

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: David Ramos Fernández

ALUMNO 2:

Nombre y Apellidos: Sergio Sánchez Campo

Fecha: 07/06/2020

Profesor Responsable: Santiago Hermira Anchuelo

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?










Nuestro diario del sistema se encuentra activado, que es como se encuentra por defecto en Postgres y esto lo podemos comprobar mirando el atributo archive_mode que se encuentra dentro del archivo CONF en data postgres. Vemos que está en modo off lo que significa que está activo el diario.

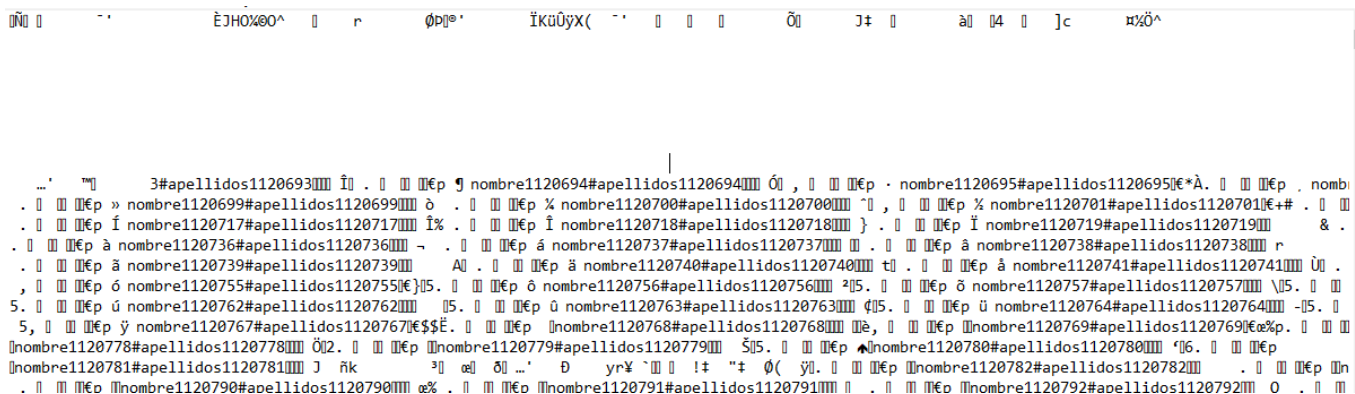
```
# - Archiving -
```

```
archive_mode = on                # enables archiving; off, on, or always
                                  # (change requires restart)
```

Se guardan en el directorio de postgresql data, en el archivo pg_wal.

El tamaño de estos archivos es de 16 MB que es el que viene por defecto en Postgres.

	0000000100000027000000A3	17/04/2020 18:21	Archivo	16.384 KB
	0000000100000027000000A4	17/04/2020 18:21	Archivo	16.384 KB
	0000000100000027000000A5	17/04/2020 18:21	Archivo	16.384 KB
	0000000100000027000000A6	17/04/2020 18:21	Archivo	16.384 KB
	0000000100000027000000A7	17/04/2020 18:21	Archivo	16.384 KB
	0000000100000027000000A8	17/04/2020 18:21	Archivo	16.384 KB
	0000000100000027000000A9	17/04/2020 18:21	Archivo	16.384 KB
	0000000100000027000000AA	17/04/2020 18:21	Archivo	16.384 KB
	0000000100000027000000AB	17/04/2020 18:21	Archivo	16.384 KB



```
INSERT INTO public."Tienda"  
VALUES (10002,'T1','Madrid','Madrid','Madrid');
```

```
@lg (@l' [T1]MadridMadridMadrid / 0 0 K' /
```

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.

El comando pg_waldump.exe sirve para interpretar un fichero WAL y obtener información y estadísticas entendibles.

Para ejecutar este comando nos vamos a la consola de comandos, en la dirección bin de Postgres y ejecutamos:

```
pg_waldump -z ../data/pg_wal/000000010000000270000000B6
```

Y nos da como resultado los siguientes valores:

```
D:\postgresql\bin>pg_waldump -z ../data/pg_wal/000000010000000270000000B6
```

Type	N	(%)	Record size	(%)	FPI size	(%)	Combined size	(%)
XLOG	2	(12,50)	228	(24,13)	0	(0,00)	228	(0,38)
Transaction	2	(12,50)	68	(7,20)	0	(0,00)	68	(0,11)
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	5	(31,25)	250	(26,46)	0	(0,00)	250	(0,42)
Heap2	0	(0,00)	0	(0,00)	0	(0,00)	0	(0,00)
Heap	2	(12,50)	108	(11,43)	16352	(27,78)	16460	(27,52)
Btree	3	(18,75)	159	(16,83)	22204	(37,72)	22363	(37,39)
Hash	2	(12,50)	132	(13,97)	20308	(34,50)	20440	(34,18)
Sin	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	16		945	[1,58%]	58864	[98,42%]	59809	[100%]

-En la primera columna se muestran los tipos de registros que se encuentran almacenados en dicho wal. En este caos podemos observar que la mayoría de filas son o, a excepción de:

XLOG, Transaction, Standby, Heap, Btree y Hash

-En la segunda columna (N) se muestra el número de registros de cada tipo. Observamos que el más numeroso es de tipo Standby con un 31,25%, que el número total de registros almacenados es de 16.

-En la tercera columna (Record size) se aprecia el tamaño de los registros, que en este wal ocupa un 1,58%.

-En la siguiente columna (FPI size) aparece el tamaño que ocupan las imágenes de página completa. En este caso es de un 98,42%.

-Finalmente la columna (Combined size) que es una combinación de las dos anteriores. Y su tamaño es de un 100% del archivo.

En conclusión este archivo esta compuesto principalmente por registros Hash, Btree y Heap de tipo imágenes de página completa. Y en cambio el porcentaje que ocupan los registros es muy pequeño.

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.

Dos de las maneras para obtener el id de la transacción anterior son:

- Buscando dicha transacción en el archivo log y localizando el valor que corresponde al id.

```
      "Id_tienda", "Nombre", "Ciudad", "Barrio", "Provincia")
VALUES (10002, 'T1', 'Madrid', 'Madrid', 'Madrid');
3116%13868%4/27 LOG: EXECUTOR STATISTICS
3116%13868%4/27 DETALLE: ! system usage stats:
      !      0.000000 s user, 0.000000 s system, 0.020084 s elapsed
      !      [0.109375 s user, 0.203125 s system total]
3116%13868%4/27 SENTENCIA: INSERT INTO public."Tienda"(
      "Id_tienda", "Nombre", "Ciudad", "Barrio", "Provincia")
VALUES (10002, 'T1', 'Madrid', 'Madrid', 'Madrid');
-----
```

-Obteniendo el id de la transacción actual (que es la siguiente a la anterior) mediante el siguiente comando y restándole 1, ya que queremos saber el id de la transacción anterior.

```
select * from txid_current();
```

```
3117
```

Una vez que tenemos el id, aplicamos el comando pg_wal para leer su información:

```
pg_waldump -x 3116 ../data/pg_wal/000000010000000270000000B7
```

```
D:\postgresql\bin>pg_waldump -x 3116 ../data/pg_wal/000000010000000270000000B7
mgr: Heap      len (rec/tot): 54/ 8234, tx: 3116, lsn: 27/87009270, prev 27/87009238, desc: INSERT off 103 flags 0x01, blkref #0: rel 1663/25437/26227 blk 372 FPM
mgr: Heap2     len (rec/tot): 56/ 56, tx: 3116, lsn: 27/87009288, prev 27/87009270, desc: CLEAN remxid 3115, blkref #0: rel 1663/25437/26227 blk 357
mgr: Btree     len (rec/tot): 53/ 7513, tx: 3116, lsn: 27/87009288, prev 27/87009288, desc: INSERT_LEAF off 206, blkref #0: rel 1663/25437/26792 blk 15 FPM
mgr: Hash      len (rec/tot): 66/ 11070, tx: 3116, lsn: 27/87009288, prev 27/87009288, desc: INSERT off 1, blkref #0: rel 1663/25437/26793 blk 684 FPM, blkref #1: rel 1663/25437/26793 blk 0 FPM
mgr: Btree     len (rec/tot): 53/ 7589, tx: 3116, lsn: 27/87009288, prev 27/87009288, desc: INSERT_LEAF off 99, blkref #0: rel 1663/25437/26794 blk 43 FPM
mgr: Hash      len (rec/tot): 66/ 8698, tx: 3116, lsn: 27/87011088, prev 27/87009288, desc: INSERT off 138, blkref #0: rel 1663/25437/26795 blk 111 FPM, blkref #1: rel 1663/25437/26795 blk 0 FPM
mgr: Btree     len (rec/tot): 53/ 7513, tx: 3116, lsn: 27/87013710, prev 27/87011088, desc: INSERT_LEAF off 386, blkref #0: rel 1663/25437/26796 blk 15 FPM
mgr: Transaction len (rec/tot): 34/ 34, tx: 3116, lsn: 27/87015C08, prev 27/87013710, desc: COMMIT 2020-06-04 18:50:56.730292 Hora de verano romance
```

Nos encontramos con Heap, Heap2, Btree, Hash y Transaction. Y en la columna más a la derecha podemos ver a que tarea corresponde cada registro. Observamos que el Heap y los Hash se encargan de realizar inserts. Los Btree insertan hojas. El Heap2 se encarga de realizar una limpieza. Y finalmente Transaction se encarga de realizar el Commit y se indica la fecha y hora a la que se realizó dicho Commit(18:50:56).

