C8: invalid record length at 1/5420800: wanted 24, got 0*

Es debido a que intentamos acceder y leer un archivo WAL incompleto (aún se puede escribir en él) de un servidor o por acceder mientras un servidor se está recuperando de algún bloqueo.

Cuestión 4: Determinar el identificador de la transacción que realizó la operación anterior. Aplicar el comando anterior al último fichero de WAL que se ha generado y mostrar los registros que se han creado para esa transacción. ¿Qué se puede ver? Interpretar los resultados obtenidos.

1. Buscar el identificador de la transacción anterior

Como hemos podido observar en la captura anterior de emplear el comando `-help`, los archivos WAL tienen su propio sistema identificador, llamado XID (id de transacción). Según el propio manual de Postgres (pag 204), los XID son otro tipo de identificador (a parte del OID) usado por el sistema para identificar transacciones, proviene de la abreviación de *xact identifier* y tiene 32 bits.

Ahora que sabemos esto, debemos buscar el XID de nuestra transacción(insert). Según la página 364 del manual de [Postgres](#) encontramos dos instrucciones muy útiles, **txid_current()**, que nos informa de que obtiene la ID de la transacción y que en caso de ser necesario, asigna una nueva si la transacción actual no tiene ninguna.

También encontramos **txid_current_if_assigned()** que cumple las mismas funciones que el anterior sólo que devuelve NULL si la transacción actual no tiene ID.

```
SELECT * FROM txid_current();
```

The screenshot shows a PostgreSQL query editor window titled "tienda on postgres@PostgreSQL 12". It has tabs for "Query Editor" and "Query History". The query editor contains two lines of SQL code: `1 SELECT * FROM txid_current();` and `2 --SELECT * FROM txid_current_if_assigned();`. Below the query editor, there are tabs for "Data Output", "Explain", "Messages", and "Notifications". The "Data Output" tab is active, displaying a table with the following structure:

	txid_current
	bigint
1	545

Aplicamos dicho comando en una consulta y obtenemos un resultado, el XID de la transacción, que es 545.

2. Aplicar el comando anterior al último fichero WAL generado + mostrar registros creados para dicha transacción

Para este paso volvemos a CMD y volvemos a aplicar el mismo comando anterior.

```
C:\Program Files\PostgreSQL\12\data\pg_wal>pg_waldump.exe -x 545 -z 000000010000000100000005
```

Esta vez añadimos el campo --xid con el que asignamos el XID

```
C:\Program Files\PostgreSQL\12\data\pg_wal>pg_waldump.exe --xid 545 -z 000000010000000100000005
Type              N      (%)      Record size      (%)      FPI size      (%)      Combined size      (%)
-----
XLOG              0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Transaction      1 (100,00)     34 (100,00)      0 ( 0,00)      34 (100,00)
Storage          0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
CLOG             0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Database         0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Tablespace       0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
MultiXact        0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
RelMap           0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Standby          0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Heap2            0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Heap             0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Btree            0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Hash             0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Gin              0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Gist             0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Sequence         0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
SPGist           0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
BRIN             0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
CommitTs         0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
ReplicationOrigin 0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Generic          0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
LogicalMessage   0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
-----
Total            1              34 [100,00%]      0 [0,00%]      34 [100%]
pg_waldump: fatal: error in WAL record at 1/5420910: invalid record length at 1/5420948: wanted 24, got 0
C:\Program Files\PostgreSQL\12\data\pg_wal>
```

3. Interpretación de los resultados

Podemos observar que de la misma manera que antes, seguimos teniendo las mismas funciones, pero esta vez los valores de estas son muy bajos, o en todo salvo en la función de transacción.

Cuestión 5: Se va a crear un backup de la base de datos TIENDA. Este backup será utilizado más adelante para recuperar el sistema frente a una caída del sistema. Realizar solamente el backup mediante el procedimiento descrito en el apartado 25.3 del manual (versión 12 es "*Continuous Archiving and point-in-time recovery (PITR)*").

Para poder crear un backup de la base de datos hemos seguido las instrucciones del [manual](#).

Primero, hemos modificado el archivo postgresql.conf:

```
#-----
# WRITE-AHEAD LOG
#-----

# - Settings -

wal_level = replica                # minimal, replica, or logical
                                   # (change requires restart)
#fsync = on                        # flush data to disk for crash safety
                                   # (turning this off can cause
                                   # unrecoverable data corruption)

# - Archiving -

archive_mode = on                  # enables archiving; off, on, or always
                                   # (change requires restart)
archive_command = 'copy "%p" "C:\\PostgreSQL\\wal\\%f"' # command to use to archive a logfile segment
                                   # placeholders: %p = path of file to archive
                                   # %f = file name only
                                   # e.g. 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f'
#archive_timeout = 0              # force a logfile segment switch after this
                                   # number of seconds; 0 disables
```

wal_level lo dejamos en replica ya que usamos una versión superior a la 10.

Archive_mode = on

Archive_command = insertamos la ruta

Información adicional obtenida de [aquí](#).

Ahora, por cmd desde la carpeta C:\Program Files\PostgreSQL\12\bin hemos ejecutado pg_basebackup para realizar la copia de seguridad y lo hemos almacenado en el directorio raíz en una carpeta llamada copia.

```
C:\Program Files\PostgreSQL\12\bin>pg_basebackup -h localhost -U postgres -D C:\Copia\tienda.backup
Contraseña:
```

Obtenemos la copia de los archivos de data correctamente en el directorio indicado:

> Este equipo > Disco local (C:) > Copia > tienda.backup

Nombre	Fecha de modificación	Tipo	Tamaño
base	30/05/2020 16:47	Carpeta de archivos	
global	30/05/2020 16:47	Carpeta de archivos	
log	30/05/2020 16:47	Carpeta de archivos	
pg_commit_ts	30/05/2020 16:47	Carpeta de archivos	
pg_dynshmem	30/05/2020 16:47	Carpeta de archivos	
pg_logical	30/05/2020 16:47	Carpeta de archivos	
pg_multixact	30/05/2020 16:47	Carpeta de archivos	
pg_notify	30/05/2020 16:47	Carpeta de archivos	
pg_replslot	30/05/2020 16:47	Carpeta de archivos	
pg_serial	30/05/2020 16:47	Carpeta de archivos	
pg_snapshots	30/05/2020 16:47	Carpeta de archivos	
pg_stat	30/05/2020 16:47	Carpeta de archivos	
pg_stat_tmp	30/05/2020 16:47	Carpeta de archivos	
pg_subtrans	30/05/2020 16:47	Carpeta de archivos	
pg_tblspc	30/05/2020 16:47	Carpeta de archivos	
pg_twophase	30/05/2020 16:47	Carpeta de archivos	
pg_wal	30/05/2020 16:47	Carpeta de archivos	
pg_xact	30/05/2020 16:47	Carpeta de archivos	
backup_label	30/05/2020 16:46	Archivo	1 KB
current_logfiles	30/05/2020 16:47	Archivo	1 KB
pg_hba	30/05/2020 16:47	Archivo CONF	5 KB
pg_ident	30/05/2020 16:47	Archivo CONF	2 KB
PG_VERSION	30/05/2020 16:47	Archivo	1 KB
postgresql.auto	30/05/2020 16:47	Archivo CONF	1 KB
postgresql	30/05/2020 16:47	Archivo CONF	27 KB

Pg_basebackup hace una copia de seguridad de todo el sistema de archivos del cluster de la base de datos. La copia que crea es CONSISTENTE. Copia exactamente todo lo que hay en la carpeta data en el momento que se hace la copia.

Pg_basebackup -> para backup de un cluster de la base de datos de postgres activo.

Hace una copia binaria de los archivos del cluster.

El backup se hace con una conexión postgres y usa el protocolo de replicación.

La conexión se debe hacer con un usuario con permisos de replicación (en nuestro caso postgres que los tiene) además de permisos en pg_hba.con.

Comandos a tener en cuenta:

-D -> para indicar el directorio en que guardar la carpeta copia del cluster

-F -> formato (t de tar y p de plain)

-h -> host

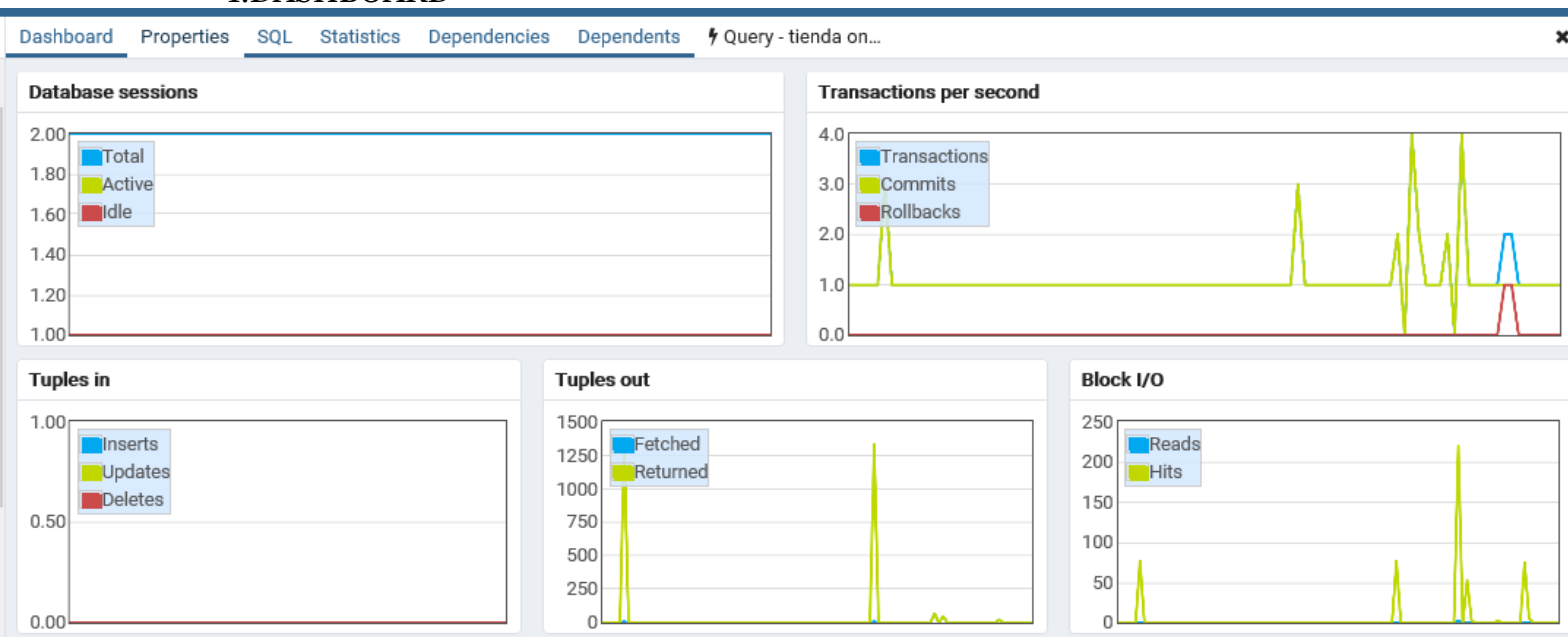
-p -> puerto tcp en el que el cluster está activo

-u -> username con el que nos queremos conectar (con privilegios de replicación)

Cuestión 6: ¿Qué herramientas disponibles tiene PostgreSQL para controlar la actividad de la base de datos en cuanto a la concurrencia y transacciones? ¿Qué información es capaz de mostrar? ¿Dónde se guarda dicha información? ¿Cómo se puede mostrar?

1. Herramientas de Postgres para controlar la actividad de concurrencia y transacciones + qué información muestra

1.DASHBOARD



El dashboard contiene una serie de gráficas que nos informa del estado de servidor. Entre ellas están las de Tuples In/Out (tuplas buscadas y retornadas al usuario),

Blocks I/O(con Hits, aciertos en caché o Reads en disco) y Database session, ya mencionada en el Ejercicio 1.

Una **transacción** es una unidad lógica de procesamiento compuesta de instrucciones individuales, que acceden y modifican los datos de nuestra base de datos y se ejecutan por completo o no.

La **conurrencia** se encarga de asegurar la propiedad de aislamiento entre transacciones (mientras está una activa y ejecutándose no se cuele otra en medio, etc). Cada SGBD tiene su propio esquema de control de concurrencias. Las relacionaremos con el concepto de bloqueo, que es un mecanismo de control concurrente para el acceso a un dato. Un bloqueo será compartido (sólo lectura) o exclusivo (escritura).

Con estos conceptos, podemos analizar la última gráfica que es la que nos interesa, la de Transactions per seconds. Esta nos informa de los diversos estados de las transacciones que tenemos. ‘**Transactions**’ son las que están activas, pero cuyos cambios aún no son visibles a todos los usuarios (tienen un BEGIN), ‘**Commits**’ son las transacciones finalizadas con éxito, lo que quiere decir que la transacción ya ha registrado todos los cambios y son visibles para todos los usuarios de la base de datos. ‘**Rollback**’ son las transacciones abortadas de manera física (realizadas por el usuario, ya que los abortos de transacciones por fallos se denominan lógicas).

2.SERVER ACTIVITY

La pestaña Sessions nos muestra las transacciones actuales y diversa información ellas. Nos dicen su PID (identificador), User, el usuario que las generó (en nuestro caso somos nosotros los únicos), Application sobre que se realizará la transacción, Backend consiste en cuando se creó el servidor para atender a la base de datos. State nos indica si la transacción esta activa o en espera.

Server activity									
Sessions							Search		
	PID	User	Application	Client	Backend start	State	Wait Event	Blocking PIDs	
+	1656	postgres	pgAdmin 4 - DB:tienda	::1	2020-04-23 18:02:28 CEST	active			
+	9568	postgres	pgAdmin 4 - CONN:1211245	::1	2020-04-23 18:02:43 CEST	idle	Client: ClientRead		

La pestaña de Locks es otra que nos interesa, dado que es la que nos informa de los bloqueos. Postgres controla las transacciones por medio de bloqueos (como si de un semáforo se tratara). Sigue el método FIFO, la primera transacción que llega es la que comienza. Si otra quiere también realizar alguna modificación sobre algún campo que actualmente la transacción anterior está empleando, debe esperar a que esta termine.

Como se comentó, el empleo de bloqueos asegura la consistencia de la base de datos ya que es una manera muy sencilla de manejar las transacciones.

Mode: es el modo de bloqueo. X(Exclusive), C/S(Shared). En este caso nos informa que es un compartido, por lo que la transacción sólo quiere leer (esto lo pueden hacer varias a la vez). Granted nos informa de si está concedida la transacción o no. Observamos que es true, lo que significa que se puede ejecutar, que, coreectamente, observamos que aparece en la pestaña Sessions con un estado activo. La transacción esta concedida por que somos el único ususario de Postgres.