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)"*).

En primer lugar habría que verificar que el archivado WAL está funcionando, pero esto ya lo comprobamos en la cuestión 1.

A continuación creamos una carpeta backup donde almacenaremos el backup.

Dentro de ese archivo creamos una carpeta wals donde vamos a ir copiando wals que se generen en la base de datos. Para ello debemos cambiar archive_command en el postgresql.conf.

```
archive_command = 'copy "%p" "D:\\backup\\wals%f"'          # command to use to archive a
logfile segment
```

A continuación, habría que conectarse a la base de datos como superusuario y ejecutar el siguiente comando:

```
C:\Archivos de programa\Postgresql\12\bin>pg_basebackup -h localhost -p 5432 --username=postgres -D D:\backup
Contraseña:
```

Este equipo > Datos (D:) > backup				
	Nombre	Fecha de modificación	Tipo	Tamaño
★	PG_VERSION	08/06/2020 13:16	Archivo	1 KB
★	postgresql.auto	08/06/2020 13:16	Archivo CONF	1 KB
★	postgresql	08/06/2020 13:16	Archivo CONF	27 KB
★	pg_hba	08/06/2020 13:16	Archivo CONF	5 KB
★	pg_ident	08/06/2020 13:16	Archivo CONF	2 KB
	current_logfiles	08/06/2020 13:11	Archivo	1 KB
	datos_productos	08/06/2020 13:11	Documento de te...	4 KB
	datos_ticket	08/06/2020 13:11	Documento de te...	12 KB
	datos_Ticket_Productos	08/06/2020 13:11	Documento de te...	30 KB
	datos_tienda	08/06/2020 13:11	Documento de te...	1 KB
s	datos_Tienda_Productos	08/06/2020 13:11	Documento de te...	24 KB
	datos_trabajadores	08/06/2020 13:11	Documento de te...	5 KB
	backup_label	08/06/2020 13:09	Archivo	1 KB
	backup_label.old	08/06/2020 13:09	Archivo OLD	1 KB
	wals	08/06/2020 14:12	Carpeta de archivos	
	pg_wal	08/06/2020 13:40	Carpeta de archivos	
	global	08/06/2020 13:16	Carpeta de archivos	
	pg_replslot	08/06/2020 13:16	Carpeta de archivos	
	pg_serial	08/06/2020 13:16	Carpeta de archivos	
	pg_snapshots	08/06/2020 13:16	Carpeta de archivos	
	pg_stat	08/06/2020 13:16	Carpeta de archivos	
	pg_stat_tmp	08/06/2020 13:16	Carpeta de archivos	
	pg_subtrans	08/06/2020 13:16	Carpeta de archivos	
	pg_tblspc	08/06/2020 13:16	Carpeta de archivos	
	pg_twophase	08/06/2020 13:16	Carpeta de archivos	
	pg_xact	08/06/2020 13:16	Carpeta de archivos	
	log	08/06/2020 13:16	Carpeta de archivos	
	pg_commit_ts	08/06/2020 13:16	Carpeta de archivos	
	pg_dynshmem	08/06/2020 13:16	Carpeta de archivos	
	pg_logical	08/06/2020 13:16	Carpeta de archivos	
	pg_multixact	08/06/2020 13:16	Carpeta de archivos	
	pg_notify	08/06/2020 13:16	Carpeta de archivos	
	base	08/06/2020 13:11	Carpeta de archivos	

Tras ejecutarse ese comando observamos que se ha creado una copia de nuestro cluster de datos en el archivo backup creado anteriormente.

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?

Presenta la herramienta **MVCC** O Multiversion Concurrency Control, esta herramienta se encarga de que cada estado de SQL “vea” una instantánea de los datos

anteriores. Gracias a esto evita que se lean datos inconsistentes originados por transacciones concurrentes. Por otra parte, al controlar la concurrencia de esta manera y no mediante metodologías de bloqueo se consigue reducir la contención de bloqueo y permitir un buen rendimiento en entornos multiusuario.

También emplea la **Transaction Isolation** la cual define 4 niveles diferentes. El primero, serializable, que es el más estricto y cualquier proceso concurrente debe producir lo mismo que si cada uno corriese de uno en uno en el mismo orden.

Los 3 restantes se definen como fenómenos, el resultado de interacciones entre transacciones concurrentes, las cuales no deben ocurrir en cada nivel. Los fenómenos que se prohíben a varios niveles son: dirty read (que una transacción lea datos escritos por una transacción concurrente que no haya hecho commit), nonrepeatable read (que una transacción relea datos que hayan sido previamente leídos y cuyos datos hayan sido modificados por otra transacción), phantom read (que una transacción reejecute una query y devuelva un conjunto de columnas que satisfacen una condición de búsqueda pero dicho conjunto ha cambiado debido a otra transacción reciente que ha hecho commit) y serialization anomaly (el resultado del commit de un conjunto de transacciones es inconsistente).

Esta información se puede mostrar mediante el catálogo del sistema pg_stat_activity que recoge información sobre la actividad del servidor:

- datid: OID de la base de datos a la que está conectada.
- datname: nombre de la base de datos a la que se encuentra conectada.
- pid: ID del proceso.
- usesysid: nombre del usuario logeado.
- application_name: nombre de la aplicación a la que está conectada.
- client_addr: dirección IP del cliente conectado.
- client_hostname: Host name del cliente conectado.
- client_port: puerto TCP del cliente.
- backend_start: hora a la que comenzó el proceso.
- xact_start: hora a la que la transacción actual comenzó.
- query_start: hora a la que la query actual comenzó.
- state_change: hora a la que cambió el estado por última vez.
- waiting: indica si está esperando a un bloqueo.
- state: estado actual del proceso.
- query: la consulta más reciente.

O mediante el catálogo del sistema `pg_locks` que muestra todos los bloqueos del servidor de la base de datos. Este catálogo puede mostrar:

- locktype: el tipo de objeto bloqueable.
- database: el OID de la base de datos sobre el que se ha producido el bloqueo.
- relation: OID de la relación en la que se ha producido el bloqueo.
- page: número de bloque del bloque.
- tuple: número de tupla objetivo del bloqueo.
- virtualxid: ID virtual de la transacción objetivo del bloqueo.
- transactionid: ID de la transacción objetivo del bloqueo.
- classid: OID del catálogo del sistema que contiene el objetivo del bloqueo.
- objid: OID del objetivo del bloqueo.
- objsubid: número de columna del objetivo del bloqueo.
- virtualtransaction: ID virtual de la transacción que está sosteniendo o esperando al bloqueo.
- pid: ID del proceso del servidor que está sosteniendo o esperando al bloqueo.
- mode: nombre del modo de bloqueo del proceso.
- granted: indica si el proceso es sostenido o está esperando.
- fastpath: indica si el bloque fue producido por una ruta rápida o por la tabla principal.

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.

Comenzamos creando el rol de usuario que podrá leer las tablas, actualizarlas e insertar datos en ellas.

```
create role usuario;  
GRANT SELECT ON ALL TABLES IN SCHEMA PUBLIC TO usuario;  
GRANT INSERT ON ALL TABLES IN SCHEMA PUBLIC TO usuario;  
GRANT UPDATE ON ALL TABLES IN SCHEMA PUBLIC TO usuario;
```

A continuación creamos los 2 usuarios, `usuario1` y `usuario2` que pertenecerán al rol `usuario`. Con sus respectivas contraseñas.

```
CREATE USER usuario1 WITH PASSWORD 'usuario1' IN ROLE usuario;
CREATE USER usuario2 WITH PASSWORD 'usuario2' IN ROLE usuario;
```

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?

En primer lugar comenzamos la transacción, para ellos ejecutamos la siguiente consulta:

```
START TRANSACTION
```

A continuación, insertamos una nueva tienda en la base de datos.

```
INSERT INTO public."Tienda"(  
    "Id_tienda", "Nombre", "Ciudad", "Barrio", "Provincia")  
VALUES (100000000, 'T1', 'Madrid', 'Madrid', 'Madrid');
```

Finalmente vamos a comprobar la información del sistema para comprobar si esta tienda aparece en la base de datos aunque realmente no se haya insertado todavía, ya que la transacción continua activa.

Para ello ejecutamos:

```
SELECT * FROM "Tienda" where "Id_tienda"=100000000
```

Y obtenemos:

Id_tienda [PK] integer	Nombre text	Ciudad text	Barrio text	Provincia text
100000000	T1	Madrid	Madrid	Madrid

Comprobamos las tiendas que se encuentran en la base de datos y observamos que la tienda que hemos insertado previamente ya se encuentra en la base de datos aunque no se haya cerrado la transacción. Esto se debe a que nuestro superusuario está leyendo la información de manera local y no de la memoria global.

		PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
+	■	▶	13436	postgres	pgAdmin 4 - DB:TiendaPro	::1	2020-06-05 18:14:22 CEST	active	
+	■	▶	14612	postgres	pgAdmin 4 - CONN:5644635	::1	2020-06-05 18:21:10 CEST	idle in transaction	Client: ClientRead

Se encuentra activo en el sistema 2 transacciones. Una de ellas, la primera está activa y la otra parada.