Server activity											
Sessions Locks Prepared Transactions									Q Search		
PID	Lock type	Target relation	Page	Tuple	vXID (target)	XID (target)	Class	Object ID	vXID (owner)	Mode	Granted?
1656	relation	pg_locks							4/8399	AccessShareLock	true

2. Dónde se guarda la información y cómo se muestra

Toda la información de los bloqueos (el elemento que controla las transacciones y concurrencias) se guarda en la vista pg_locks (que almacena todos los bloqueos concedidos y denegados de las transacciones). Para poder visualizar lo que tenemos basta con una sencilla consulta:

```
SELECT * FROM pg_locks;
```

tienda on postgres@PostgreSQL 12															
Query Editor Query History															
1 SELECT * FROM pg_locks;															
2															
Data Output Explain Messages Notifications															
	locktype text	database oid	relation oid	page integer	tuple smallint	virtualxid text	transactionid xid	classid oid	objid oid	objsubid smallint	virtualtransaction text	pid integer	mode text	granted boolean	fastpath boolean
1	relation	32778	12143	[null]	[null]	[null]	[null]			[null]	5/438	9568	Acce...	true	true
2	virtualxid			[null]	[null]	5/438	[null]			[null]	5/438	9568	Excl...	true	true

id	mode text	granted boolean
58	AccessShareLock	true
58	ExclusiveLock	true

Detalle de apreciación de los dos tipos de Locks, el share(compartido) y el exclusivo.

Cuestión 7: Crear dos usuarios en la base de datos que puedan acceder a la base de datos TIENDA identificados como usuario1 y usuario2 que tengan permisos de lectura/escritura a la base de datos tienda, pero que no puedan modificar su estructura. Describir el proceso seguido.

Para poder crear los dos usuarios que nos piden en el enunciado de la práctica, tenemos dos opciones crearlos directamente desde pg admin mediante una consulta o a través del cmd de nuestro ordenador conectándonos a la base de datos.

[Nota: nuestro usuario1 es usuario5, debido a las pruebas que hemos realizado.]

CMD

Primero a través de cmd hemos accedido a Postgres

```
PS C:\Program Files\PostgreSQL\12\bin> .\psql.exe -U postgres
Contraseña para usuario postgres: 
psql (12.1)
ADVERTENCIA: El código de página de la consola (850) difiere del código
de página de Windows (1252).
Los caracteres de 8 bits pueden funcionar incorrectamente.
Vea la página de referencia de psql «Notes for Windows users»
para obtener más detalles.
Digite «help» para obtener ayuda.

postgres=#
```

Crearemos el usuario5 en el gestor de bases de datos

```
postgres=# CREATE USER usuario5 WITH PASSWORD '123';
CREATE ROLE
```

Vamos a conectar al usuario5 con la base de datos que nos interesa, en este caso es 'tienda'.

```
postgres=# GRANT CONNECT ON DATABASE tienda TO usuario5;
GRANT
```

Ahora vamos a dar permisos a las tablas de la base de dato **tienda**, por lo tanto, primero tendremos que acceder a la base de datos mediante \c base_dato. Después concederemos permisos de lectura y escritura a ese usuario. También concederemos permisos de acceso al esquema.

```
postgres=# \c tienda
Ahora está conectado a la base de datos «tienda» con el usuario «postgres».
tienda=# GRAN ALL PRIVILEGES ON ALL TABLES SCHEM PUBLIC TO usuario5;
ERROR:  error de sintaxis en o cerca de «/»
LÍNEA 1: /du
        ^
tienda=# GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO usuario5;
GRANT
tienda=# GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO usuario5;
GRANT
```

Después ejecutaremos un \dp que nos mostrará información de cada tabla junto con el usuario que hemos creado con sus respectivos privilegiados. Como forma de verificación de la correcta creación de usuarios.

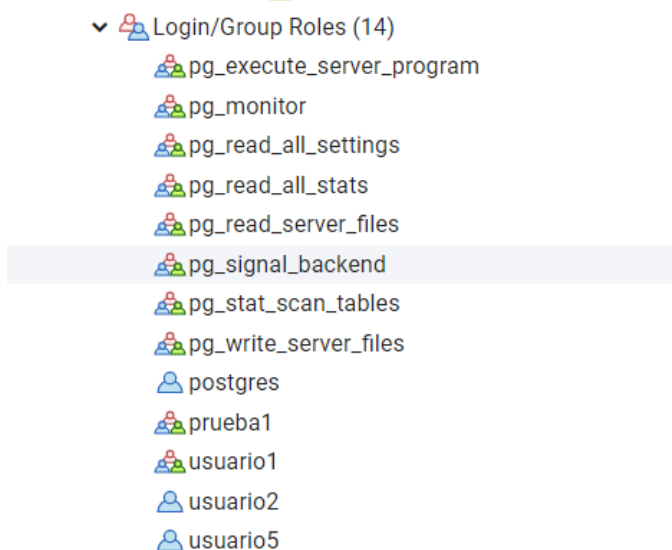
```

tienda=# \dp

```

Esquema	Nombre	Tipo	Privilegios			Políticas
			Privilegios	Privilegios de acceso a columnas		
public	producto	tabla	postgres=arwdDxt/postgres+			
public	ticket	tabla	postgres=arwdDxt/postgres+			
public	tienda	tabla	postgres=arwdDxt/postgres+			
public	tiendaproductos	tabla	postgres=arwdDxt/postgres+			
public	trabajador	tabla	postgres=arwdDxt/postgres+			
(5 filas)			usuario5=arwdDxt/postgres			

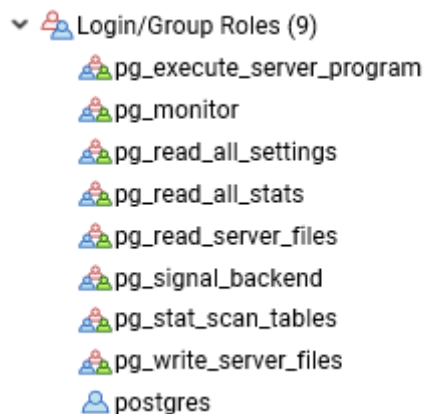
Y como podemos comprobar, aparece correctamente en pgAdmin.



Pg Admin

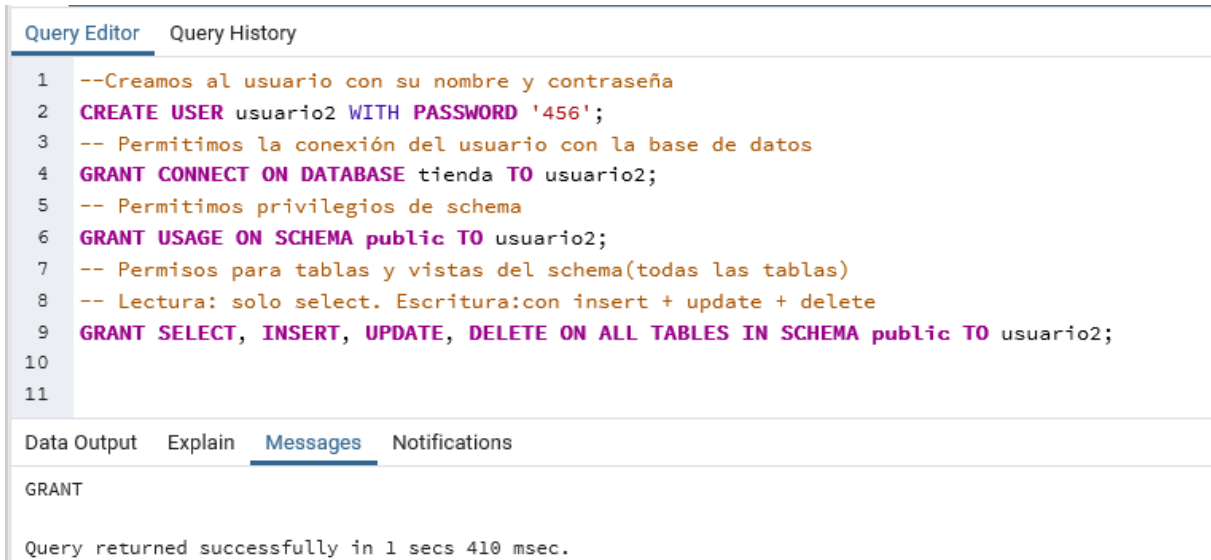
El segundo usuario lo crearemos de otra manera para poder explicar este método también, que es empleando PgAdmin.

Al principio (antes de crear al usuario2) esto es lo que tenemos en los usuarios/roles de grupo:



Si aplicamos la siguiente consulta, podemos crear un usuario con permisos de lectura y escritura sobre todas las tablas de nuestro esquema public, de forma que este usuario podrá hacer un insert, delete, update y select sobre los elementos de las tablas, pero sin modificar su estructura.

```
CREATE USER usuario2 WITH PASSWORD '456';
GRANT CONNECT ON DATABASE tienda TO usuario2;
GRANT USAGE ON SCHEMA public TO usuario2;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public
TO usuario2;
```



The screenshot shows a SQL query editor with a 'Query Editor' tab. The query is as follows:

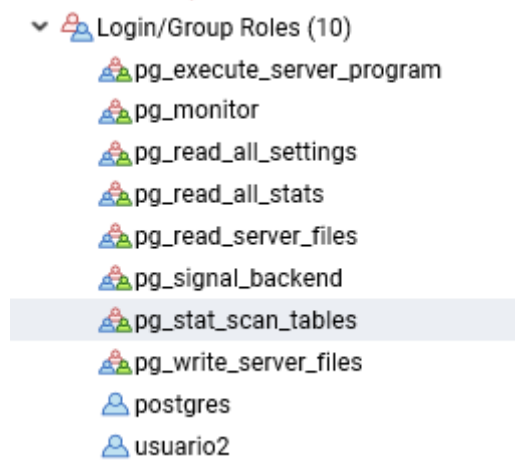
```
1 --Creamos al usuario con su nombre y contraseña
2 CREATE USER usuario2 WITH PASSWORD '456';
3 -- Permitimos la conexión del usuario con la base de datos
4 GRANT CONNECT ON DATABASE tienda TO usuario2;
5 -- Permitimos privilegios de schema
6 GRANT USAGE ON SCHEMA public TO usuario2;
7 -- Permisos para tablas y vistas del schema(todas las tablas)
8 -- Lectura: solo select. Escritura: con insert + update + delete
9 GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO usuario2;
10
11
```

Below the query editor, there are tabs for 'Data Output', 'Explain', 'Messages', and 'Notifications'. The 'Messages' tab is selected, showing the output:

```
GRANT
```

Query returned successfully in 1 secs 410 msec.

Refrescamos nuestra base de datos y observamos al usuario creado correctamente:



Cuestión 8: Abrir una transacción que inserte una nueva tienda en la base de datos (NO cierre la transacción). Realizar una consulta SQL para mostrar todas las tiendas de la base de datos dentro de esa transacción. Consultar la información sobre lo que se encuentra actualmente activo en el sistema. ¿Qué conclusiones se pueden extraer?

1. Abrir transacción e insertar nueva tienda sin cerrarla

Procederemos a realizar la transacción sin cerrarla (NO hacemos commit)

Query Editor

```
1 Begin;
2 Insert into tienda values (123456789, 'TiendaNueva1', 'CiudadNueva1', 'BarrioNuevo1', 'ProvinciaNueva1');
```

Data Output Explain Notifications Query History Messages

INSERT 0 1

Query returned successfully in 84 msec.

2. Consulta -> mostrar las tiendas de la transacción

```
3 select * from tienda
```

Data Output Explain Notifications Query History Messages

	idtienda [PK] integer	nombre text	ciudad text	barrio text	provincia text
1	123456789	TiendaNuev...	CiudadNue...	BarrioNuevo1	ProvinciaNueva1

Cuando realizamos la consulta sobre la tabla tienda aparece lo que hemos insertado porque estamos dentro de la transacción y por lo tanto esos datos están de momento en memoria local y solo es visible por el usuario que ha realizado la transacción.

3. ¿Qué hay activo en el sistema?

Server activity

Sessions Locks Prepared Transactions

Q Search

	PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
✖	13824	postgres	pgAdmin 4 - DB:tienda	::1	2020-05-27 13:23:49 CEST	active		
✖	16452	postgres	pgAdmin 4 - CONN:4748027	::1	2020-05-27 13:23:55 CEST	idle in transaction	Client: ClientRead	

En la pestaña de dashboard en Sessions, observamos que tenemos una transacción activa en tienda que es en la que aún no hemos ejecutado el Commit.

Cuestión 9: Cierre la transacción anterior. Utilizando pgAdmin o psql, abrir una transacción T1 en el usuario1 que realice las siguientes operaciones sobre la base de datos TIENDA. NO termine la transacción.

1.Cerramos la transacción anterior

```
1  --Begin;
2  --Insert into tienda values (123456789, 'TiendaNueva1', 'CiudadNueva1', 'BarrioNuevo1', 'ProvinciaNueva1');
3  commit;
4
```

Data Output Explain Notifications Query History Messages

COMMIT

Query returned successfully in 82 msec.

2.Abrimos nueva transacción

Nota: nuestro usuario1 es usuario5

Vamos a abrir otra transacción que será realizada por el usuario5, que es el que tenemos creado anteriormente del apartado anterior. Primero comprobamos en qué usuario estamos, después cambiaremos al usuario5 y realizaremos de nuevo la misma comprobación para confirmar.

```
1 select current_user
```

Data Output		Explain	Notifications	Query History
	current_user name			
1	postgres			

```
6 set role usuario5;
```

Data Output	Explain	Notifications	Query History	Messages
SET				
Query returned successfully in 110 msec.				

```
5  
6 select current_user
```

Data Output		Explain	Notifications	Query History
	current_user name			
1	usuario5			

Ahora procedemos a realizar la transacción que se nos pide, pero NO la terminaremos:

- Inserte una nueva tienda con ID_TIENDA 1000.
- Inserte un trabajador de la tienda anterior.
- Inserte un nuevo ticket del trabajador anterior con número 54321.

```

1 Begin;
2 insert into tienda values(1000,'TiendaNueva2','CiudadNueva2','BarrioNuevo2','ProvinciaNueva2');
3 insert into trabajador values (123456,'78945612A','Nombre1','Apellidos1','Puesto1',1500,1000);
4 insert into ticket values (54321,'01-01-2020',150,123456);
5
6

```

Data Output Explain Notifications Query History Messages

INSERT 0 1

Query returned successfully in 153 msec.

```

Begin;
Insert into tienda values (1000, 'TiendaNueva2', 'CiudadNueva2',
'BarrioNuevo2', 'ProvinciaNueva2');
Insert into trabajador values (123456, '789456112A', 'Nombre1',
'Apellido1', 'Puesto1', 1500,1000);
Insert into ticket values (54321, '01-01-2020',150, 123456);

```

Como podemos observar, la transacción no lleva el COMMIT, por lo que NO está comprometida, lo que significa que hemos insertado datos, pero Postgres no lo ve, no aparece creado, estos datos están sólo en memoria local de la propia transacción. Para nuestro usuario (usuario5) está, pero para el resto NO. Hasta que no se compromete y sale a memoria global, los datos no son visibles para todos los usuarios de la base de datos.