PID	Lock type	Target relation	Page	Tuple	vXID (target)	XID (target)	Class	Object ID	vXID (owner)	Mode	Granted?
13436	relation	pg_locks							7/1129	AccessShareLock	true
14612	relation	"Tienda"							10/7471	RowExclusiveLock	true

Ambas tienen a true el Granted lo que significa que se les concedió el bloqueo requerido. Y observando el modo de bloqueo vemos que la primera está en modo de acceso compartido y la segunda, cuyo objetivo era la tabla "Tienda" está en exclusivo, ya que íbamos a insertar una tupla y por lo tanto necesitaba escribir.

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. Simplemente:

- 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.

Para cerrar la anterior transacción empleamos el siguiente comando:

```

1
END

```

```

COMMIT

```

```

Query returned successfully in 51 msec.

```

A continuación, vamos a comenzar a realizar transacciones con varios usuario a la vez y para realizar este procedimiento de una manera más sencilla vamos a emplear la consola de comandos.

Comenzamos conectándonos con el usuario1 y comenzando una transacción (BEGIN). Continuamos insertando la nueva tienda con id 1000. Observamos que se ha insertado adecuadamente así que continuamos con el Trabajador para la tienda anterior y con el Ticket con número 54321 para el Trabajador anterior.

```

TiendaPro=> BEGIN;
BEGIN
TiendaPro=> INSERT INTO public."Tienda"(
TiendaPro(>      "Id_tienda", "Nombre", "Ciudad", "Barrio", "Provincia")
TiendaPro->      VALUES (1000, 'T1', 'Madrid', 'Madrid', 'Madrid');
INSERT 0 1
TiendaPro=> INSERT INTO public."Trabajador"(
TiendaPro(>      codigo_trabajador, "DNI", "Nombre", "Apellidos", "Puesto", "Salario", "Id_tienda_Tienda")
TiendaPro->      VALUES (0, '71046348S', 'nombre', 'apellido', 'puesto', 1500, 1000);
INSERT 0 1
TiendaPro=> INSERT INTO public."Ticket"
TiendaPro-> VALUES (54321, 45456, '2019-03-16', 0);
INSERT 0 1

```

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

Para comprobar que se introdujeron adecuadamente los datos ejecutamos una consulta que muestre todas las tuplas de la Tabla que presenten la PK del dato introducido anteriormente.

Esta consulta debe ser ejecutada en el usuario1, ya que no hemos realizado el Commit todavía y por lo tanto esta información no se ha pasado a memoria global.

```

TiendaPro=> SELECT * from "Tienda" where      "Id_tienda"=1000;
 Id_tienda | Nombre | Ciudad | Barrio | Provincia
-----+-----+-----+-----+-----
      1000 | T1     | Madrid | Madrid | Madrid
(1 fila)

TiendaPro=> SELECT * from "Trabajador" where      codigo_trabajador=0;
codigo_trabajador | DNI | Nombre | Apellidos | Puesto | Salario | Id_tienda_Tienda
-----+-----+-----+-----+-----+-----+-----
              0 | 71046348S | nombre | apellido | puesto | 1500 | 1000
(1 fila)

TiendaPro=> SELECT * from "Ticket" where      "codigo_trabajador_Trabajador"=0;
 N|| de ticket | Importe | fecha | codigo_trabajador_Trabajador
-----+-----+-----+-----
      54321 | 45456 | 2019-03-16 | 0
(1 fila)

```

Podemos observar que se introdujeron adecuadamente todos los valores.

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?

Abrimos una nueva pestaña de la consola de comandos e iniciamos psql con el usuario2. A continuación ejecutamos las mismas consultas del ejercicio anterior y comprobamos que no hay ningún dato registrado. Esto se debe a que como no hemos hecho Commit todavía, los datos no se han pasado a memoria global. Y el usuario1 lo está leyendo de manera local.

```
D:\postgresql\bin>psql -U usuario2 -d TiendaPro
Contraseña para usuario usuario2:
psql (12.2)
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.

TiendaPro=> SELECT * from "Tienda" where "Id_tienda"=1000;
 Id_tienda | Nombre | Ciudad | Barrio | Provincia
-----+-----+-----+-----+-----
(0 filas)

TiendaPro=> SELECT * from "Trabajador" where codigo_trabajador=0;
codigo_trabajador | DNI | Nombre | Apellidos | Puesto | Salario | Id_tienda_Tienda
-----+-----+-----+-----+-----+-----+-----
(0 filas)

TiendaPro=> SELECT * from "Ticket" where "codigo_trabajador_Trabajador"=0;
 N|| de tickect | Importe | fecha | codigo_trabajador_Trabajador
-----+-----+-----+-----
(0 filas)

TiendaPro=>
```

		PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
+	■ ▶	3296	postgres	pgAdmin 4 - DB:TiendaPro	::1	2020-06-05 23:22:41 CEST	active		
+	■ ▶	9356	postgres	pgAdmin 4 - CONN:2780323	::1	2020-06-05 23:23:52 CEST	idle	Client: ClientRead	
+	■ ▶	10128	usuario2	psql	::1	2020-06-05 23:47:17 CEST	idle	Client: ClientRead	
+	■ ▶	10824	usuario1	psql	::1	2020-06-05 23:47:05 CEST	idle in transaction	Client: ClientRead	

La actividad que podemos observar es que hay dos sesiones activas desde el psql. Una de ellas pertenece al usuario1 y la otra al usuario2. En la columna State podemos comprobar que el usuario1 se encuentra en medio de una transacción, ya que todavía no ha habido ningún Commit de dicha transacción.

PID	Lock type	Target relation	Page	Tuple	vXID (target)	XID (target)	Class	Object ID	vXID (owner)	Mode	Granted?
3296	relation	pg_locks							4/1464	AccessShareLock	true
10824	relation	"indice_Tienda_idB"							7/139	RowShareLock	true
10824	relation	"indice_Tienda_allB"							7/139	AccessShareLock	true
10824	relation	"indice_Tienda_allB"							7/139	RowShareLock	true
10824	relation	"indiceProvincias"							7/139	AccessShareLock	true
10824	relation	"indiceProvincias"							7/139	RowShareLock	true
10824	relation	"Tienda_pk"							7/139	AccessShareLock	true
10824	relation	"Tienda_pk"							7/139	RowShareLock	true
10824	relation	"DNI_UNIQUE"							7/139	RowShareLock	true
10824	relation	"indice_Tienda_idH"							7/139	AccessShareLock	true
10824	relation	"Ticket"							7/139	RowExclusiveLock	true
10824	relation	"Trabajador"							7/139	RowShareLock	true
10824	relation	"Trabajador"							7/139	RowExclusiveLock	true
10824	relation	"Tienda"							7/139	AccessShareLock	true
10824	relation	"Tienda"							7/139	RowShareLock	true
10824	relation	"Tienda"							7/139	RowExclusiveLock	true
10824	relation	"indice_Trabajador_idB"							7/139	RowShareLock	true
10824	relation	"indice_Trabajador_CodoldB"							7/139	RowShareLock	true
10824	relation	"indice_Trabajador_allB"							7/139	RowShareLock	true
10824	relation	"Trabajador_pk"							7/139	RowShareLock	true
10824	relation	"indice_Trabajador_codb"							7/139	RowShareLock	true
10824	relation	"indice_Trabajador_codH"							7/139	RowShareLock	true
10824	relation	"indice_Trabajador_idH"							7/139	RowShareLock	true
10824	relation	"indiceSalario"							7/139	RowShareLock	true

Respecto a los locks observamos que el número es muy elevado y que la mayoría pertenecen al usuario1 ya que es el que ha estado realizando los inserts. Pero también observamos en la última columna que todas las peticiones de bloqueo han sido concedidas.

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?

No, ya que todavía no se ha realizado ninguna operación para comprometerlos. Los datos de la 2.10 se obtienen de la memoria local de dicha transacción. Por eso el usuario1 si es capaz de ver las nuevas tuplas, ya que fue el que las quiere insertar y se encuentran en su memoria local.

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?

El usuario1 realiza el COMMIT y a continuación ejecutamos de nuevo las consultas anteriores. Usuario1 obtiene el mismo resultado, ya que ha hecho Commit de los datos, los cuales anteriormente ya podía leer de memoria local.

```

TiendaPro=> COMMIT;
COMMIT
TiendaPro=> SELECT * from "Tienda" where          "Id_tienda"=1000;
 Id_tienda | Nombre | Ciudad | Barrio | Provincia
-----+-----+-----+-----+-----
      1000 | T1     | Madrid | Madrid | Madrid
(1 fila)

TiendaPro=> SELECT * from "Trabajador" where      codigo_trabajador=0;
codigo_trabajador | DNI      | Nombre | Apellidos | Puesto | Salario | Id_tienda_Tienda
-----+-----+-----+-----+-----+-----+-----
              0 | 71046348S | nombre | apellido  | puesto | 1500    | 1000
(1 fila)

TiendaPro=> SELECT * from "Ticket" where          "codigo_trabajador_Trabajador"=0;
N|| de ticket | Importe | fecha      | codigo_trabajador_Trabajador
-----+-----+-----+-----+-----
      54321 | 45456   | 2019-03-16 | 0
(1 fila)

```

Pero el usuario2 ha variado y ahora también muestra las nuevas tuplas insertadas. Eso se debe a que con el Commit dichos datos han pasado a memoria global y de esta manera el resto de usuarios de la base de datos son capaces de leer los nuevos datos insertados, ya que se han comprometido.

```

D:\postgresql\bin>psql -U usuario2 -d TiendaPro
Contraseña para usuario usuario2:
psql (12.2)
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.

TiendaPro=> SELECT * from "Tienda" where          "Id_tienda"=1000;
 Id_tienda | Nombre | Ciudad | Barrio | Provincia
-----+-----+-----+-----+-----
      1000 | T1     | Madrid | Madrid | Madrid
(1 fila)

TiendaPro=> SELECT * from "Trabajador" where      codigo_trabajador=0;
codigo_trabajador | DNI      | Nombre | Apellidos | Puesto | Salario | Id_tienda_Tienda
-----+-----+-----+-----+-----+-----+-----
              0 | 71046348S | nombre | apellido  | puesto | 1500    | 1000
(1 fila)

TiendaPro=> SELECT * from "Ticket" where          "codigo_trabajador_Trabajador"=0;
N|| de ticket | Importe | fecha      | codigo_trabajador_Trabajador
-----+-----+-----+-----+-----
      54321 | 45456   | 2019-03-16 | 0
(1 fila)

```

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.

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

Vamos a emplear el superusuario Postgres.

Comenzamos la transacción y ejecutamos las 3 inserciones.

```
START TRANSACTION;
Insert into "Tienda"
VALUES (2000, 'T1', 'Madrid', 'Madrid', 'Madrid');
Insert into "Trabajador"
VALUES (50, '71046347S', 'nombre', 'apellido', 'puesto', 1500, 2000);
Insert into "Ticket"
VALUES (54322, 45456, '2019-03-16', 50);
```

A continuación, ejecutamos el Update del trabajador con código 50.

```
UPDATE "Trabajador" SET "Id_tienda_Tienda"= 1000 WHERE codigo_trabajador=50;
```

```
UPDATE 1
```

```
Query returned successfully in 42 msec.
```

Observamos que el Update se ha realizado correctamente y hacemos commit de la transacción

```
COMMIT;
```

Tras el Commit ejecutamos diversas consultas para observar el resultado de las insercciones y el update.

```
SELECT * from "Tienda" where "Id_tienda"=1000;
```


	Id_tienda [PK] integer	Nombre text	Ciudad text	Barrio text	Provincia text
1	1000	T1	Madrid	Madrid	Madrid

```
SELECT * from "Trabajador" where codigo_trabajador=50;
```

	codigo_trabajador [PK] integer	DNI character varying (9)	Nombre character varying	Apellidos character varying	Puesto character varying	Salario integer	Id_tienda_Tienda integer
1	50	71046347S	nombre	apellido	puesto	1500	1000

```
SELECT * from "Trabajador" where "Id_tienda_Tienda"= 1000;
```

	codigo_trabajador [PK] integer	DNI character varying (9)	Nombre character varying	Apellidos character varying	Puesto character varying	Salario integer	Id_tienda_Tienda integer
1	0	71046348S	nombre	apellido	puesto	1500	1000
2	50	71046347S	nombre	apellido	puesto	1500	1000

```
SELECT * from "Ticket" where "N° de ticket"=54322;
```

	N° de ticket [PK] integer	Importe integer	fecha date	codigo_trabajador_Trabajador integer
1	54322	45456	2019-03-...	50

Las tuplas de Tienda y Ticket se encuentran de la misma manera que se insertaron. Y la tupla de Trabajador ha variado de acuerdo al Update y presenta solamente los datos actualizados, ya que el Update se produjo antes del Commit y por lo tanto los datos se comprometieron una vez que ya se había variado el Id_tienda_Tienda. El Update no genera ningún error ya que se actualiza una FK de un valor existente a otro que también existía. Si se hubiese actualizado a otro valor diferente, cuya PK no existiese se hubiese producido un error y se habría generado un Rollback de toda la transacción. Ya que la transacción o se ejecuta toda o no se ejecuta.

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?

Repetimos la misma consulta de la cuestión 9, pero en este caso acabamos con un Rollback. Y vamos a realizarla con el usuario1 para luego comparar la consulta con el usuario2.

```
D:\postgresql\bin>psql -U usuario1 -d TiendaPro
Contraseña para usuario usuario1:
psql (12.2)
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.

TiendaPro=> BEGIN;
BEGIN
TiendaPro=> INSERT INTO public."Tienda"(
TiendaPro(>      "Id_tienda", "Nombre", "Ciudad", "Barrio", "Provincia")
TiendaPro->      VALUES (3000, 'T1', 'Madrid', 'Madrid','Madrid');
INSERT 0 1
TiendaPro=>
TiendaPro=> INSERT INTO public."Trabajador"(
TiendaPro(>      codigo_trabajador, "DNI", "Nombre", "Apellidos", "Puesto", "Salario", "Id_tienda_Tienda")
TiendaPro->      VALUES (100, '71046346S', 'nombre', 'apellido', 'puesto', 1500, 3000);
INSERT 0 1
TiendaPro=> INSERT INTO public."Ticket"
TiendaPro-> VALUES (54323, 45456, '2019-03-16', 100);
INSERT 0 1
TiendaPro=>
TiendaPro=>
TiendaPro=> SELECT * from "Tienda" where      "Id_tienda"=3000;
 Id_tienda | Nombre | Ciudad | Barrio | Provincia
-----+-----+-----+-----+-----
      3000 | T1     | Madrid | Madrid | Madrid
(1 fila)

TiendaPro=> SELECT * from "Trabajador" where      codigo_trabajador=100;
codigo_trabajador | DNI      | Nombre | Apellidos | Puesto | Salario | Id_tienda_Tienda
-----+-----+-----+-----+-----+-----+-----
          100 | 71046346S | nombre | apellido | puesto |    1500 |          3000
(1 fila)

TiendaPro=> SELECT * from "Ticket" where      "codigo_trabajador_Trabajador"=100;
 N| de tickect | Importe | fecha      | codigo_trabajador_Trabajador
-----+-----+-----+-----+-----
54323 | 45456 | 2019-03-16 |          100
(1 fila)

TiendaPro=> ROLLBACK;
ROLLBACK
```

Una vez realizado el Rollback realizamos la consulta con ambos usuarios

```
TiendaPro=>
TiendaPro=> SELECT * from "Tienda" where          "Id_tienda"=3000;
 Id_tienda | Nombre | Ciudad | Barrio | Provincia
-----+-----+-----+-----+-----
(0 filas)

TiendaPro=> SELECT * from "Trabajador" where      codigo_trabajador=100;
 codigo_trabajador | DNI | Nombre | Apellidos | Puesto | Salario | Id_tienda_Tienda
-----+-----+-----+-----+-----+-----+-----
(0 filas)

TiendaPro=> SELECT * from "Ticket" where          "codigo_trabajador_Trabajador"=100;
 N|| de tickect | Importe | fecha | codigo_trabajador_Trabajador
-----+-----+-----+-----+-----
(0 filas)
```

```
D:\postgresql\bin>psql -U usuario2 -d TiendaPro
Contraseña para usuario usuario2:
psql (12.2)
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.

TiendaPro=> SELECT * from "Tienda" where          "Id_tienda"=3000;
 Id_tienda | Nombre | Ciudad | Barrio | Provincia
-----+-----+-----+-----+-----
(0 filas)

TiendaPro=> SELECT * from "Trabajador" where      codigo_trabajador=100;
 codigo_trabajador | DNI | Nombre | Apellidos | Puesto | Salario | Id_tienda_Tienda
-----+-----+-----+-----+-----+-----+-----
(0 filas)

TiendaPro=> SELECT * from "Ticket" where          "codigo_trabajador_Trabajador"=100;
 N|| de tickect | Importe | fecha | codigo_trabajador_Trabajador
-----+-----+-----+-----+-----
(0 filas)

TiendaPro=>
```

Obtenemos en los 2 el mismo resultado, que no se ha introducido la tupla. Esto se debe a que al haber hecho un Rollback dicha transacción es deshecha y por lo tanto se revierten todas las modificaciones que se hayan realizado durante esa transacción. Lo cual incluye también las modificaciones a nivel local. Usuario2 no iba a presentar las tuplas ya que no se había hecho Commit de la transacción ,pero tampoco lo hace Usuario1 ya que el Rollback también revierte las modificaciones a nivel local.

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.

Una vez cerradas todas las operaciones anteriores abrimos psql con el usuario1 e insertamos los datos especificados:

```
D:\postgresql\bin>psql -U usuario1 -d TiendaPro
Contraseña para usuario usuario1:
psql (12.2)
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.

TiendaPro=> INSERT INTO public."Tienda"(
TiendaPro(>   "Id_tienda", "Nombre", "Ciudad", "Barrio", "Provincia")
TiendaPro->   VALUES (31145, 'T1', 'Madrid', 'Madrid','Madrid');
INSERT 0 1
TiendaPro=>
TiendaPro=> INSERT INTO public."Trabajador"(
TiendaPro(>   codigo_trabajador, "DNI", "Nombre", "Apellidos", "Puesto", "Salario", "Id_tienda_Tienda")
TiendaPro->   VALUES (45678, '71046345S', 'nombre', 'apellido', 'puesto', 1500, 31145);
INSERT 0 1
TiendaPro=>
```

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?