Cuestión 10: Realizar cualquier consulta SQL que muestre los datos anteriores insertados para ver que todo está correcto.

Desde el Usuario1 (nuestro usuario5)

Observamos que aparecen los resultados de la consulta de los valores que hemos introducido aunque no esté comprometida la transacción, pero hasta que no hagamos COMMIT no se guardarán los datos, dado que de momento sólo están visibles en la **memoria local** de dicha transacción, y es necesario ese COMMIT para comprometerla y sacarla a memoria global los cambios de forma que sean visible a todos los usuarios.

4 --insert into ticket values (54321, '01-01-2020',150, 123456);
5
6 select * from tienda
7

Data Output

Explain

Notifications

Query History

Messages

	idtienda [PK] integer	nombre text	ciudad text	barrio text	provincia text
1	123456789	TiendaNueva1	CiudadNueva1	BarrioNuevo1	ProvinciaNueva1
2	1000	TiendaNueva2	CiudadNueva2	BarrioNuevo2	ProvinciaNueva2

Query Editor

1 --Begin;
2 --Insert into tienda values (1000, 'TiendaNueva2', 'CiudadNueva2', 'BarrioNuevo2', 'ProvinciaNueva2');
3 --Insert into trabajador values (123456, '789456112A', 'Nombre1', 'Apellido1', 'Puesto1', 1500,1000);
4 --Insert into ticket values (54321, '01-01-2020',150, 123456);
5
6 --select * from tienda
7 select * from trabajador
8 --select * from ticket
9

Data Output

Explain

Notifications

Query History

Messages

	codigotrabajador [PK] integer	dni character varying	nombre character varying	apellidos character varying	puesto character varying	salario integer	idtienda integer
1	123456	789456112A	Nombre1	Apellido1	Puesto1	1500	1000

7 --select * from trabajador
8 select * from ticket
9

Data Output

Explain

Notifications

Query History

Messages

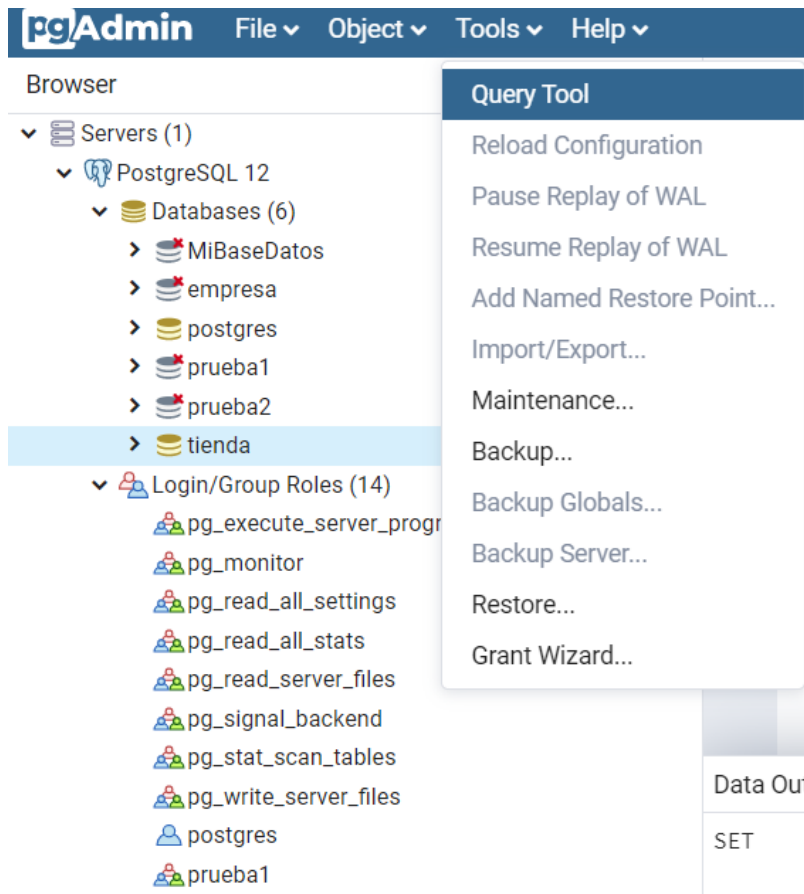
	numticket [PK] integer	fecha text	importe integer	codigotrabajador integer
1	54321	01-01-2020	150	123456

Podemos observar también que las tres transacciones tienen el modo 'ExclusiveLock', esto es debido a que quieren un acceso exclusivo para realizar un write (en este caso un insert). También vemos que en granted -> true, lo que significa que tienen permisos de escritura concedidos. Esto implica que las transacciones de la pregunta 9 están aún esperando un COMMIT o bien un ROLLBACK.

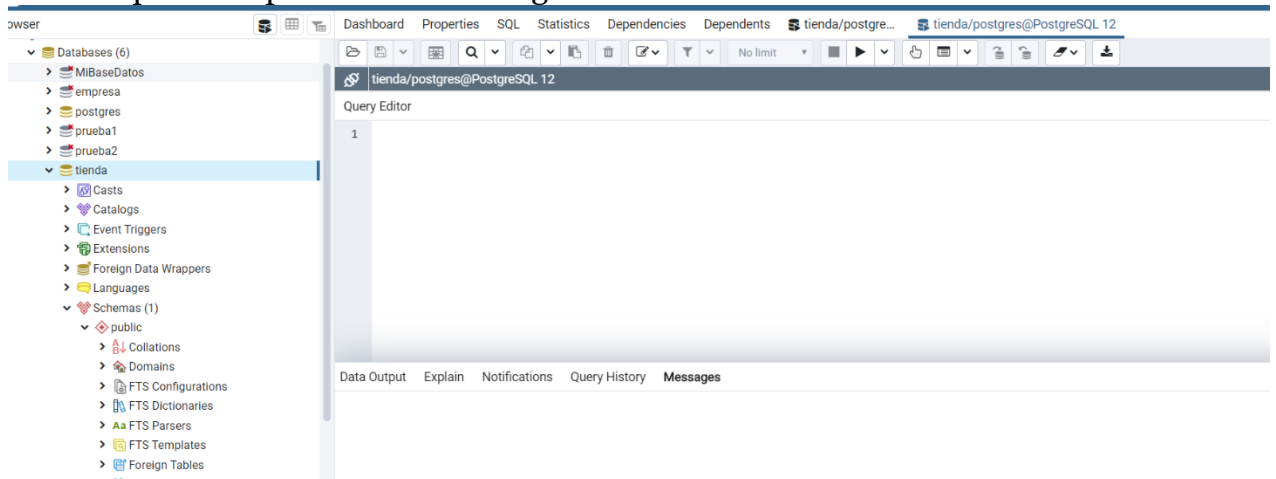
Server activity											
Sessions			Locks			Prepared Transactions			Q Search		
PID	Lock type	Target relation	Page	Tuple	vXID (target)	XID (target)	Class	Object ID	vXID (owner)	Mode	Granted?
13824	relation	pg_locks							7/3530	AccessShareLock	true
16452	relation	pg_class_tblspc_relfilenode_index							6/1700	AccessShareLock	true
16452	relation	pg_class_relnname_nsp_index							6/1700	AccessShareLock	true
16452	relation	pg_class_oid_index							6/1700	AccessShareLock	true
16452	relation	pg_constraint							6/1700	AccessShareLock	true
16452	relation	pg_class							6/1700	AccessShareLock	true
16452	relation	pg_type_typname_nsp_index							6/1700	AccessShareLock	true
16452	relation	pg_type_oid_index							6/1700	AccessShareLock	true
16452	relation	pg_attribute_relid_attnum_index							6/1700	AccessShareLock	true
16452	relation	pg_attribute_relid_attnam_index							6/1700	AccessShareLock	true
16452	relation	pg_type							6/1700	AccessShareLock	true
16452	relation	pg_attribute							6/1700	AccessShareLock	true
16452	relation	tienda_pkey							6/1700	AccessShareLock	true
16452	relation	ticket							6/1700	RowExclusiveLock	true
16452	relation	trabajador							6/1700	RowExclusiveLock	true
16452	relation	tienda							6/1700	AccessShareLock	true
16452	relation	tienda							6/1700	RowExclusiveLock	true

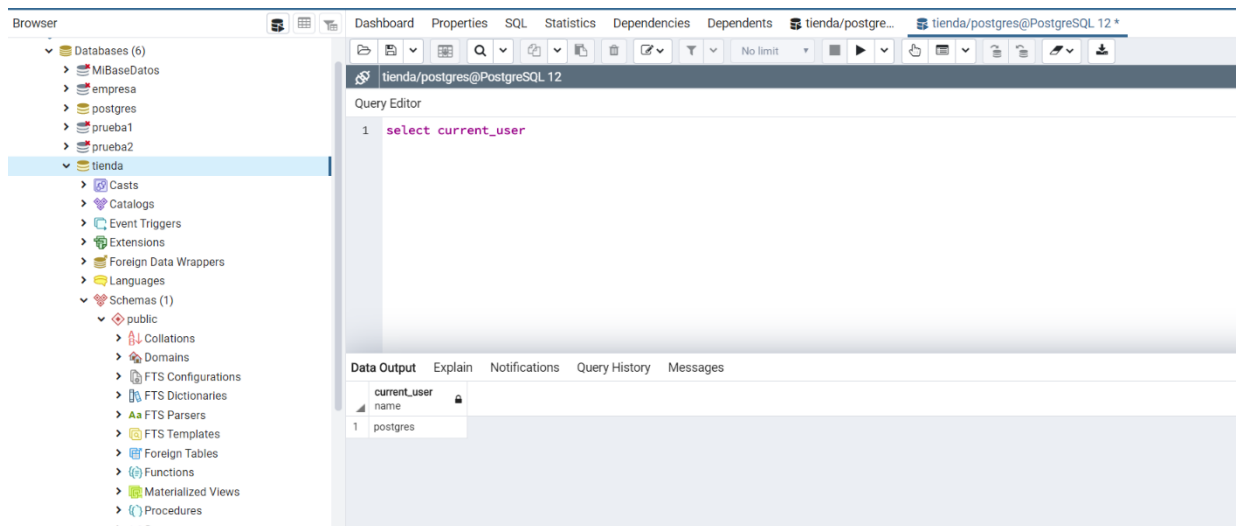
Cuestión 11: Establecer una nueva conexión con pgAdmin o psql a la base de datos con el usuario2 (abrir otra sesión diferente a la abierta actualmente que pertenezca al usuario2) y realizar la misma consulta. ¿Se nota algún cambio? En caso afirmativo, ¿a qué puede ser debido el diferente funcionamiento en la base de datos para ambas consultas? ¿Qué información de actividad hay registrada en la base de datos en este momento?

Primero abriremos una nueva sesión para que otro usuario acceda a la base de datos, mediante la consola:

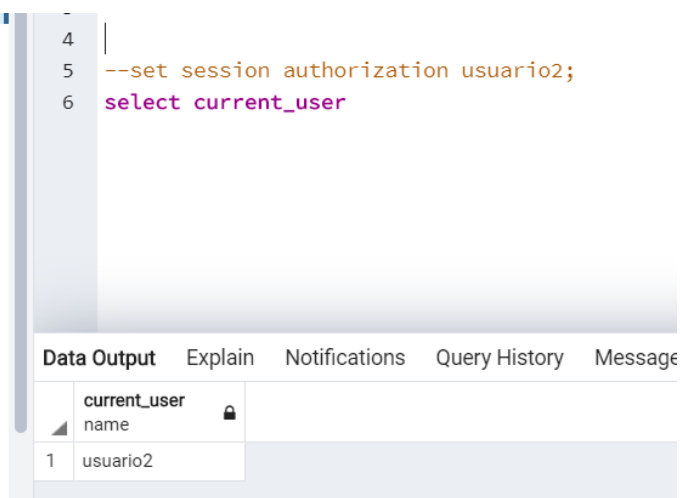
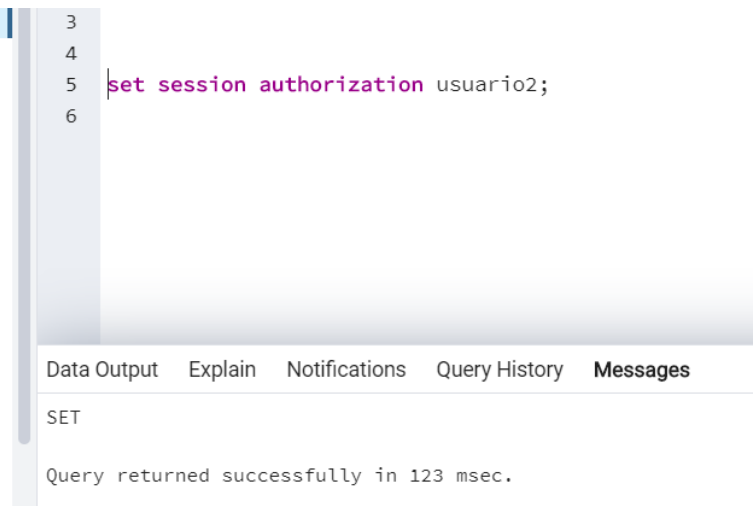


Ahora con dos sesiones activas sobre la misma base de datos. En la nueva sesión el usuario que viene por defecto es Postgres.





Cambiamos el usuario que viene por defecto a usuario2 y lo comprobamos:



Procedemos a hacer repetir las mismas consultas en tienda, trabajador y ticket:

Dashboard Properties SQL Statistics Dependencies Dependents tienda/postgre... tienda/postgres@PostgreSQL 12 *

tienda/postgres@PostgreSQL 12

Query Editor

```
1
2 select * from tienda
3
4
5
6 --set session authorization usuario2;
7 s--elect current_user;
```

Data Output Explain Notifications Query History Messages

	idtienda [PK] integer	nombre text	ciudad text	barrio text	provincia text
1	123456789	TiendaNueva1	CiudadNueva1	BarrioNuevo1	ProvinciaNueva1

```
1
2 --select * from tienda
3 select * from trabajador
4 --select * from ticket
5
6
7 --set session authorization usuario2;
8 s--elect current_user;
```

Data Output Explain Notifications Query History Messages

	codigotrabajador [PK] integer	dni character varying	nombre character varying	apellidos character varying	puesto character varying	salario integer	idtienda integer

tienda/postgres@PostgreSQL 12

Query Editor

```

1
2  --select * from tienda
3  --select * from trabajador
4  select * from ticket
5
6
7  --set session authorization usuario2;
8  select current_user;

```

Data Output Explain Notifications Query History Messages

numticket [PK] integer	fecha text	importe integer	codigotrabajador integer

Como vemos el usuario2 no es capaz de visualizar lo que ha hecho usuario5 ya que todavía no ha terminado la transacción T1 con un COMMIT (por lo que sólo está en la memoria local de esa transacción como se comentó anteriormente) y así poder hacer que aparezca esos datos reflejados en la base de datos tienda y visible para otros usuarios.