A continuación iniciamos sesión con el usuario2, iniciamos transacción y modificamos la tienda que insertamos en el ejercicio anterior.

```
D:\postgresql\bin>psql -U usuario2 -d TiendaPro
Contraseña para usuario usuario2:
psql (12.2)
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.
```

```
TiendaPro=>
TiendaPro=> BEGIN;
BEGIN
TiendaPro=> UPDATE "Tienda" SET  "Nombre"='Tienda Alcalá' WHERE "Id_tienda"=31145;
UPDATE 1
TiendaPro=>
```

Miramos la actividad de la base de datos:

		PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
+	■	▶	3296	postgres	pgAdmin 4 - DB:TiendaPro	::1	2020-06-05 23:22:41 CEST	active	
+	■	▶	8128	usuario1	psql	::1	2020-06-06 00:12:45 CEST	idle	Client: ClientRead
+	■	▶	8996	usuario2	psql	::1	2020-06-06 00:12:51 CEST	idle in transaction	Client: ClientRead
+	■	▶	9356	postgres	pgAdmin 4 - CONN:2780323	::1	2020-06-05 23:23:52 CEST	idle	Client: ClientRead

Observamos que están los usuarios 1 y 2. El 1 se encuentra en idle, ya que no está realizando ninguna transacción. En cambio, el 2 se encuentra esperando pero en medio de una transacción ya que hemos iniciado la transacción en 2, hemos realizado un update pero no hemos comprometido los datos ni se ha producido un Rollback.

PID	Lock type	Target relation	Page	Tuple	vXID (target)	XID (target)	Class	Object ID	vXID (owner)	Mode	Granted?
3296	relation	pg_locks							4/3030	AccessShareLock	true
8996	relation	"indice_Tienda_idB"							8/8	RowExclusiveLock	true
8996	relation	"indice_Tienda_allB"							8/8	RowExclusiveLock	true
8996	relation	"indice_Tienda_idH"							8/8	RowExclusiveLock	true
8996	relation	"Tienda_pk"							8/8	RowExclusiveLock	true
8996	relation	"Tienda"							8/8	RowExclusiveLock	true
8996	relation	"indiceProvincias"							8/8	RowExclusiveLock	true

Los bloqueos son básicamente exclusivos ya que se están realizando inserciones y actualizaciones las cuales requieren de escritura. Pero de momento no se ha producido ningún bloqueo ya que solo T2 está en medio de una transacción activa.

Nos conectamos con el superusuario Postgres y ejecutamos:

```
SELECT * from "Tienda" where "Id_tienda"=31145;
```

Data Output

Explain

Messages

Notifications

	Id_tienda [PK] integer	Nombre text	Ciudad text	Barrio text	Provincia text
1	31145	T1	Madrid	Madrid	Madrid

```
SELECT * from "Trabajador" where codigo_trabajador=45678;
```

	codigo_trabajador [PK] integer	DNI character varying (9)	Nombre character varying	Apellidos character varying	Puesto character varying	Salario integer	Id_tienda_Tienda integer
1	45678	71046345S	nombre	apellido	puesto	1500	31145

Vemos que las inserciones se realizaron adecuadamente y que todavía no se ha producido ningún cambio, ya que la transacción del usuario2 no se ha comprometido todavía.

Vemos que la información se ha actualizado correctamente, esto se debe a que el usuario1

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?

Comenzamos una transacción con el usuario1 y realizamos el Update:

```
TiendaPro=>
TiendaPro=> BEGIN;
BEGIN
TiendaPro=>
TiendaPro=> UPDATE "Trabajador" SET "Salario"=3000 WHERE codigo_trabajador=45678;
UPDATE 1
```

Miramos la actividad en la base de datos:

	PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
3296	postgres	pgAdmin 4 - DB:TiendaPro	::1	2020-06-05 23:22:41 CEST	active			
8128	usuario1	psql	::1	2020-06-06 00:12:45 CEST	idle in transaction	Client: ClientRead		
8996	usuario2	psql	::1	2020-06-06 00:12:51 CEST	idle in transaction	Client: ClientRead		
9356	postgres	pgAdmin 4 - CONN:2780323	::1	2020-06-05 23:23:52 CEST	idle	Client: ClientRead		

Observamos que en este caso, el usuario1 también se encuentra en espera en medio de una transacción al igual que el usuario2, esto se debe a que hemos iniciado una transacción con el primer usuario y no la hemos comprometido, al igual que ocurría y sigue ocurriendo con el usuario2.

Realizamos las siguientes consultas para comprobar la información de la base de datos:

```
Select * from "Tienda" where "Id_tienda"=31145;
```

	Id_tienda [PK] integer	Nombre text	Ciudad text	Barrio text	Provincia text
1	31145	T1	Madrid	Madrid	Madrid

Select * from "Trabajador" where "Id_tienda_Tienda"=31145

	codigo_trabajador [PK] integer	DNI character varying (9)	Nombre character varying	Apellidos character varying	Puesto character varying	Salario integer	Id_tienda_Tienda integer
1	45678	71046345S	nombre	apellido	puesto	1500	31145

Observamos que no se ha producido ningún cambio respecto a la cuestión anterior, debido a que simplemente hemos iniciado una nueva transacción pero no la hemos comprometido por lo que la información no ha pasado a memoria global.

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?

Realizamos el Update en el usuario2 pero observamos que este Update no se produce:

```
TiendaPro=>
TiendaPro=> UPDATE "Trabajador" SET "Puesto"='Capataz' WHERE codigo_trabajador=45678;
```

Para analizar la causa de que no se produzca el Update comprobamos la actividad de la base de datos:

		PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
+	■	▶ 3296	postgres	pgAdmin 4 - DB:TiendaPro	::1	2020-06-05 23:22:41 CEST	active		
+	■	▶ 8128	usuario1	psql	::1	2020-06-06 00:12:45 CEST	idle in transaction	Client: ClientRead	
+	■	▶ 8996	usuario2	psql	::1	2020-06-06 00:12:51 CEST	active	Lock: transactionid	8128
+	■	▶ 9356	postgres	pgAdmin 4 - CONN:2780323	::1	2020-06-05 23:23:52 CEST	idle	Client: ClientRead	

Nos fijamos en el usuario2 y vemos que presenta un estado activo y un bloqueo, esta era la razón por la que no se ejecutaba la consulta. Comprobamos que se encuentra bloqueada por la transacción con PID 8128, que es la perteneciente al usuario1. Entonces sabemos que T2 se encuentra bloqueada por T1, esto se debe a que T2 quiere modificar el trabajador con código 45678, por lo que necesitaría un bloqueo exclusivo de este dato para poder modificarlo. Pero anteriormente T1 también había realizado un Update de este mismo dato, por lo que lo tiene bloqueado de manera exclusiva y como todavía no ha realizado Commit lo sigue teniendo bloqueado. Por lo tanto, T2 no podrá avanzar hasta que T1 haga Commit o suelte dicho bloqueo.

```
Select * from "Tienda" where "Id_tienda"=31145;
```

Data Output

Explain

Messages

Notifications

	Id_tienda [PK] integer	Nombre text	Ciudad text	Barrio text	Provincia text	
1	31145	T1	Madrid	Madrid	Madrid	

```
Select * from "Trabajador" where "Id_tienda_Tienda"=31145
```

Data Output		Explain	Messages	Notifications			
	codigo_trabajador [PK] integer	DNI character varying (9)	Nombre character varying	Apellidos character varying	Puesto character varying	Salario integer	Id_tienda_Tienda integer
1	45678	71046345S	nombre	apellido	puesto	1500	31145

La base de datos continua con los mismos datos porque aunque se realicen cambios en una transacción si esta no se compromete, estos cambios no se reflejarán en la base de datos.

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?

Volvemos a la transacción T1 y ejecutamos el Update:

```
TiendaPro=> UPDATE "Tienda" SET "Barrio"='El Ensanche' WHERE "Id_tienda"=31145;
ERROR: se ha detectado un deadlock
DETALLE: El proceso 1284 espera ShareLock en transacción 3078; bloqueado por proceso 4060.
El proceso 4060 espera ShareLock en transacción 3079; bloqueado por proceso 1284.
SUGERENCIA: Vea el registro del servidor para obtener detalles de las consultas.
CONTEXTO: mientras se actualizaba la tupla (275,103) en la relación «Tienda»
```

Al ejecutar el comando se genera un Deadlock, lo que obliga a cancelar una de las transacciones, que en este caso será T1. El deadlock se genera debido a que T1 quiere actualizar la Tienda con id 31145, para realizar este Update necesitara el bloqueo no solo de dicha tupla si no también de aquellas que presentan dicho id como FK, que en este caso es el trabajador que estaba intentando actualizar T2. Por lo que T1 debe esperar a que T2 acabe su Update para comenzar, pero en la cuestión anterior ya habíamos observado que T2 estaba bloqueada ya que necesitaba un dato que tenía en modo exclusivo T1. En conclusión, T2 está bloqueado por T1 y T1 está bloqueado por T2, lo cual genera un deadlock y obliga a matar una de las transacciones(T1).