RESUMEN:

Usuario1(Usuario5)->Inicia transacción sin comprometerla. Sólo es visible en su memoria local, por eso desde ese usuario podemos ver las salidas de la consulta de select.

Usuario2 -> hace la consulta de select y no ve nada. Esto se debe a que para ella no existe lo que ha hecho T1, hasta que T1 haga commit y así pase a memoria global donde todos los usuarios de la base de datos puedan verlo.

Cuestión 12: ¿Se encuentran los nuevos datos físicamente en las tablas de la base de datos? Entonces, ¿de dónde se obtienen los datos de la cuestión 2.10 y/o de la 2.11?

Los datos insertados en el ejercicio 9 por el usuario1 (nuestro usuario5) están sólo en la memoria local de esa transacción y de ese usuario, por eso si se realiza el select se pueden ver. Hasta que no se comprometa con un COMMIT, la transacción no finalizará y los datos modificados no saldrán a memoria global, por lo que hasta ese momento nadie podrá ver esas modificaciones. Esto se debe a que Postgres emplea un esquema de recuperación con modificación diferida, y los datos sólo serán visibles con un commit, al contrario que, si fuera inmediato, que saldría inmediatamente a global con realizar un write.

Los datos de la cuestión 10 se obtienen de la memoria local de T1 y la cuestión 11 obtiene los datos (que en verdad no obtiene) de la memoria global.

Cuestión 13: Finalizar con éxito la transacción T1 y realizar la consulta de la cuestión 2.10 y 2.11 sobre ambos usuarios conectados. ¿Qué es lo que se obtiene ahora? ¿Por qué?

Finalizamos la T1 realizada por el usuario1 (nuestro usuario5) y repetimos las consultas anteriores.

Dashboard Properties SQL Statistics Dependencies Dependents tienda/postgres@PostgreSQL 12 * tienda/postgre...

tienda/postgres@PostgreSQL 12

Query Editor

```
1 begin;
2 insert into tienda values(1000,'TiendaNueva2','CiudadNueva2','BarrioNuevo2','ProvinciaNueva2');
3 insert into trabajador values(123456,'78945612A','Nombre1','Apellido1','Puesto1',1500,1000);
4 insert into ticket values(54321,'01-01-2020',150,123456);
5 commit;
6 --select current_user;
7 |
```

Data Output Explain Notifications Query History Messages

COMMIT

Query returned successfully in 100 msec.

Usuario1 (Nuestro Usuario5)

```
5
6
7 select * from tienda
8
9
10
11
12
13
14
```

Data Output Explain Notifications Query History Messages

	idtienda [PK] integer	nombre text	ciudad text	barrio text	provincia text
1	123456789	TiendaNueva1	CiudadNueva1	BarrioNuevo1	ProvinciaNueva1
2	1000	TiendaNueva2	CiudadNueva2	BarrioNuevo2	ProvinciaNueva2


```
4
5
6 select * from trabajador
7
8
9
10
11
12
13
14
```

Data Output Explain Notifications Query History Messages

	codigotrabajador [PK] integer	dni character varying	nombre character varying	apellidos character varying	puesto character varying	salario integer	idtienda integer
1	123456	78945612A	Nombre1	Apellido1	Puesto1	1500	1000

```
4
5
6 select * from ticket
7
8
9
10
11
12
13
14
```

Data Output Explain Notifications Query History Messages

	numticket [PK] integer	fecha text	importe integer	codigotrabajador integer
1	54321	01-01-2020	150	123456

Usuario2

Dashboard Properties SQL Statistics Dependencies Dependents tienda/postgre... tienda/postgres@PostgreSQL 12 *

tienda/postgres@PostgreSQL 12









Query Editor

```
1
2 select * from tienda
3 --select * from trabajador
4 --select * from ticket
5
6
7 --set session authorization usuario2;
8 --elect current_user;
```

Data Output Explain Notifications Query History Messages

	idtienda [PK] integer	nombre text	ciudad text	barrio text	provincia text
1	123456789	TiendaNueva1	CiudadNueva1	BarrioNuevo1	ProvinciaNueva1
2	1000	TiendaNueva2	CiudadNueva2	BarrioNuevo2	ProvinciaNueva2

```
1
2 --select * from tienda
3 select * from trabajador
4 --select * from ticket
5
6
7 --set session authorization usuario2;
8 --select current_user;
9
```

Data Output		Explain	Notifications	Query History	Messages		
 codigotrabajador [PK] integer	 dni character varying	 nombre character vary	 apellidos character varying	 puesto character varying	 salario integer	 idtienda integer	
1	123456	789456112A	Nombre1	Apellido1	Puesto1	1500	1000

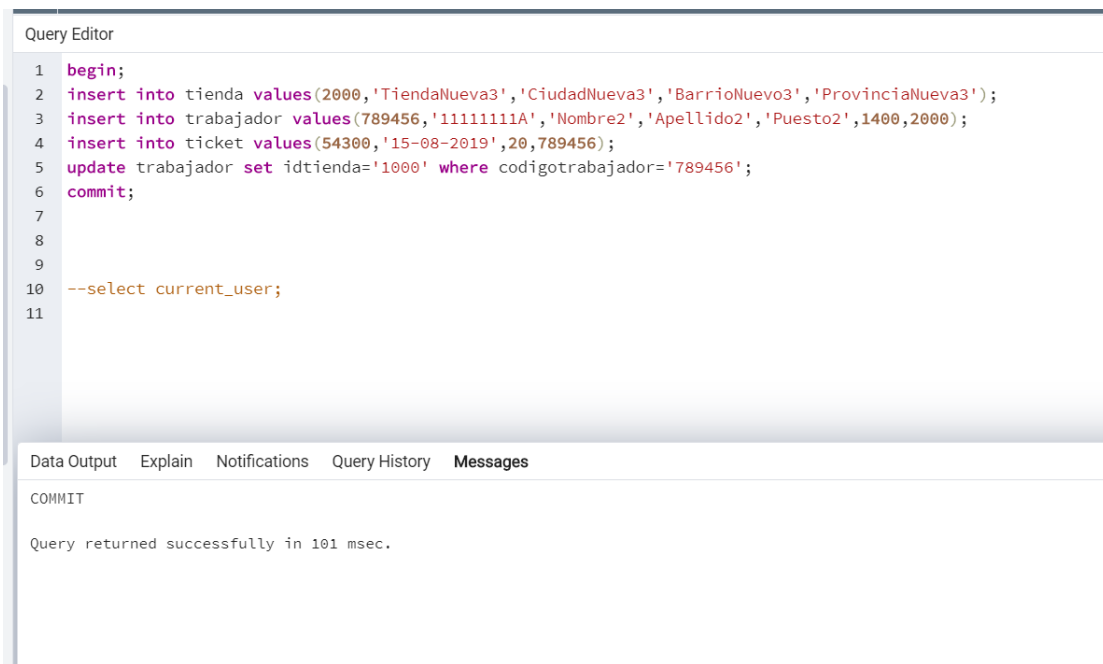
```
1
2 --select * from tienda
3 --select * from trabajador
4 select * from ticket
5
6
7 --set session authorization usuario2;
8 --select current_user;|
9
```

Data Output		Explain	Notifications	Query History	Messages
	numticket [PK] integer		fecha text	importe integer	codigotrabajador integer
1	54321		01-01-2020	150	123456

Como vemos una vez finalizada la transacción T1 y comprometida con commit, las modificaciones en la base de datos han salido a memoria global y ya son visibles para todos los usuarios de la base de datos. Es decir, están actualizadas con los datos que le hemos insertado.

Cuestión 14: Sin ninguna transacción en curso, abrir una transacción en un usuario cualquiera y realizar las siguientes operaciones:

- Insertar una tienda nueva con ID_TIENDA a 2000.
- Insertar un trabajador de la tienda 2000.
- Insertar un ticket del trabajador anterior con número 54300.
- Hacer una modificación del trabajador para cambiar el número de tienda de 2000 a 1000.
- Cerrar la transacción.



The screenshot shows a SQL Query Editor window. The top pane contains a SQL script with 11 lines of code. The bottom pane shows the output of the query, which is 'COMMIT' followed by a message: 'Query returned successfully in 101 msec.' The tabs at the bottom of the editor are 'Data Output', 'Explain', 'Notifications', 'Query History', and 'Messages'.

```
1 begin;
2 insert into tienda values(2000,'TiendaNueva3','CiudadNueva3','BarrioNuevo3','ProvinciaNueva3');
3 insert into trabajador values(789456,'11111111A','Nombre2','Apellido2','Puesto2',1400,2000);
4 insert into ticket values(54300,'15-08-2019',20,789456);
5 update trabajador set idtienda='1000' where codigotrabajador='789456';
6 commit;
7
8
9
10 --select current_user;
11
```

COMMIT

Query returned successfully in 101 msec.

```
begin;
insert into tienda
values(2000,'TiendaNueva3','CiudadNueva3','BarrioNuevo3','Provincia
Nueva3');
insert into trabajador
values(789456,'11111111A','Nombre2','Apellido2','Puesto2',1400,2000
);
insert into ticket values(54300,'15-08-2019',20,789456);
update trabajador set idtienda='1000' where
codigotrabajador='789456';
commit;
```

¿Cuál es el estado final de la base de datos? ¿Por qué?

Cuando realizamos la transacción vemos que se ejecuta sin ningún problema y que se actualiza el dato correctamente como observamos en la imagen de abajo, no hay ningún tipo de problema de concurrencia.

```

7
8  select * from trabajador
9
10 --select current_user;
11

```

Data Output
Explain
Notifications
Query History
Messages

	codigotrabajador [PK] integer	dni character varying	nombre character varying	apellidos character varying	puesto character varying	salario integer	idtienda integer
1	123456	78945612A	Nombre1	Apellido1	Puesto1	1500	1000
2	789456	11111111A	Nombre2	Apellido2	Puesto2	1400	1000

Cuestión 15: Repetir la cuestión 9 con otra tienda, trabajador y ticket. Realizar la misma consulta de la cuestión 10, pero ahora terminar la transacción con un ROLLBACK y repetir la consulta con los mismos dos usuarios. ¿Cuál es el resultado? ¿Por qué?

1.Repetimos transacción del ejercicio 9

Ahora procedemos a realizar la transacción que se nos pide, pero finalizando con un ROLLBACK:

- Inserte una nueva tienda con ID_TIENDA 3000.
- Inserte un trabajador de la tienda anterior. (ID_ TRABAJADOR 7891011)
- Inserte un nuevo ticket del trabajador anterior con número 98765.

Query Editor

```

1 begin;
2 insert into tienda values(3000,'TiendaNueva4','CiudadNueva4','BarrioNuevo4','ProvinciaNueva4');
3 insert into trabajador values(7891011,'222222A','Nombre4','Apellido4','Puesto4',2500,3000);
4 insert into ticket values(98765,'11-05-2019',100,7891011);
5 rollback;
6
7 --select current_user

```

Data Output Explain Notifications Query History Messages

ROLLBACK

Query returned successfully in 88 msec.

```

Begin;
insert into tienda
values(3000,'TiendaNueva4','CiudadNueva4','BarrioNuevo4','ProvinciaNueva4');
insert into trabajador
values(7891011,'222222A','Nombre4','Apellido4','Puesto4',2500,3000);
insert into ticket values(98765,'11-05-2019',100,7891011);
rollback;

```

2.Repetimos consulta del ejercicio 10

Usuario1 (Usuario5):

Query Editor









```






1 --begin;
2 --insert into tienda values(3000,'TiendaNueva4','CiudadNueva4','BarrioNuevo4','ProvinciaNueva4');
3 --insert into trabajador values(7891011,'222222A','Nombre4','Apellido4','Puesto4',2500,3000);
4 --insert into ticket values(98765,'11-05-2019',100,7891011);
5 --rollback;
6
7 --select current_user
8 select * from tienda
9 --select * from trabajador
10 --select * from ticket

```

Data Output Explain Notifications Query History Messages

	idtienda [PK] integer	nombre text	ciudad text	barrio text	provincia text
1	123456789	TiendaNueva1	CiudadNueva1	BarrioNuevo1	ProvinciaNueva1
2	1000	TiendaNueva2	CiudadNueva2	BarrioNuevo2	ProvinciaNueva2
3	2000	TiendaNueva3	CiudadNueva3	BarrioNuevo3	ProvinciaNueva3

Data Output		Explain	Notifications	Query History	Messages				
	 codigotrabajador [PK] integer	 dni character varying	 nombre character varying	 apellidos character varying	 puesto character varying	 salario integer	 idtienda integer		
1		123456	789456112A	Nombre1	Apellido1	Puesto1	1500	1000	
2		789456	11111111A	Nombre2	Apellido2	Puesto2	1400	1000	

Data Output		Explain	Notifications	Query History	Messages
	numticket [PK] integer 		fecha text 	importe integer 	codigotrabajador integer 
1	54321		01-01-2020	150	123456
2	54300		15-08-2019	20	789456

Data Output	Explain	Notifications	Query History	Messages	
idtienda [PK] integer	nombre text	ciudad text	barrio text	provincia text	
1	123456789	TiendaNueva1	CiudadNueva1	BarrioNuevo1	ProvinciaNueva1
2	1000	TiendaNueva2	CiudadNueva2	BarrioNuevo2	ProvinciaNueva2
3	2000	TiendaNueva3	CiudadNueva3	BarrioNuevo3	ProvinciaNueva3

```

4  --select * from tienda;
5  select * from trabajador;
6  --select * from ticket;

```

Data Output		Explain	Notifications	Query History	Messages										
	codigotrabajador [PK] integer		dni character varying		nombre character varying		apellidos character varying		puesto character varying		salario integer		idtienda integer		
1			123456		789456112A		Nombre1		Apellido1		Puesto1		1500		1000
2			789456		11111111A		Nombre2		Apellido2		Puesto2		1400		1000

```

1  --set session authorization usuario2;
2  --select current_user;
3
4  --select * from tienda;
5  --select * from trabajador;
6  select * from ticket;

```

Data Output	Explain	Notifications	Query History	Messages
	numticket [PK] integer	fecha text	importe integer	codigotrabajador integer
1	54321	01-01-2020	150	123456
2	54300	15-08-2019	20	789456

Estamos creando la transacción T1', pero esta vez en vez de comprometerla, realizaremos un rollback. Esto implica que esta transacción nunca saldrá de la memoria local, ya que antes de salir la desharemos. Rollback deshace la transacción actual y descarta todas las actualizaciones que se han producido en ella.

Por eso mismo es independiente del usuario, el resultado de las consultas será el mismo, no se mostrará lo que hemos realizado.

Cuestión 16: Cerrar todas las sesiones anteriores. Abrir una sesión con el usuario1 de la base de datos TIENDA. Insertar la siguiente información en la base de datos:

- Insertar una tienda con id_tienda de 31145.

- Insertar un trabajador que pertenezca a la tienda anterior y tenga un código de 45678.

Desde el ususario1 (nuestro usuario5) e insertamos:

```
1
2 insert into tienda values(31145,'TiendaNueva5','CiudadNueva5','BarrioNuevo5','ProvinciaNueva5');
3 insert into trabajador values(45678,'33333333A','Nombre5','Apellido5','Puesto5',2800,31145);
4
5
6
7
8
```

Explain Data Output Notifications Query History Messages

INSERT 0 1

Query returned successfully in 78 msec.

```
--begin
insert into tienda
values(31145,'TiendaNueva5','CiudadNueva5','BarrioNuevo5','Provincia
Nueva5');
insert into trabajador
values(45678,'33333333A','Nombre5','Apellido5','Puesto5',2800,31145)
;
--commit
```

Cuestión 17: Abrir una sesión con el usuario2 a la base de datos TIENDA. Abrir una transacción T2 en este usuario2 y realizar una modificación de la tienda código 31145 para cambiar el nombre a “Tienda Alcalá”. ¿Qué actividad hay registrada en la base de datos? ¿Cuál es la información guardada en la base de datos? ¿Por qué?

Cambiamos a usuario2 y abrimos una transacción T2 (sin cerrarla, al igual que en el ejercicio 8). Modificamos la tienda con ID_TIENDA 31145 y cambiamos el nombre a “Tienda Alcalá”

```

1 begin;
2 update tienda set nombre='Tienda Alcalá' where idtienda='31145'
3 |
4
5 --set session authorization usuario2;
6 --select current_user;

```

Explain Data Output Notifications Query History Messages

UPDATE 1

Query returned successfully in 85 msec.

```

begin;
update tienda set nombre='Tienda Alcalá' where idtienda='31145'

```

Actividad en la base de datos

tienda/postgres@PostgreSQL 12

Query Editor

```

1 --begin;
2 --update tienda set nombre='Tienda Alcalá' where idtienda='31145'
3 select * from tienda
4
5 --set session authorization usuario2;
6 --select current_user;

```

Explain Data Output Notifications Query History Messages

	idtienda [PK] integer	nombre text	ciudad text	barrio text	provincia text
1	123456789	TiendaNueva1	CiudadNueva1	BarrioNuevo1	ProvinciaNueva1
2	1000	TiendaNueva2	CiudadNueva2	BarrioNuevo2	ProvinciaNueva2
3	2000	TiendaNueva3	CiudadNueva3	BarrioNuevo3	ProvinciaNueva3
4	31145	Tienda Alcalá	CiudadNueva5	BarrioNuevo5	ProvinciaNueva5

La información guardada en la base de datos sólo se encuentra en la memoria local de la T2 puesto que no se ha comprometido y no ha salido a memoria global para poder ser vista por todos los usuarios, por lo que sólo será visible para el usuario2. El PID de esta transacción es 8480. La transacción T1 tiene el PID 6016 (se creó al cambiar al usuario1/nuestro usuario5). Esta transacción es una exclusiva, por que pide escribir en la base de datos (update).

Server activity											
Sessions									Search		
		PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs		
+	■	6016	postgres	pgAdmin 4 - CONN:5380426	::1	2020-05-28 11:29:31 CEST	idle	Client: ClientRead			
+	■	8480	postgres	pgAdmin 4 - CONN:3489704	::1	2020-05-28 11:20:07 CEST	idle in transaction	Client: ClientRead			

Details											
Backend type		client backend									
Query started at		2020-05-28 11:29:19 CEST									
Last state changed at		2020-05-28 11:29:19 CEST									
SQL		<pre> 1 begin; 2 update tienda set nombre='Tienda Alcalá' where idtienda='31145' 3 4 5 </pre>									

Server activity											
Locks									Search		
PID	Lock type	Target relation	Page	Tuple	vXID (target)	XID (target)	Class	Object ID	vXID (owner)	Mode	Granted?
8480	relation	tienda_pkey							6/1801	RowExclusiveLock	true
8480	relation	tienda							6/1801	RowExclusiveLock	true
13824	relation	pg_locks							7/11519	AccessShareLock	true

Cuestión 18. Abra una transacción T1 en el usuario1. Haga una actualización del trabajador con número 45678 para cambiar el salario a 3000. ¿Qué actividad hay registrada en la base de datos? ¿Cuál es la información guardada en la base de datos? ¿Por qué?

Cambiamos a usuario1 (nuestro usuario5) y abrimos una transacción T1 (sin cerrarla, al igual que en el ejercicio 8). Actualizamos el trabajador ID_TRABAJADOR 45678 y su salario a 3000.

1	begin;
2	update trabajador set salario='3000' where codigotrabajador='45678'
3	

Explain	Data Output	Notifications	Query History	Messages
UPDATE 1				
Query returned successfully in 84 msec.				

```

begin;
update trabajador set salario='3000' where codigotrabajador='45678'

```

Al igual que en el ejercicio anterior, esta transacción se queda en la memoria local de la T1 y sólo esta visible para esta, hasta que se comprometa con commit y salga a memoria global. El PID de la T1 es 6016. Esta transacción es una exclusiva ya que pide escribir un dato.

```

1 select * from trabajador
2 --begin;
3 --update trabajador set salario='3000' where codigotrabajador='45678'
4

```

Explain Data Output Notifications Query History Messages

	codigotrabajador [PK] integer	dni character varying	nombre character varying	apellidos character varying	puesto character varying	salario integer	idtienda integer
1	123456	789456112A	Nombre1	Apellido1	Puesto1	1500	1000
2	789456	11111111A	Nombre2	Apellido2	Puesto2	1400	1000
3	7891011	222222A	Nombre4	Apellido4	Puesto4	2500	31145
4	45678	33333333A	Nombre5	Apellido5	Puesto5	3000	31145

Server activity								
Sessions				Locks		Prepared Transactions		
	PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
+	6016	postgres	pgAdmin 4 - CONN:5380426	::1	2020-05-28 11:29:31 CEST	idle in transaction	Client: ClientRead	
Details								
Backend type		client backend						
Query started at		2020-05-28 11:36:54 CEST						
Last state changed at		2020-05-28 11:36:54 CEST						
SQL		<pre> 1 begin; 2 update trabajador set salario='3000' where codigotrabajador='45678' </pre>						
+	8480	postgres	pgAdmin 4 - CONN:3489704	::1	2020-05-28 11:20:07 CEST	idle in transaction	Client: ClientRead	
+	13824	postgres	pgAdmin 4 - DB:tienda	::1	2020-05-27 13:23:49 CEST	active		

Server activity											
Sessions				Locks		Prepared Transactions					
PID	Lock type	Target relation	Page	Tuple	vXID (target)	XID (target)	Class	Object ID	vXID (owner)	Mode	Granted?
6016	relation	trabajador_pkey							8/1724	RowExclusiveLock	true
6016	relation	trabajador							8/1724	RowExclusiveLock	true
8480	relation	tienda_pkey							6/1801	RowExclusiveLock	true
8480	relation	tienda							6/1801	RowExclusiveLock	true
13824	relation	pg_locks							7/11863	AccessShareLock	true

Cuestión 19: En la transacción T2, realice una modificación del trabajador con código 45678 para cambiar el puesto a “Capataz”. ¿Qué actividad hay registrada en la base de datos? ¿Cuál es la información guardada en la base de datos? ¿Por qué?

Seguimos con el usuario1 (nuestro usuario5) y en la transacción T2 (sin cerrarla, al igual que en el ejercicio 8). Modificamos el trabajador con ID_TRABAJADOR 45678 y cambiamos el puesto a ‘Capataz’.

```

1  --begin;
2  update trabajador set puesto='Capataz' where idtienda='45678'
3  --select * from trabajador
4
5  --set session authorization usuario2;
6  --select current_user;

```

Explain Data Output Notifications Query History Messages

UPDATE 0

Query returned successfully in 78 msec.

update trabajador set puesto='Capataz' where idtienda='45678'

Server activity											
Sessions Locks Prepared Transactions										Q Search	↺
PID	Lock type	Target relation	Page	Tuple	vXID (target)	XID (target)	Class	Object ID	vXID (owner)	Mode	Granted?
6016	relation	trabajador_pkey							8/1724	RowExclusiveLock	true
6016	relation	trabajador							8/1724	RowExclusiveLock	true
8480	relation	tienda_pkey							6/1801	RowExclusiveLock	true
8480	relation	trabajador_pkey							6/1801	RowExclusiveLock	true
8480	relation	tienda							6/1801	RowExclusiveLock	true
8480	relation	trabajador							6/1801	RowExclusiveLock	true
13824	relation	pg_locks							7/12532	AccessShareLock	true

Como podemos observar, tenemos 'update 0' por lo que realmente NO se ha realizado el update, ni siquiera para la propia memoria local. Esto se debe a que esta T1 quiere hacer un write (update) en la tabla trabajador, de la misma manera que quiere T2, otro write(update) sobre la tabla trabajador. Se produce un conflicto entre estas 2 transacciones dado que T1 pide un bloqueo después de T2, pero T1 solicita un bloqueo antes que T2 sobre la tabla trabajadores, (todos bloqueos exclusivos). Se concede a T2 el primer update de tienda y a T1 el segundo de trabajador, pero a T2 ahora debe esperar a que T1 libere el bloqueo concedido que tiene sobre la tabla trabajador (commit), hasta entonces, tendrá que esperar.

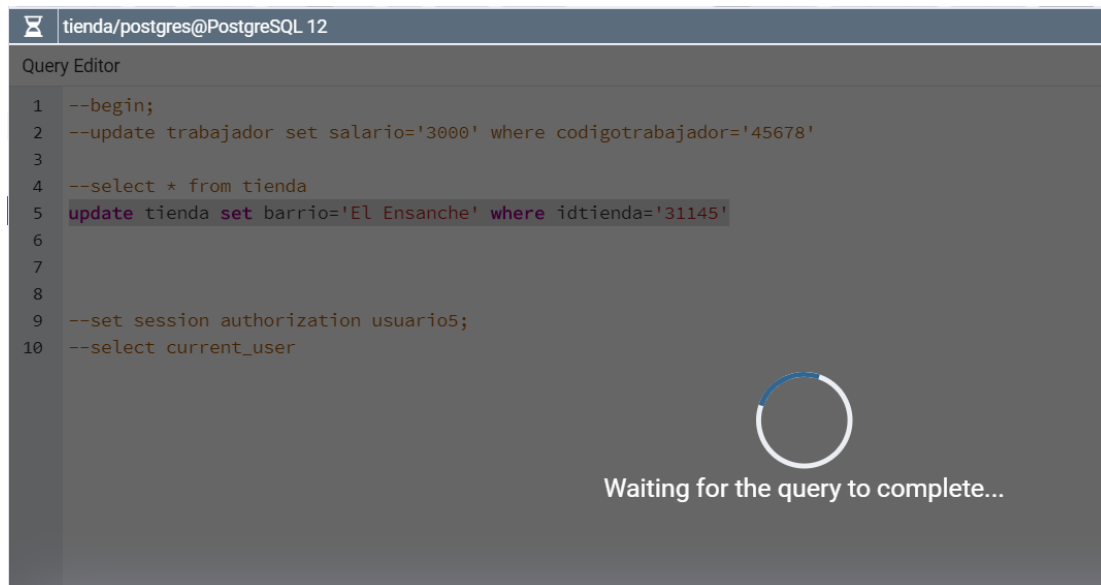
Instantes	T2	T1	
0	Beggin; Update tienda		Siguiendo el protocolo de 2 fases: Para T2 se solicitan los bloqueos en crecimiento sobre tienda y trabajador (Exclusivos). Para T1 se solicitan también exclusivos sobre trabajador y tienda. Situación actual -> T2 espera a T1 (no hay interbloqueo aún porque T1 no depende/espera a T2)
1		Beggin; Update trabajador	
2	Update trabajador		

Cuestión 20: En la transacción T1, realice una modificación de la tienda con código 31145 para modificar el barrio y poner “El Ensanche”. ¿Qué actividad hay registrada en la base de datos? ¿Cuál es la información guardada en la base de datos? ¿Por qué?

Cambiamos a usuario1 (nuestro usuario5) y abrimos una transacción T1 (sin cerrarla, al igual que en el ejercicio 8). Actualizamos el barrio de la tienda con código 31145 y su nombre ‘El Ensanche’

```
update tienda set barrio='El Ensanche' where idtienda='31145'
```

Como se aprecia, obtenemos una pantalla en la que Postgres está ‘pensando’ porque sucede un interbloqueo y tiene que analizar y decidir qué transacción deshacer.



Instantes	T2	T1	Situación actual -> observamos que ahora SÍ se ha producido un interbloqueo o deadlock. T2 espera a T1 en el instante 2 para modificar la tabla trabajador, pero T1 espera a su vez a que T2 termine para poder modificar la tabla tienda. En esta situación lo que hay que hacer es liberar el interbloqueo. En nuestro caso, sería conveniente deshacer (rollback) la más reciente, o sea, T1 (conservando la primera que llegó).
0	Beggin; Update tienda		
1		Beggin; Update trabajador	
2	Update trabajador		
3		Update tienda	

Grafo de espera:



El interbloqueo o deadlock se produce por que estas dos transacciones se están esperando a sí mismas. Las transacciones NUNCA avanzarán y el sistema colapsará hasta que se decida cuál desharemos.

Cuestión 21: Comprometa ambas transacciones T1 y T2. ¿Cuál es el valor final de la información modificada en la base de datos TIENDA? ¿Por qué?

Usuario2 – T2

```
1 --begin;
2 --update tienda set nombre='Tienda Alcalá' where idtienda='31145'
3 --update trabajador set puesto='Capataz' where idtienda='45678'
4 commit;
5 --set session authorization usuario2;
6 --select current_user;
```

Explain Data Output Notifications Query History Messages

COMMIT

Query returned successfully in 136 msec.