Tras haberse cancelado T1 observamos que el Update de T2 se ha realizado, debido a que T1 soltó el bloqueo necesario para que T2 continuara.


```

UPDATE 1
TiendaPro=> UPDATE "Trabajador" SET "Puesto"='Capataz' WHERE codigo_trabajador=45678;
UPDATE 1
TiendaPro=>

```

Analizando la actividad registrada en la base de datos, podemos volver a verificar lo anterior, observamos los dos usuarios y vemos que el estado del usuario1 es: libre en medio de una transacción pero ha sido abortado(debido al deadlock), mientras que el usuario2 continua libre en medio de la transacción.

	PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
3296	postgres	pgAdmin 4 - DB:TiendaPro	::1	2020-06-05 23:22:41 CEST	active			
8128	usuario1	psql	::1	2020-06-06 00:12:45 CEST	idle in transaction (aborted)	Client: ClientRead		
8996	usuario2	psql	::1	2020-06-06 00:12:51 CEST	idle in transaction	Client: ClientRead		
9356	postgres	pgAdmin 4 - CONN:2780323	::1	2020-06-05 23:23:52 CEST	idle	Client: ClientRead		

Si volvemos a observar los valores de la base de datos podemos comprobar que no se ha alterado ningún valor, ya que sigue sin comprometerse ninguna transacción.

```
SELECT * from "Tienda" where "Id_tienda"=31145;
```

Data Output

Explain

Messages

Notifications

	Id_tienda [PK] integer	Nombre text	Ciudad text	Barrio text	Provincia text
1	31145	T1	Madrid	Madrid	Madrid

```
SELECT * from "Trabajador" where codigo_trabajador=45678;
```

	codigo_trabajador [PK] integer	DNI character varying (9)	Nombre character varying	Apellidos character varying	Puesto character varying	Salario integer	Id_tienda_Tienda integer
1	45678	71046345S	nombre	apellido	puesto	1500	31145

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?

Finalmente comprometemos ambas transacciones.

Usuario 1:

```
TiendaPro=> COMMIT;  
ROLLBACK
```

Al comprometer el primer usuario observamos un Rollback, este se debe a que en la cuestión anterior se abortó la transacción T1 y por tanto ninguna de las modificaciones realizadas en medio de dicha transacción se iban a mantener. Así que se deshace completamente

Usuario 2:

```
TiendaPro=> COMMIT;  
COMMIT
```

La transacción del usuario2 se compromete correctamente ya que no se generó ningún error en medio de su ejecución.

Si nos volvemos a meter en la actividad de la base de datos observamos que tanto el usuario1 y el usuario2 ya no se encuentran en ninguna transacción y no están realizando ninguna actividad. Esto es lógico ya que ya se han producido los commit o rollback de cada una de las transacciones.

	PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
1284	usuario1	psql		::1	2020-06-03 12:05:37 CEST	idle	Client: ClientRead	
3028	postgres	pgAdmin 4 - DB:TiendaPro		::1	2020-06-03 09:34:24 CEST	active		
4060	usuario2	psql		::1	2020-06-03 12:05:55 CEST	idle	Client: ClientRead	
14852	postgres	pgAdmin 4 - CONN:1971851		::1	2020-06-03 09:36:16 CEST	idle	Client: ClientRead	

Finalmente si nos vamos a los valores que contiene la base de datos veremos que ahora si han variado:

```
SELECT * from "Tienda" where "Id_tienda"=31145;
```

Data Output

Explain

Messages

Notifications

	Id_tienda [PK] integer	Nombre text	Ciudad text	Barrio text	Provincia text
1	31145	Tienda Alcal	Madrid	Madrid	Madrid

```
SELECT * from "Trabajador" where codigo_trabajador=45678;
```

Data Output		Explain	Messages	Notifications				
	<div><div><div></div><div>codigo_trabajador</div><div>[PK] integer</div></div></div>		<div><div><div></div><div>DNI</div><div>character varying (9)</div></div></div>	<div><div><div></div><div>Nombre</div><div>character varying</div></div></div>	<div><div><div></div><div>Apellidos</div><div>character varying</div></div></div>	<div><div><div></div><div>Puesto</div><div>character varying</div></div></div>	<div><div><div></div><div>Salario</div><div>integer</div></div></div>	<div><div><div></div><div>Id_tienda_Tienda</div><div>integer</div></div></div>
1		45678	71046345S	nombre	apellido	Capataz	1500	31145

Los campos que han sido modificados han sido el Nombre de la Tienda que ha pasado a Tienda Alcalá y el Puesto del Trabajador que ha pasado a Capataz. Ambas fueron las modificaciones realizadas por la transacción T2, que es la que hizo Commit. Todas las modificaciones realizadas por T1 vemos que no se han reflejado en la base de datos puesto que hubo un Rollback.

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.

Iniciamos sesión con usuario1 en psql y realizamos un Insert de una Tienda con Id 6789.

```
D:\postgresql\bin>psql -U usuario1 -d TiendaPro
Contraseña para usuario usuario1:
psql (12.2)
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.

TiendaPro=>
TiendaPro=> INSERT INTO public."Tienda"(
TiendaPro(>      "Id_tienda", "Nombre", "Ciudad", "Barrio", "Provincia")
TiendaPro->      VALUES (6789, 'T1', 'Madrid', 'Madrid','Madrid');
INSERT 0 1
```

A continuación, comenzamos una transacción y realizamos una modificación en la tupla introducida anteriormente y le cambiamos el nombre a Mediamarkt.

```
TiendaPro=> BEGIN;
BEGIN
TiendaPro=> UPDATE "Tienda" SET "Nombre"='Mediamarkt' WHERE "Id_tienda"=6789;
UPDATE 1
TiendaPro=>
```

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?

Iniciamos sesión con el usuario2, comenzamos una transacción(Begin) y le cambiamos el nombre a la tupla anterior a Saturn.

```
D:\postgresql\bin>psql -U usuario2 -d TiendaPro
Contraseña para usuario usuario2:
psql (12.2)
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.

TiendaPro=> BEGIN;
BEGIN
TiendaPro=>
TiendaPro=> UPDATE "Tienda" SET "Nombre"='Saturn' WHERE "Id_tienda"=6789;
```

Lo que ocurre es que dicho Update no se produce, está bloqueado. Este bloqueo se produce porque T2 quiere modificar la Tienda introducido en la cuestión anterior, para ello necesita escribir por lo que requerirá de un bloqueo exclusivo de ese dato. Pero en la cuestión anterior T1 también inicio una transacción en la que modificaba ese mismo dato, por lo que T1 tiene un bloqueo exclusivo sobre ese dato, el cual no soltará hasta que haga Commit. En conclusión, este Update no se puede producir hasta que se acaba la transacción T1 y suelte sus bloqueos.

Analizando la actividad del sistema podemos corroborar lo anterior. El usuario1 se encuentra en medio de una transacción, lo cual es lógico ya que hemos iniciado una transacción con Begin pero no hemos hecho Commit. Y el usuario2 esta en estado activo pero lockeado, se encuentra bloqueado por la transacción 4624 la cual corresponde a la del usuario1. Esto es lógico ya que como hemos comentado anteriormente, en primer lugar el usuario1 realizo un Update en la tupla Tienda con Id 6789, para lo que necesito un bloqueo exclusivo, ya que necesita escribir. A continuación usuario2 va a realizar un Update en esa misma tupla pero para ello necesitaría un bloqueo exclusiva de ese dato, lo cual no se puede conceder ya que lo tiene el usuario1 también en modo exclusivo. Asi que la única manera de que usuario2 se desbloqueara es que usuario1 soltara dicho bloqueo al realizar un Commit de su transacción.

		PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
🔴	■	▶	4624	usuario1	psql	2020-06-04 10:37:02 CEST	idle in transaction	Client: ClientRead	
🔴	■	▶	6444	postgres	pgAdmin 4 - CONN:9217985	2020-06-04 10:22:39 CEST	idle	Client: ClientRead	
🔴	■	▶	14000	postgres	pgAdmin 4 - DB:TiendaPro	2020-06-04 10:22:20 CEST	active		
🔴	■	▶	16228	usuario2	psql	2020-06-04 10:37:59 CEST	active	Lock: transactionid	4624

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?

Comprometemos la transacción del usuario1 mediante un Commit.

```
TiendaPro=> COMMIT;
COMMIT
TiendaPro=>
```

Tras realizar dicho Commit observamos en el usuario2 que su Update avanza y pone UPDATE 1. Esto se debe a que estaba bloqueado por la transacción T1 y al hacer Commit, T1 libero sus bloqueos y T2 pudo continuar con su actividad.

```
TiendaPro=> UPDATE "Tienda" SET "Nombre"='Saturn' WHERE "Id_tienda"=6789;
UPDATE 1
TiendaPro=>
```

Ahora vamos a comprobar la información en cada uno de los usuarios:

Usuario1:

```
TiendaPro=> select * from "Tienda" WHERE "Id_tienda"=6789;
 Id_tienda |  Nombre   | Ciudad | Barrio | Provincia
-----+-----+-----+-----+-----
      6789 | Mediamarkt | Madrid | Madrid | Madrid
(1 fila)
```

En el usuario1 observamos que el nombre es Mediamarkt, esto se debe a que el usuario1 ha comprometido la transacción y por lo tanto la modificación al nombre se ha mantenido. Y en cambio el Update del usuario2 no se ha comprometido por lo que solo está registrado en memoria local en el usuario2.