Usuario1(Usuario5) – T1

Query Editor

```
1 --begin;
2 --update trabajador set salario='3000' where codigotrabajador='45678'
3
4 --update tienda set barrio='El Ensanche' where idtienda='31145'
5 commit;
6 --set session authorization usuario5;
7 --select current_user;
```

Explain Data Output Notifications Query History Messages

ROLLBACK

Query returned successfully in 124 msec.

Al comprometer ambas transacciones, vemos que **T1** realizada por el usuario1(en nuestro caso **usuario5**) aparece un rollback esto se debe a que postgres ha deshecho esa transacción (por ser la más reciente) y cuando sucede esto significa que cualquier

Se ha intentado realizar el update (write) en la tabla trabajador, pero no se ha podido debido a que en ese momento **T1** (ya que lo solicito antes que **T2**) tenía un permiso exclusivo sobre esos datos y por lo tanto el cambio no se ha podido realizar.

```
1 --begin;
2 --update tienda set nombre='Tienda Alcalá' where idtienda='31145'
3 --update trabajador set puesto='Capataz' where idtienda='45678'
4 --commit;
5 --set session authorization usuario2;
6 --select current_user;
7 select * from tienda
```

Explain	Data Output	Notifications	Query History	Messages		
<div><div><div></div></div></div> <div>idtienda</div> <div>[PK] integer</div> <div><div></div></div>	<div><div><div></div></div></div> <div>nombre</div> <div>text</div> <div><div></div></div>	<div><div><div></div></div></div> <div>ciudad</div> <div>text</div> <div><div></div></div>	<div><div><div></div></div></div> <div>barrio</div> <div>text</div> <div><div></div></div>	<div><div><div></div></div></div> <div>provincia</div> <div>text</div> <div><div></div></div>		
1	123456789	TiendaNueva1	CiudadNueva1	BarrioNuevo1	ProvinciaNueva1	
2	1000	TiendaNueva2	CiudadNueva2	BarrioNuevo2	ProvinciaNueva2	
3	2000	TiendaNueva3	CiudadNueva3	BarrioNuevo3	ProvinciaNueva3	
4	31145	Tienda Alcalá	CiudadNueva5	BarrioNuevo5	ProvinciaNueva5	

```
1 --begin;
2 --update tienda set nombre='Tienda Alcalá' where idtienda='31145';
3 --update trabajador set puestos='Capataz' where idtienda='45678';
4 --commit;
5 --set session authorization usuario2;
6 --select current_user;
7 select * from trabajador
```

Explain	Data Output	Notifications	Query History	Messages						
	codigotrabajador [PK] integer	dni character varying	nombre character varying	apellidos character varying	puesto character varying	salario integer	idtienda integer			
1	123456	789456112A	Nombre1	Apellido1	Puesto1	1500	1000			
2	789456	11111111A	Nombre2	Apellido2	Puesto2	1400	1000			
3	7891011	222222A	Nombre4	Apellido4	Puesto4	2500	31145			
4	45678	33333333A	Nombre5	Apellido5	Puesto5	2800	31145			

Cuestión 22: Cerrar todas las sesiones anteriores. Abrir una sesión con el usuario1 de la base de datos TIENDA. Insertar en la tabla tienda una nueva tienda con código 6789. Abrir una transacción T1 en este usuario y realizar una modificación de la tienda con código 6789 y actualizar el nombre a “Mediamarkt”. No cierre la transacción.

1.Abrimos sesión con el usuario1 (nuestro usuario5) e insertamos en la tabla tienda una nueva con ID TIENDA 6789 (INSERT normal, con begin y commit).

```
Insert into tienda
values(6789,'TiendaNueva7','CiudadNueva7','BarrioNuevo7','ProvinciaNueva7');
```

```
1 insert into tienda values(6789,'TiendaNueva7','CiudadNueva7','BarrioNuevo7','ProvinciaNueva7');
2 --set session authorization usuario5;
3 --select current_user;
```

Explain Data Output Notifications Query History Messages

INSERT 0 1

Query returned successfully in 136 msec.

2.Abrimos una transacción T1 nueva con el usuario1/usuario5 y realizamos una modificación (update sobre esa tienda con ID 6789) y cambiamos el nombre a ‘Mediamarkt’ sin cerrar la transacción.

```
begin;
update tienda set nombre='MediaMarkat' where idtienda='6789'
```

```
1 begin;
2 update tienda set nombre='MediaMarkat' where idtienda='6789'
3 --insert into tienda values(6789,'TiendaNueva7','CiudadNueva7','BarrioNuevo7','ProvinciaNueva7');
4 --set session authorization usuario5;
5 --select current_user;
```

Explain Data Output Notifications Query History Messages

UPDATE 1

Query returned successfully in 140 msec.

T1 pide un bloqueo exclusivo (update) sobre la tabla tienda y se concede sin problema, se ejecuta y almacena en memoria local de T1 (hasta commit no sale a global).

tienda/postgres@PostgreSQL 12

Query Editor

```

1  --begin;
2  --update tienda set nombre='MediaMarkat' where idtienda='6789'
3  select * from tienda
4  --insert into tienda values(6789,'TiendaNueva7','CiudadNueva7','BarrioNuevo7','ProvinciaNueva7');
5  --set session authorization usuario5;
6  --select current_user;

```

Explain Data Output Notifications Query History Messages

	idtienda [PK] integer	nombre text	ciudad text	barrio text	provincia text
1	123456789	TiendaNuev...	CiudadNue...	BarrioNu...	ProvinciaNue...
2	1000	TiendaNuev...	CiudadNue...	BarrioNu...	ProvinciaNue...
3	2000	TiendaNuev...	CiudadNue...	BarrioNu...	ProvinciaNue...
4	31145	Tienda Alcalá	CiudadNue...	BarrioNu...	ProvinciaNue...
5	6789	MediaMarkat	CiudadNue...	BarrioNu...	ProvinciaNue...

Como se puede observar si hacemos un select desde el usuario1/usuario5 aparece correctamente modificado.

Cuestión 23: Abrir una sesión con el usuario2 de la base de datos TIENDA. Abrir una transacción T2 en este usuario y realizar una modificación de la tienda con código 6789 y cambiar el nombre a “Saturn”. No cierre la transacción. ¿Qué es lo que ocurre? ¿Por qué? ¿Qué información se puede obtener de la actividad de ambas transacciones en el sistema? ¿Es lógica esa información? ¿Por qué?

1. Con el usuario2 abrimos otra transacción T2 y modificaremos (update) la tienda con ID 6789 y cambiamos su nombre a ‘Saturn’ sin comprometer la transacción

```
update tienda set nombre='Saturn' where idtienda='6789'
```

Query Editor

```

1  begin;
2  update tienda set nombre='Saturn' where idtienda='6789'
3  --set session authorization usuario2;
4  --select current_user;

```

Waiting for the query to complete...

✖

18032

postgres

pgAdmin 4 - CONN:1179769

::1

2020-05-28 13:42:21 CEST

active

Lock: transactionid

17368

Details

Backend type

client backend

Query started at

2020-05-28 13:44:44 CEST

Last state changed at

2020-05-28 13:44:44 CEST

SQL

```

1 begin;
2 update tienda set nombre='Saturn' where idtienda='6789'
3 --set session authorization usuario2;
4 --select current_user;

```

2.Expliación de lo que sucede

Instantes	T1	T2	Situación actual -> T2 espera a que T1 se comprometa y libere el bloqueo exclusivo que tiene sobre la tabla tienda. Hasta entonces, T2 deberá esperar.
0	Beggin; XLOC Tienda		
1		Beggin; XLOC Tienda	

¿Qué es lo que ocurre? Postgres se bloquea como si se tratase de un interbloqueo (AUNQUE NO LO ES ya que T1 no espera a NADIE) ya que espera a que T1 acabe, se comprometa, modifique los datos sobre memoria global y libere el bloqueo.

El problema y la diferencia de este ejercicio frente al otro es que en la fase de crecimiento en la que se piden los bloqueos, T1 solicita el suyo sobre tienda (que se le concede sin problemas) y T2 en su fase pide un bloqueo exclusivo e incompatible sobre la tabla tienda. Es incompatible desde el primer instante que lo solicita.

¿Por qué? Detecta el problema de que 2 transacciones pidan un bloqueo exclusivo sobre la misma tienda y pone a T2 a esperar hasta que T1 libere recursos.

¿Qué información se puede obtener de la actividad de ambas transacciones en el sistema? La primera transacción T1 se ejecuta correctamente en su memoria local, al contrario que la transacción T2 que espera.

¿Es lógica esa información? Sí.

¿Por qué? La información es lógica dado que T2 llegó a pedir el bloqueo exclusivo después de T1 al que obviamente se le concedió por que no suponía ningún problema con ningún otro bloqueo de ninguna otra transacción. Hasta que T1 no finalice, T2 no va a poder escribir (update) NADA porque T1 sigue necesitando/empleando la tabla tienda. Esta técnica de no permitir a T2 escribir a la vez que T1 surge por una de las propiedades ACID, la de I (Isolation), que cumple la función de asegurar el aislamiento de los datos. T1 se ve como si sólo ella existiera en la base de datos y se gestiona la concurrencia mediante estos bloqueos. También nos asegura la C (Consistency) que nos garantiza la consistencia de los datos, que no se mezclan los datos ni se pierden.

Cuestión 24: Comprometa la transacción **T1**, ¿Qué es lo que ocurre? ¿Por qué? ¿Cuál es el estado final de la información de la tienda con código 6789 para ambos usuarios? ¿Por qué?

Hemos realizado el commit en la transacción **T1** y por lo tanto la modificación que hemos hecho pasa a memoria global y se ve reflejada a su vez en la base de datos que será visible por el resto de los usuarios.

```
1 --begin;
2 --update tienda set nombre='MediaMarkat' where idtienda='6789'
3 --select * from tienda
4 --insert into tienda values(6789,'TiendaNueva7','CiudadNueva7','BarrioNuevo7','ProvinciaNueva7');
5 --set session authorization usuario5;
6 --select current_user;
7 commit;
```

Explain Data Output Notifications Query History Messages

COMMIT

Query returned successfully in 83 msec.

```
7 commit;
8 select * from tienda
```

Explain Data Output Notifications Query History Messages

	idtienda [PK] integer	nombre text	ciudad text	barrio text	provincia text
1	123456789	TiendaNuev...	CiudadNue...	BarrioNu...	ProvinciaNue...
2	1000	TiendaNuev...	CiudadNue...	BarrioNu...	ProvinciaNue...
3	2000	TiendaNuev...	CiudadNue...	BarrioNu...	ProvinciaNue...
4	31145	Tienda Alcalá	CiudadNue...	BarrioNu...	ProvinciaNue...
5	6789	MediaMarkat	CiudadNue...	BarrioNu...	ProvinciaNue...

Teníamos la transacción **T2** del ejercicio anterior en espera ya que **T1** tenía exclusividad sobre la tabla tienda que también es la que quiere modificar **T2**. Apenas realizamos el commit se liberan esos datos y por lo tanto **T2** ya puede hacer el update (write) sin problemas porque ahora es el que tiene exclusividad sobre los datos. Y como observamos en las siguientes imágenes ahora en la memoria local de **T2** aparece lo que hemos realizado con el update aunque en la base de datos está lo que hemos insertado en **T1**. Por lo tanto, la información final de la tabla tienda será la que hay cuando ejecutamos el commit de **T1**.

```

1 begin;
2 update tienda set nombre='Saturn' where idtienda='6789'
3 --set session authorization usuario2;
4 --select current_user;

```

UPDATE 1

Query returned successfully in 10 min 20 secs.

Desde el usuario2 - T2 esto es lo que observamos:

```

1 --begin;
2 --update tienda set nombre='Saturn' where idtienda='6789'
3 select * from tienda
4 --set session authorization usuario2;
5 --select current_user;

```

Explain Data Output Notifications Query History Messages

	idtienda [PK] integer	nombre text	ciudad text	barrio text	provincia text
1	123456789	TiendaNuev...	CiudadNue...	BarrioNu...	ProvinciaNue...
2	1000	TiendaNuev...	CiudadNue...	BarrioNu...	ProvinciaNue...
3	2000	TiendaNuev...	CiudadNue...	BarrioNu...	ProvinciaNue...
4	31145	Tienda Alcalá	CiudadNue...	BarrioNu...	ProvinciaNue...
5	6789	Saturn	CiudadNue...	BarrioNu...	ProvinciaNue...

Cuestión 25: Comprometa la transacción T2, ¿Qué es lo que ocurre? ¿Por qué? ¿Cuál es el estado final de la información de la tienda con código 6789? ¿Por qué?

```

1 --begin;
2 --update tienda set nombre='Saturn' where idtienda='6789'
3 --select * from tienda
4 --set session authorization usuario2;
5 --select current_user;
6 commit;

```

Explain Data Output Notifications Query History Messages

COMMIT

Query returned successfully in 76 msec.

Usuario1/Ususario5

Query Editor

```
1 --begin;
2 --update tienda set nombre='MediaMarkat' where idtienda='6789'
3 --select * from tienda
4 --insert into tienda values(6789,'TiendaNueva7','CiudadNueva7','BarrioNuevo7','ProvinciaNueva7');
5 --set session authorization usuario5;
6 --select current_user;
7 --commit;
8 select * from tienda
```

Explain

Data Output

Notifications

Query History

Messages

	idtienda [PK] integer	nombre text	ciudad text	barrio text	provincia text
1	123456789	TiendaNuev...	CiudadNue...	BarrioNu...	ProvinciaNue...
2	1000	TiendaNuev...	CiudadNue...	BarrioNu...	ProvinciaNue...
3	2000	TiendaNuev...	CiudadNue...	BarrioNu...	ProvinciaNue...
4	31145	Tienda Alcalá	CiudadNue...	BarrioNu...	ProvinciaNue...
5	6789	Saturn	CiudadNue...	BarrioNu...	ProvinciaNue...

Usuario2

Query Editor

```
1 --begin;
2 --update tienda set nombre='Saturn' where idtienda='6789'
3 select * from tienda
4 --set session authorization usuario2;
5 --select current_user;
6 --commit;
```

Explain

Data Output

Notifications

Query History

Messages

	idtienda [PK] integer	nombre text	ciudad text	barrio text	provincia text
1	123456789	TiendaNuev...	CiudadNue...	BarrioNu...	ProvinciaNue...
2	1000	TiendaNuev...	CiudadNue...	BarrioNu...	ProvinciaNue...
3	2000	TiendaNuev...	CiudadNue...	BarrioNu...	ProvinciaNue...
4	31145	Tienda Alcalá	CiudadNue...	BarrioNu...	ProvinciaNue...
5	6789	Saturn	CiudadNue...	BarrioNu...	ProvinciaNue...

¿Qué es lo que ocurre? Como podemos observar, la transacción T2 ya ha finalizado de forma exitosa y los resultados ya han salido a memoria global (como comparativa en las imágenes se puede ver que la modificación aparece ya visible para todos los usuarios, tanto el 2 como el 1/5)

¿Por qué? El resultado es exitoso porque T1 ya finalizó y se comprometió y liberó el bloqueo sobre tienda, por lo que T1 cogió el relevo del bloqueo, hizo sus modificaciones y se comprometió correctamente.

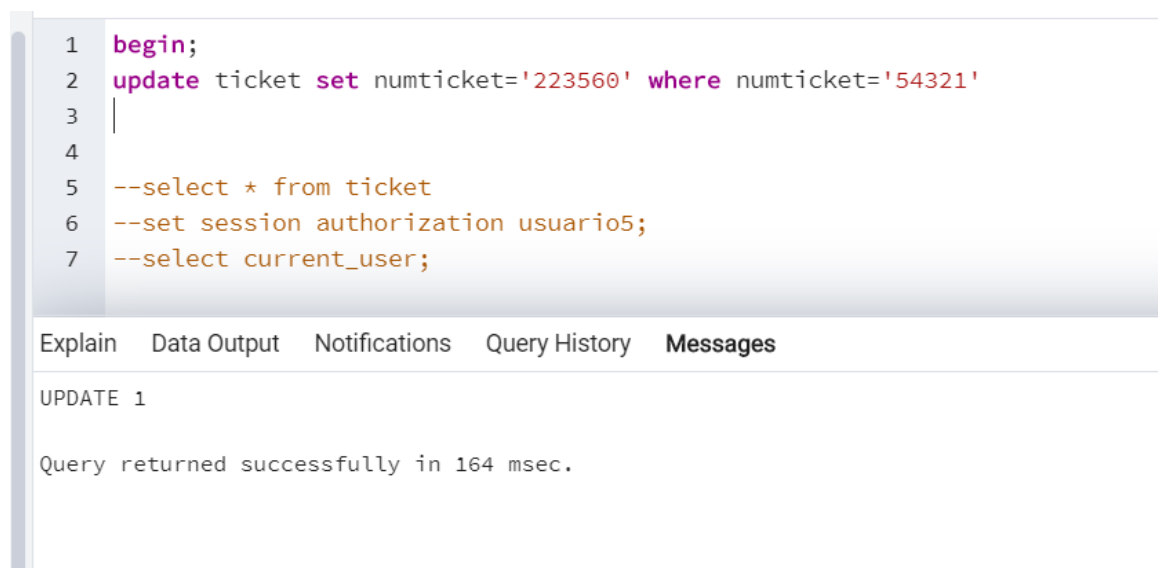
¿Cuál es el estado final de la información de la tienda con código 6789? ¿Por qué?

El estado final de la tabla tienda, es con el nombre 'Saturn' ya que en **T2** al ejecutar el **commit** los datos pasan a memoria global y por lo tanto se actualiza la tabla tienda que será visible desde otros usuarios que acceden a la base de datos, como vemos en las imágenes superiores.

Cuestión 26: Cerrar todas las sesiones anteriores. Abrir una sesión con el usuario1 de la base de datos TIENDA. Abrir una transacción T1 en este usuario y realizar una modificación del ticket con número 54321 para cambiar su código a 223560. Abra otro usuario diferente del anterior y realice una transacción T2 que cambie la fecha del ticket con número 54321 a la fecha actual. No cierre la transacción.

1. Abrimos una transacción T1 con el usuario1 (nuestro usuario5) y modificamos (update) sobre la tabla ticket y modificamos el ID TICKET de 54321 a 223560.

```
begin;
update ticket set numticket='223560' where numticket='54321'
```

A screenshot of a SQL IDE interface. The top part shows a code editor with the following SQL code:

```
1 begin;
2 update ticket set numticket='223560' where numticket='54321'
3
4
5 --select * from ticket
6 --set session authorization usuario5;
7 --select current_user;
```

Below the code editor, there is a tabbed interface with tabs for 'Explain', 'Data Output', 'Notifications', 'Query History', and 'Messages'. The 'Data Output' tab is selected, showing the result of the UPDATE query: 'UPDATE 1'. Below this, a message states: 'Query returned successfully in 164 msec.'

El update de la T1 se realiza correctamente.


2. Con el usuario 2 abrimos una transacción T2 para modificar la fecha del ticket con número 54321 a la fecha actual

```
begin;
update ticket set fecha='29-05-2020' where numticket='54321'
```

```

Query Editor
1  Begin;
2  update ticket set fecha='29-05-2020' where numticket='54321'
3  --select * from ticket
4
5  --set session authorization usuario2
6  --select current_user;

```



Waiting for the query to complete...

El update de la T2 no se lleva a cabo, debe espera a que T1 se comprometa (commit).

Instantes	T1	T2	
0	Beggin; XLOC Ticket		<p>Situación actual -> T1 pide permisos de bloqueo exclusivo sobre la tabla Ticket y se le concede.</p> <p>T2 quiere también pedir escribir sobre la tabla Ticket, pero deberá esperar a que T1 termine, se comprometa y libere el bloqueo.</p> <p>Por otro lado, T2 quiere acceder a un dato con un ID Ticket = 54321. Esto será un problema dado que cuando T1 acabe y se comprometa, modificará la memoria global y el ID del Ticket por 223560, por lo que T2 nunca podrá llevarse a cabo ya que eseID Ticket no existirá más.</p>
1		Beggin; XLOC Ticket	

Cuestión 27: Comprometa la transacción T1, ¿Qué es lo que ocurre? ¿Por qué? ¿Cuál es el estado de la información del ticket con código 54321 para ambos usuarios? ¿Por qué?


```
tienda/postgres@PostgreSQL 12

Query Editor

1  --begin;
2  --update ticket set numticket='223560' where numticket='54321'
3  commit;
4
5  --select * from ticket
6  --set session authorization usuario5;
7  --select current_user;

Explain  Data Output  Notifications  Query History  Messages

COMMIT

Query returned successfully in 125 msec.
```

¿Qué es lo que ocurre? ¿Por qué?

Hemos realizado el commit, y los datos de que estaban en memoria local pasan a memoria global y por lo tanto se actualiza la base de datos con esa modificación que será visible por otros usuarios de la base de datos.

¿Cuál es el estado de la información del ticket con código 54321 para ambos usuarios? ¿Por qué?

Antes del commit la información que hemos introducido estaba en memoria local de **T1** y al comprometer la transacción hemos pasado esos datos a la memoria global por lo tanto el estado final de esta tabla será con la modificación que hemos hecho, ID Ticket = 223560 para todos los usuarios de la base de datos. No entra en conflicto con **T2** ya que es otra transacción y **T1** tenía permisos de exclusividad.

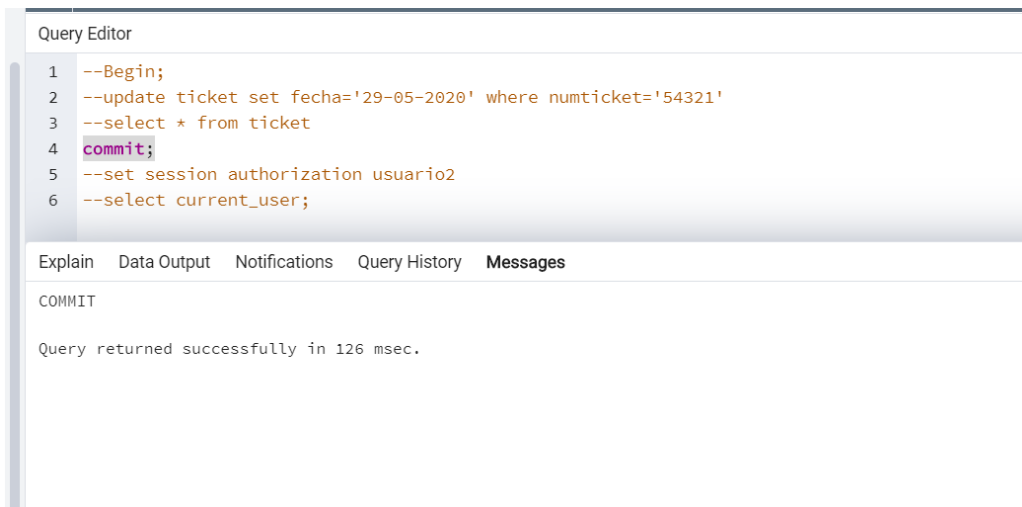
```
1  --begin;
2  --update ticket set numticket='223560' where numticket='54321'
3  --commit;
4  select * from ticket
5  --set session authorization usuario5;
6  --select current_user;
```

	numticket [PK] integer	fecha text	importe integer	codigotrabajador integer
1	54300	15-08-2019	20	789456
2	223560	01-01-2020	150	123456

Observamos que la modificación se ha llevado a cabo con éxito.

Cuestión 28: Comprometa la transacción T2, ¿Qué es lo que ocurre? ¿Por qué? ¿Cuál es el estado final de la información del ticket con número 54321 para ambos usuarios? ¿Por qué?

Comprometemos la T2:



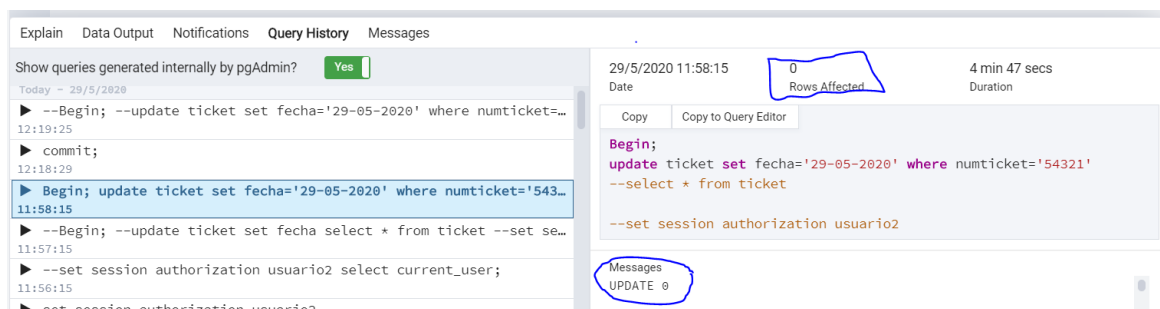
```
1  --Begin;
2  --update ticket set fecha='29-05-2020' where numticket='54321'
3  --select * from ticket
4  commit;
5  --set session authorization usuario2
6  --select current_user;
```

COMMIT

Query returned successfully in 126 msec.

¿Qué es lo que ocurre? ¿Por qué?

La transacción **T2** en el ejercicio 26, estaba esperando a que **T1** liberara los datos ya que tenía permisos de exclusividad sobre ellos y **T2** quería acceder a la misma tupla. Cuando en el ejercicio anterior hemos hecho el commit sobre **T1**, automáticamente postgres ha hecho que **T2** tenga permisos sobre esos datos y pueda ejecutar su transacción. Aunque en este caso el update no se ha podido hacer ya estamos intentando cambiar la fecha mediante un identificador que ha modificado por **T1** y dicho ID Ticket ya no existe.



The screenshot shows the 'Query History' and 'Messages' panels. In the 'Query History' panel, the query 'Begin; update ticket set fecha='29-05-2020' where numticket='54321'...' is highlighted. The 'Messages' panel shows the message 'UPDATE 0', indicating that no rows were affected by the update operation.

En esta captura mostramos el 'update 0' que nos muestra que 0 rows/filas han sido modificadas por que obviamente el ticket no existe más con ese identificador.

¿Cuál es el estado de la información del ticket con código 54321 para ambos usuarios? ¿Por qué?

Como no hemos podido hacer el update por problemas de consistencia de los datos ya que hemos intentado actualizar un valor mediante un identificador incorrecto que previamente en T1 ha sido modificado, el estado final de la tabla tienda es el siguiente:

Usuario2:

```
1  --Begin;
2  --update ticket set fecha='29-05-2020' where numticket='54321'
3  select * from ticket
4  --commit;
5  --set session authorization usuario2
6  --select current_user;
```

Explain	Data Output	Notifications	Query History	Messages
	numticket [PK] integer	fecha text	importe integer	codigotrabajador integer
1	54300	15-08-2019	20	789456
2	223560	01-01-2020	150	123456

Usuario1(Nuestro Usuario5):

```
1  --begin;
2  --update ticket set numticket='223560' where numticket='54321'
3  --commit;
4  select * from ticket
5  --set session authorization usuario5;
6  --select current_user;
```

Explain	Data Output	Notifications	Query History	Messages
	numticket [PK] integer	fecha text	importe integer	codigotrabajador integer
1	54300	15-08-2019	20	789456
2	223560	01-01-2020	150	123456

Como se observa, desde los dos usuarios y para toda la base de datos el ID Ticket se ha modificado correctamente a 223560 pero su fecha no, se mantiene con la original.

Cuestión 29: ¿Qué es lo que ocurre en el sistema gestor de base de datos si dentro de una transacción que cambia el importe del ticket con número 223560 se abre otra transacción que borre dicho ticket? ¿Por qué?

Supongamos una transacción T1 que quiere modificar (write-update) el importe de un ticket con ID Ticket = 2235600 y otra transacción T2 quiere hacer un delete de ese ticket con el ID Ticket = 2235600.

Si T2 se abre después de T1, T1 modificará el importe del ticket sin problema, ya que llegó primero a solicitar el bloqueo y se le concede sin problema. T1 realiza la modificación y se compromete, manda los cambios a memoria global y acaba. T2 toma el relevo del bloqueo exclusivo sobre el ticket y lo borra, se compromete y en memoria global desaparece todo el ticket con ese ID (223560).

Esta gestión que tiene el SGBD de Postgres nos asegura como se mencionó en ejercicios anteriores, la propiedad de aislamiento entre transacciones. Cada SGBD tiene su propio control de concurrencia y el de Postgres está basado en bloqueos.