Usuario2:

```
TiendaPro=> select * from "Tienda" WHERE "Id_tienda"=6789;
Id_tienda | Nombre | Ciudad | Barrio | Provincia
-----+-----+-----+-----+-----
        6789 | Saturn | Madrid | Madrid | Madrid
(1 fila)
```

En cambio, en el usuario2 aparece en nombre Saturn, esto se debe a que es la última información que se registró de esa columna, ya que cuando T1 hizo Commit registro en la columna Nombre Mediamarkt, pero tras ese Commit se desbloqueó T2 y realizó el Update a la Tienda, cambiando el nombre de Mediamarkt a Saturn, pero esta información solo se encuentra de manera local en el usuario2 ya que no la ha comprometido.

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?

Realizamos el commit de T2:

```
TiendaPro=> COMMIT;
COMMIT
TiendaPro=>
```

Lo que ocurre es que se compromete la información de T2 correctamente ya que no se ha producido ningún error en medio de su actividad. Por lo que se producirá la modificación del nombre de la tupla de MediaMarkt a Saturn. Porque la modificación de Saturn se produjo después de la modificación de MediaMarkt.

Vamos a comprobar esto mediante una consulta:

```
select * from "Tienda" WHERE "Id_tienda"=6789;
```

	Id_tienda [PK] integer	Nombre text	Ciudad text	Barrio text	Provincia text
1	6789	Saturn	Madrid	Madrid	Madrid

El estado final de la información de dicha Tienda es con Saturn en la columna Nombre. Esto se debe a que el Update de T2 se produjo después del Commit de T1, por lo tanto cuando T2 hace Commit actualiza el Nombre a Saturn.

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.

Iniciamos la sesión con el usuario1, comenzamos una transacción y realizamos el Update pertinente.

```
Contraseña para usuario usuario1:
psql (12.2)
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.

TiendaPro=> BEGIN;
BEGIN
TiendaPro=>
TiendaPro=> UPDATE "Ticket" SET  "N de ticket"=223560 WHERE  "N de ticket" =54321;
UPDATE 1
TiendaPro=>
```

A continuación, abrimos sesión con usuario2, abrimos también otra transacción y ejecutamos el Update al Ticket con número 54321.

```
Contraseña para usuario usuario2:
psql (12.3)
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.

TiendaPro=> BEGIN;
BEGIN
TiendaPro=> UPDATE "Ticket" SET  fecha='2020-03-06' WHERE  "N de ticket" =54321;
```

Observamos que el Update no se produce ya que necesitaría permisos de bloqueo exclusivo sobre la tupla de Ticket con número 54321 y esta se encuentra bloqueada de manera exclusiva por usuario1. Así que T2 no podrá avanzar hasta que T1 suelte sus bloqueos.

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?

Continuamos comprometiendo la transacción T1 mediante un Commit.

```
TiendaPro=> COMMIT;  
COMMIT  
TiendaPro=>
```

Observamos que la transacción commitea adecuadamente ya que no se ha producido ningún error en dicha transacción.

Como consecuencia de dicho Commit T1 sueltas sus bloqueos, lo que permite que T2 continúe con su Update.

```
TiendaPro=> UPDATE "Ticket" SET fecha='2020-03-06' WHERE "N de ticket" =54321;  
UPDATE 0  
TiendaPro=>
```

Pero observamos que el Update ha sido de 0, esto se debe a que al hacer Commit T1 el número del Ticket se ha modificado, ha pasado de 54321 a 223560. Por lo que cuando T2 ha ido a hacer el Update ya no existía ningún Ticket con el número 54321. Por lo que no se ha producido ningún Update.

Realizamos las consultas con cada usuario para comprobar la información que contienen:

Si lo comprobamos con el Ticket con número 54321:

Usuario1

```
TiendaPro=> select * from "Ticket" WHERE "N de ticket" =54321;  
N de ticket | Importe | fecha | codigo_trabajador_Trabajador  
-----+-----+-----+-----  
(0 filas)
```

Usuario2

```
TiendaPro=> select * from "Ticket" WHERE "N de ticket" =54321;  
N de ticket | Importe | fecha | codigo_trabajador_Trabajador  
-----+-----+-----+-----  
(0 filas)
```


Comprobamos que ninguno de los dos contiene ninguna información, puesto que se ha actualizado el número del Ticket al hacer Commit de T1 y actualmente no existe ningún Ticket con dicho número.

Si comprobamos la información con el nuevo número de Ticket:

Usuario1:

```
TiendaPro=> select * from "Ticket" WHERE "N de ticket" =223560;
N de ticket | Importe | fecha | codigo_trabajador_Trabajador
-----+-----+-----+-----
223560 | 45456 | 2020-03-06 | 0
(1 fila)
```

Usuario2:

```
TiendaPro=> select * from "Ticket" WHERE "N de ticket" =223560;
N de ticket | Importe | fecha | codigo_trabajador_Trabajador
-----+-----+-----+-----
223560 | 45456 | 2020-03-06 | 0
(1 fila)
```

Ahora sí que contienen información puesto que es el número de Ticket que se corresponde con la información actual, y vemos como no hay ninguna variación entre ambos usuarios puesto que el Update de T2 era sobre un dato desfasado.

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?

Finalmente comprometemos T2:

```
TiendaPro=> COMMIT;
COMMIT
TiendaPro=>
```

Esta se compromete correctamente puesto que no se generó ningún error en dicha transacción. Pero tampoco generó ninguna modificación en ningún valor puesto que su Update era sobre un dato desfasado, por lo que al hacer Commit no se va a alterar ningún dato respecto a la cuestión anterior.

Podemos comprobar esto realizando las mismas consultas que en el apartado anterior:

Usuario1:

```
TiendaPro=> select * from "Ticket" WHERE "N de tickect" =223560;
N de tickect | Importe | fecha      | codigo_trabajador_Trabajador
-----+-----+-----+-----
223560 | 45456 | 2020-03-06 | 0
(1 fila)
```

Usuario2:

```
TiendaPro=> select * from "Ticket" WHERE "N de tickect" =223560;
N de tickect | Importe | fecha      | codigo_trabajador_Trabajador
-----+-----+-----+-----
223560 | 45456 | 2020-03-06 | 0
(1 fila)
```

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?

Para realizar esta cuestión vamos a emplear el superusuario Postgres.

Comenzamos iniciando la transacción y ejecutamos el Update:

```
START Transaction
```

```
UPDATE "Ticket" SET "Importe"=223 WHERE "N de tickect" =223560;
```

Comprobamos con el mismo usuario que dicho Update se ha producido correctamente:

```
select * from "Ticket" where "N de tickect" =223560
```

	N de tickect [PK] integer	Importe integer	fecha date	codigo_trabajador_Trabajador integer
1	223560	223	20120-03-16	0

En la actividad observamos que hay una transacción en ejecución:

	PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
5600	postgres	pgAdmin 4 - DB:TiendaPro	::1	2020-06-06 10:07:52 CEST	active			
14252	postgres	pgAdmin 4 - CONN:8940932	::1	2020-06-06 10:08:02 CEST	idle in transaction	Client: ClientRead		

A continuación, iniciamos una segunda transacción con el mismo usuario, pero nos sale un mensaje de advertencia de que ya hay una transacción en curso:

```
WARNING: ya hay una transacción en curso
START TRANSACTION
```

```
Query returned successfully in 49 msec.
```

Continuamos realizando el delete:

```
DELETE FROM "Ticket" where "N de tickect" =223560;
```

```
DELETE 1
```

```
Query returned successfully in 51 msec.
```

Si consultamos la información del Ticket:

```
select * from "Ticket" where "N de tickect" =223560
```

	N de ticket [PK] integer	Importe integer	fecha date	codigo_trabajador_Trabajador integer

Observamos que de manera local se ha producido el Delete.

Ahora vamos a comprobar la información con el usuario1:

```

TiendaPro-# ;
N de ticket | Importe | fecha | codigo_trabajador_Trabajador
-----+-----+-----+-----
223560 | 45456 | 20120-03-16 | 0
(1 fila)

```

Comprobamos que no se ha producido ninguna modificación puesto que no ha habido ningún commit.

Examinando la actividad de la base de datos comprobamos que solamente hay una transacción en ejecución y no dos. Esto significa que cuando hemos hecho el segundo Commit no ha abierto una nueva transacción anidada sobre la primera transacción sino que continuamos sobre la primera transacción. La segunda vez que escribimos START TRANSACTION no ha realizado ninguna variación.

	PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
▶	5600	postgres	pgAdmin 4 - DB:TiendaPro	::1	2020-06-06 10:07:52 CEST	active		
▶	14252	postgres	pgAdmin 4 - CONN:8940932	::1	2020-06-06 10:08:02 CEST	idle in transaction	Client: ClientRead	

A continuación, realizamos el Commit y mediante la consulta comprobamos que se ha borrado la tupla. Ya que en primer lugar la transacción la ha actualizado y luego la ha borrado.

```
select * from "Ticket" where "N de ticket" =223560
```

	N de ticket [PK] integer	Importe integer	fecha date	codigo_trabajador_Trabajador integer

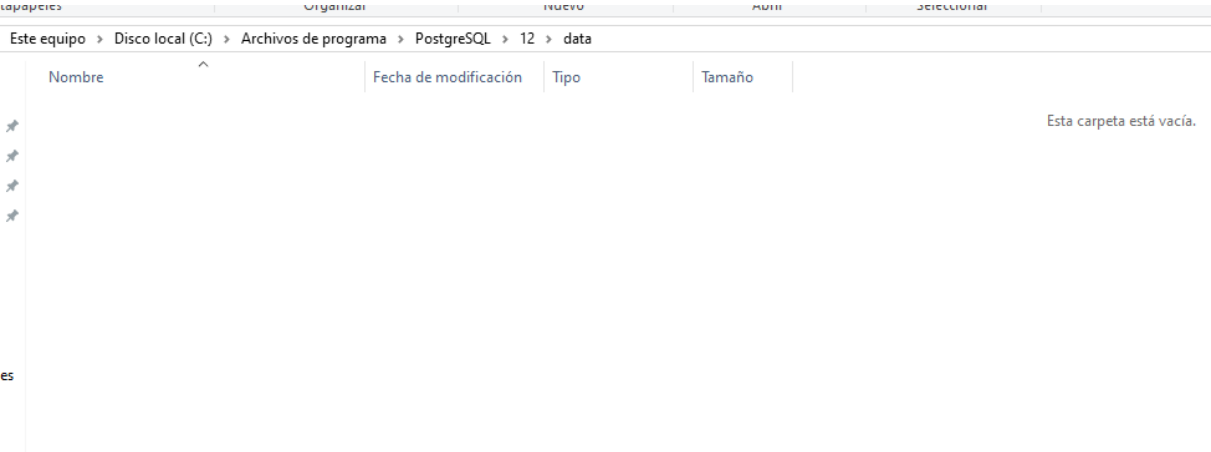
Para comprobar que solo había una transacción podemos ejecutar un nuevo Commit y nos salta el siguiente mensaje:

```
WARNING: no hay una transacción en curso
COMMIT
```

Indicando que no hay ninguna transacción en curso, puesto que no se puede abrir una nueva transacción en medio de una transacción activa.

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?

Suponemos que hemos perdido todo el cluster de datos así que nos vamos al directorio datos y borramos todos los archivos y carpetas.




Para comenzar la recuperación de datos en primer lugar debemos asegurarnos de detener el servidor. Esto lo podemos realizar deteniendo el servicio de la versión de postgresql correspondiente.







postgresql-x64-12	postgresql-x64-12 - PostgreSQL Serv...	Detenido
-------------------	--	----------

A continuación, creamos el archivo recovery.conf en el cluster de datos y modificamos el restore_command en el postgresql.conf e indicamos el directorio de nuestro backup.

```
restore_command = 'copy"D:\\backup\\%f""%p"'
# command to use to restore an archived logfile segment
# placeholders: %p = path of file to restore
#               %f = file name only
# e.g. 'cp /mnt/server/archivedir/%f %p'
# (change requires restart)
```

Nombre	Fecha de modificación	Tipo	Tamaño
 recovery.signal	06/06/2020 17:02	Archivo SIGNAL	0 KB

Los archivos que se han ido creando en la carpeta wals (que habíamos creado en la cuestión 5 y que habíamos marcado como destino de los archivos generados por archive_command) de nuestro backup las copiamos al pg_wal del backup.

ste equipo > Datos (D:) > backup > wals				
Nombre	Fecha de modificación	Tipo	Tamaño	
 wals00000003000000010000006B	08/06/2020 13:36	Archivo	16.384 KB	
 wals00000003000000010000006A	08/06/2020 13:33	Archivo	16.384 KB	
 pg_wal000000030000000100000069	08/06/2020 13:27	Archivo	16.384 KB	
 pg_wal000000030000000100000068	08/06/2020 13:16	Archivo	16.384 KB	
 pg_wal000000030000000100000068.00000...	08/06/2020 13:16	Archivo BACKUP	1 KB	
 pg_wal000000030000000100000067	08/06/2020 13:09	Archivo	16.384 KB	

Y finalmente copiamos el basebackup en el cluster de datos:

Este equipo > Disco local (C:) > Archivos de programa > PostgreSQL > 12 > data					
	Nombre	Fecha de modificación	Tipo	Tamaño	
	base	06/06/2020 17:04	Carpeta de archivos		
	global	06/06/2020 17:10	Carpeta de archivos		
	log	06/06/2020 17:10	Carpeta de archivos		
	pg_commit_ts	05/06/2020 23:32	Carpeta de archivos		
	pg_dynshmem	05/06/2020 23:32	Carpeta de archivos		
	pg_logical	06/06/2020 17:10	Carpeta de archivos		
	pg_multixact	06/06/2020 17:05	Carpeta de archivos		
	pg_notify	06/06/2020 17:10	Carpeta de archivos		
	pg_replslot	05/06/2020 23:32	Carpeta de archivos		
	pg_serial	05/06/2020 23:32	Carpeta de archivos		
	pg_snapshots	05/06/2020 23:32	Carpeta de archivos		
	pg_stat	05/06/2020 23:32	Carpeta de archivos		
	pg_stat_tmp	06/06/2020 17:12	Carpeta de archivos		
	pg_subtrans	05/06/2020 23:32	Carpeta de archivos		
	pg_tblspc	05/06/2020 23:32	Carpeta de archivos		
	pg_twophase	05/06/2020 23:32	Carpeta de archivos		
	pg_wal	06/06/2020 17:10	Carpeta de archivos		
	pg_xact	06/06/2020 17:04	Carpeta de archivos		
	backup_label.old	05/06/2020 23:30	Archivo OLD	1 KB	
	current_logfiles	06/06/2020 17:10	Archivo	1 KB	
	pg_hba	05/06/2020 23:32	Archivo CONF	5 KB	
	pg_ident	05/06/2020 23:32	Archivo CONF	2 KB	
	PG_VERSION	05/06/2020 23:32	Archivo	1 KB	
	postgresql.auto	05/06/2020 23:32	Archivo CONF	1 KB	
	postgresql	05/06/2020 23:32	Archivo CONF	27 KB	
	postmaster.opts	06/06/2020 17:10	Archivo OPTS	1 KB	
	postmaster.pid	06/06/2020 17:10	Archivo PID	1 KB	

Iniciamos el servidor:

```
C:\Archivos de programa\Postgresql\12\bin>pg_ctl start -D "C:\Archivos de programa\Postgresql\12\data"
esperando que el servidor se inicie...0%9960% LOG: iniciando PostgreSQL 12.3, compiled by Visual C++ build 1914, 64-bit
0%9960% LOG: escuchando en la dirección IPv6 «:::», port 5432
0%9960% LOG: escuchando en la dirección IPv4 «0.0.0.0», port 5432
0%9960% LOG: redirigiendo la salida del registro al proceso recolector de registro
0%9960% HINT: La salida futura del registro aparecerá en el directorio «log».
listo
servidor iniciado

C:\Archivos de programa\Postgresql\12\bin>
```

Comprobamos los datos de la tabla Tienda:

	 Id_tienda [PK] integer 	Nombre text 	Ciudad text 	Barrio text 	Provincia text 
1	10002	T1	Madrid	Madrid	Madrid

Comprobamos que presenta el contenido que tenía antes de realizar el backup, debido a es el momento en el que realizamos la copia de todo el cluster de datos y los archivos wal que presenta el backup no están totalmente actualizados por lo que no se ha podido recuperar todos los datos que presentaba la base de datos antes de realizar el backup. Si no hubiésemos perdido todo el cluster de datos podríamos haber cambiado los wal obsoletos por wal totalmente actualizados y entonces la base de datos estaría completa con todos los 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.

Como conclusión de los resultados de las cuestiones hemos comprobado que PostgreSQL presenta un sistema de recuperación basado en logs, en estos logs se almacena el registro de todas las modificaciones o actividades que se han realizado sobre la base de datos para que en caso de caída de la base de datos se pueda recuperar la información. Además, hemos observado que emplea una técnica de modificación diferida de la base de datos ya que la información no pasaba a memoria compartida hasta que se realizaba el Commit.

Respecto al protocolo de gestión de la concurrencia, emplea uno basado en bloqueos, más concretamente uno basado en un protocolo de dos fases refinado. Pues para permitir que una transacción se ejecutase o no daba dos niveles de bloqueo: compartido o exclusivo, el cual además se podía actualizar si la transacción requería de un nivel superior de bloqueo. Por lo que, si no se le podía conceder el bloqueo adecuado a una actividad de una transacción, esta no se ejecutaría hasta que se le concediese el bloqueo pertinente.

Respecto a las planificaciones logra que estas sean siempre secuenciables gracias al anteriormente nombrado sistema de bloqueos, ya que asegura que nunca se produzca una situación de inconsistencia ya que impide que haya situaciones de conflicto entre varias transacciones.

Las planificaciones no son siempre recuperables puesto que una transacción puede abrir una transacción, escribir un dato y a continuación iniciar otra transacción leer el dato que la transacción estaba escribiendo, por lo que leerá un dato obsoleto y a continuación realizar un commit antes de que la primera transacción (la que realizó primero el write) haya commiteado. Por lo tanto no es obligatoriamente recuperable, depende del orden en que se ejecuten las acciones de cada transacción.

Finalmente, en relación con lo anterior, si que puede presentar rollbacks en cascada ya que si no es obligatoriamente recuperable significa que puede tener rollbacks. Ya que si una transacción no es recuperable siempre va a presentar rollbacks. En esta práctica, por ejemplo, muchas veces una transacción ha leído un dato que estaba modificando otra transacción la cual todavía no había commiteado, que es lo que genera un rollback en cascada.

Bibliografía

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