Instantes	T1	T2	<p>En la fase de crecimiento T1 pide un bloqueo exclusivo sobre Ticket. T2 también, pero como llega después de T1 le toca esperar a que T1 finalice.</p> <p>En la fase de decrecimiento T1 suelta el bloqueo sobre la tabla ticket y T2 lo coge, modifica y se compromete.</p> <p>El resultado que obtenemos es el delete total de ese ticket con ese ID.</p>
0	Beggin; XLOC Ticket (update)		
1		Beggin; XLOC Ticket (delete)	
2	Commit;		
3		Commit;	

Hemos realizado las pruebas mediante Postgres y para ello hemos insertado una tupla en la tienda ticket para hacer las pruebas:

***Emplearemos el ID en vez de 223560 a 2235600**

```
Insert into ticket values (2235600, '02-01-2020',200, 123456);
```

- Usuario1(Nuestro usuario5) -> T1

1ªParte

Comenzamos con el begin y a cambiar el precio, esta transacción tiene exclusividad total sobre los datos, por lo que la modificación se realiza de forma exitosa.

```

1  begin;
2  update ticket set importe='300' where numticket='2235600'
3
4  --Insert into ticket values (2235600, '02-01-2020',200, 123456);
5
6  --select * from ticket
7  --set session authorization usuario5;
8  --select current_user;

```

UPDATE 1

Query returned successfully in 126 msec.

2ªParte

Como hemos realizado el commit ya los datos pasan de memoria local a global y se actualiza la base de datos y por lo tanto será visible por todos los usuarios que accedan a la base de datos.

```
1  --begin;
2  --update ticket set importe='300' where numticket='2235600'
3  commit;
4  --Insert into ticket values (2235600, '02-01-2020',200, 123456);
5
6  --select * from ticket
7  --set session authorization usuario5;
8  --select current_user;
```

Explain

Data Output

Notifications

Query History

Messages

COMMIT

Query returned successfully in 135 msec.

3ªParte

Hemos hecho un select desde el usuario5 pero lo que vemos que desde aquí será visible por parte del resto de usuarios.

```
1  --begin;
2  --update ticket set importe='300' where numticket='2235600'
3  --commit;
4  --Insert into ticket values (2235600, '02-01-2020',200, 123456);
5  select * from ticket
6  --select * from ticket
7  --set session authorization usuario5;
8  --select current_user;
```

Explain

Data Output

Notifications

Query History

Messages


	numticket [PK] integer	fecha text	importe integer	codigotrabajador integer
1	54300	15-08-2019	20	789456
2	223560	01-01-2020	150	123456
3	2235600	02-01-2020	300	123456

- Usuario2 -> **T2**

1ªParte

Cuando ejecutamos esta transacción se queda esperando a **T1**, que hasta que no acabe no liberará esos datos para que **T2** tenga permisos de exclusividad y pueda ejecutar la transacción y en este caso borrar la tupla.

```
1 begin;
2 delete from ticket where numticket='2235600'
3 --select * from ticket
4 --set session authorization usuario2;
5 --select current_user;
6
7
```

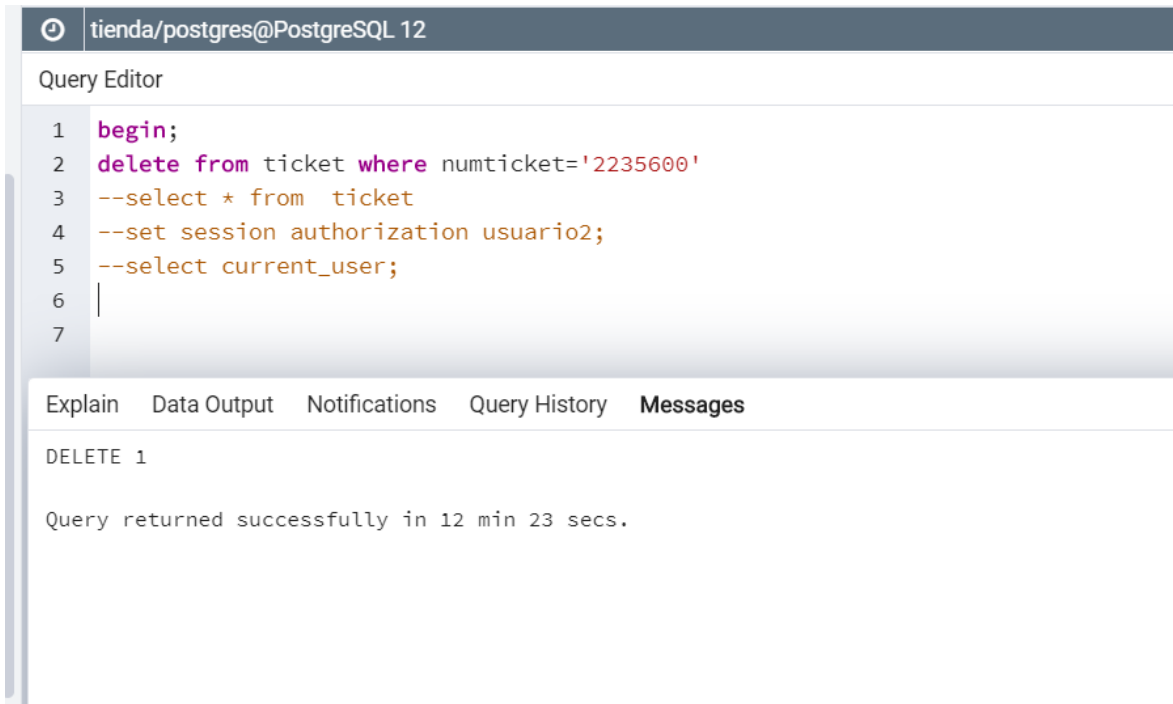


Waiting for the query to complete...

2ªParte

En el momento que ejecutamos el commit de **T1**, se liberan los datos y por lo tanto T2 ya puede disponer de ellos:

Observamos que se ha realizado exitosamente.



```
1 begin;
2 delete from ticket where numticket='2235600'
3 --select * from ticket
4 --set session authorization usuario2;
5 --select current_user;
6
7
```

tienda/postgres@PostgreSQL 12

Query Editor

Explain Data Output Notifications Query History Messages

DELETE 1

Query returned successfully in 12 min 23 secs.

3ªParte

Desde este usuario nos aparecerá el delete pero porque estamos visionando lo que hemos hecho a nivel local dentro de la transacción

```

1  --begin;
2  --delete from ticket where numticket='2235600'
3  select * from ticket
4  --set session authorization usuario2;
5  --select current_user;
6
7

```

Explain Data Output Notifications Query History Messages

	numticket [PK] integer	fecha text	importe integer	codigotrabajador integer
1	54300	15-08-20...	20	789456
2	223560	01-01-20...	150	123456

Cuando ejecutamos el commit ahora de **T2** ese delete será visible por el resto de los usuarios de la base de datos.

Query Editor

```

1  --begin;
2  --delete from ticket where numticket='2235600'
3  --select * from ticket
4  --set session authorization usuario2;
5  --select current_user;
6  commit;
7

```

Explain Data Output Notifications Query History Messages

COMMIT

Query returned successfully in 109 msec.

```

1  --begin;
2  --update ticket set importe='300' where numticket='2235600'
3  --commit;
4  --Insert into ticket values (2235600, '02-01-2020',200, 123456);
5  select * from ticket
6  --select * from ticket
7  --set session authorization usuarios;
8  --select current_user;

```

Explain Data Output Notifications Query History Messages

	numticket [PK] integer	fecha text	importe integer	codigotrabajador integer
1	54300	15-08-2019	20	789456
2	223560	01-01-2020	150	123456

Cuestión 30: Suponer que se produce una pérdida del cluster de datos y se procede a restaurar la instancia de la base de datos del punto 6. Realizar solamente la restauración (recovery) mediante el procedimiento descrito en el apartado 25.3 del manual (versión, 12) "*Continuous Archiving and point-in-time recovery (PITR)*". ¿Cuál es el estado final de la base de datos? ¿Por qué?

Simulamos la pérdida del cluster de datos. Para hacer el recovery, debemos restaurar el backup que hicimos previamente en el ejercicio 5. La recuperación se hará gracias al Sistema Gestor de recuperación. La recuperación se basará mirando los logs (redo-log) del WAL. De ahí ya se rehacen las transacciones comprometidas y descartan las no comprometidas. El algoritmo de recuperación que tiene Postgres funciona en dos partes:

1. Se van registrando en el sistema los logs de todos los write de las transacciones comprometidas que se dan. Se anotan en el WAL para tener un registro de todo lo que se ha hecho.
2. Si sucede algún fallo (errores lógicos, del sistema, etc) se recupera el contenido desde el último registro del log del WAL. De esta forma volvemos a un estado Consistente de la base de datos y aseguramos así su atomicidad y durabilidad.

Siguiendo el manual tenemos los siguientes pasos:

1. Verificamos si nuestro servidor está ejecutándose, en nuestro caso no lo está.
2. Ya que nuestro sistema por espacio nos los permite, guardamos una copia de seguridad de la carpeta C:\Program Files\PostgreSQL\12\data en otra carpeta de nuestro ordenador. En caso de no tener espacio solo es importante guardar la carpeta de pg_wal.

Este equipo > Documentos

Nombre	Fecha de modificación	Tipo	Tamaño
CURSO 2018-2019	29/11/2019 20:14	Carpeta de archivos	
CURSO 2019-2020	01/05/2020 23:48	Carpeta de archivos	
data	30/05/2020 17:34	Carpeta de archivos	

3. Eliminamos todos los archivos existentes en el directorio de datos del cluster.
4. Restauramos los archivos desde la copia de seguridad, es importante asegurarse de que se crea con el usuario del sistema de datos y NO de root. También hay que verificar que pg_tblspc se haya restaurado correctamente.

Aparece vacía ya que cuando iniciemos Postgres, el archivo tablespace_map (que a nosotras no nos aparece) a cada espacio de una tabla le asignará un enlace simbólico dentro de la carpeta pg_tblspc. Ese enlace lleva el nombre del oid del espacio en la tabla como forma de identificarlo.

5. Podríamos eliminar la carpeta llamada pg_wal ya que contiene todos los logs de las transacciones que hemos hecho hasta ahora, en ese caso como nos interesa recuperar la base de datos en el estado del ejercicio 6 no lo haremos, pero en

caso de querer recuperar las últimas transacciones tendríamos que eliminar los pg_wal de la copia de seguridad ya que probablemente estén obsoletas.

6. Establecemos la configuración de recuperación en el archivo postgresql.conf en el apartado de Checkpoints -> restore_command (destino_archivo+archivo_recuperar). También podemos modificar temporalmente el archivo pg_hba.conf para evitar que los usuarios no se conecten hasta que la recuperación no esté bien. Hay que crear un archivo .signal en el directorio data, tenemos dos opciones:
 - a. standby.signal: indica que el servidor debe iniciarse en modo espera activa.
 - b. recovery.signal: indica que el servidor debe iniciarse en modo de recuperación específico.

Si estuvieran ambos archivos presentes tiene prioridad standby.signal(espera). En versiones anteriores a PostgreSQL 12 utilizábamos recovery.conf para estas situaciones.

7. Iniciamos el servidor y entramos en modo de recuperación. Cuando acabe el servidor eliminará recovery.signal para evitar entrar en modo recuperación más tarde.
8. Verificamos la base de datos y en caso de haber modificado el pg_hba.conf se arreglaría para permitir el acceso de los usuarios a la base de datos.

Cuestión 31: A la vista de los resultados obtenidos en las cuestiones anteriores, ¿Qué tipo de sistema de recuperación tiene implementado postgresQL? ¿Qué protocolo de gestión de la concurrencia tiene implementado? ¿Por qué? ¿Genera siempre planificaciones secuenciables? ¿Genera siempre planificaciones recuperables? ¿Tiene rollbacks en cascada? Justificar las respuestas.

1.¿Qué tipo de sistema de recuperación tiene implementado postgresQL?

La recuperación que hace Postgres (el SGBD) es mirando el WAL, mira las transacciones comprometidas, que las recupera y las no comprometidas no se rehacen.

Esto implica que el esquema de recuperación es **DIFERIDA (con lista redolog)**, no inmediata (undo-redo). Esto lo hemos ido encontrando a medida que hacíamos los ejercicios, en los que veíamos que hasta que no se hacía un commit, el valor no salía a memoria global (la memoria de la base de datos) y por tanto lo que se hacía era modificar sobre la memoria local de la propia transacción de forma que no era visible hasta que se comprometía y salía a la global para ser vista para toda la base de datos. Podemos decir que la transacción se cree que 'es única en la base de datos'.

Los logs del WAL se guardan en un almacenamiento estable (en DISCO).

Se leería de memoria global los valores que no se tienen en memoria local.

No se registran los read, sólo los write, start, commit, abort y checkpoint.

Si cae el sistema sólo mira la lista redo (transacciones comprometidas) y nunca los undo y coloca los valores en la base de datos.

2.¿Qué protocolo de gestión de la concurrencia tiene implementado? ¿Por qué?

Como se explico anteriormente, cada SGBD tiene su propio esquema de control de concurrencia (que nos asegura la propiedad de aislamiento entre transacciones).

Hay varios tipos, pero el que usa Postgres es el **basado en bloqueos**.

Un bloqueo es un mecanismo de control concurrente para el acceso a un elemento de datos. Hay dos tipos, de escritura (XLOC) o exclusivo y el de lectura (SLOCK) o compartido. El exclusivo implica que sólo una transacción puede hacer uso de ese elemento (como hemos visto a lo largo de toda la práctica) y cualquier otra transacción que quiere hacer uso de ese elemento deberá esperar a que la transacción que inicialmente la tenía liberara el recurso, bien con un commit o con un rollback. En el caso de compartido, varias transacciones pueden leer un recurso a la vez.

Para asegurar la secuencialidad de conflictos (aunque no nos asegura la ausencia de interbloqueos/deadlocks) se hace uso de un protocolo de bloqueo de 2 fases:

- Fase de crecimiento, en la que se piden los bloqueos (no se liberan)
- Fase de decrecimiento en la que se liberan los bloqueos (no se obtienen)

3.¿Genera siempre planificaciones secuenciables?

NO. No siempre genera una planificación secuenciable, por eso mismo hay que estudiar su secuencialidad en cuanto a conflictos y vistas. Debemos estudiar si la planificación concurrente es equivalente a una en serie y si lo es, es secuenciable. Si la planificación es secuenciable en conflictos lo es también en vistas.

4.¿Genera siempre planificaciones recuperables?

SI. Dado que no hay rollback en cascada, la planificación siempre es recuperable. Esto implica que la transacción que hace el un write debe hacer el commit antes de que cualquier otra transacción lea los write que está haciendo dicha transacción.

5.¿Tiene rollbacks en cascada?

NO. Postgres no permite rollback en cascada, que se dan al leer un dato escrito no comprometido. No se dan ya que, como hemos comprobado, hasta que no se ejecuta al completo la transacción y se hace un commit, ninguna otra transacción puede leer de esta, por tanto, es imposible leer un dato escrito no comprometido.

Bibliografía

- Capítulo 13: Concurrency Control.
- Capítulo 25: Backup and Restore.
- Capítulo 27: Monitoring Database Activity.
- Capítulo 29: Reliability and the Write-Ahead log.