

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: Laura Mambrilla Moreno

DNI: _____

ALUMNO 2:

Nombre y Apellidos: Isabel Blanco Martínez

DNI: _____

Fecha: 26 de Abril de 2020

Profesor Responsable: Óscar Gutiérrez Blanco

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 8 de Junio.** 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?

El diario del sistema no se encuentra activo por defecto.

Para activar el archivo automático de ficheros WAL hay que definir los parámetros *wal_level*, *archive_mode* a ON y *archive_command* en postgresql.conf. De momento no pondremos ninguna ruta específica en *archive_command*.

CONFIGURACIÓN PREVIA:

```
#-----  
# WRITE-AHEAD LOG  
#-----  
  
# - Settings -  
  
#wal_level = replica          # minimal, replica, or logical  
#                                # (change requires restart)  
  
# - Archiving -  
  
#archive_mode = off           # enables archiving; off, on, or always  
#                                # (change requires restart)  
#archive_command = ''         # command to use to archive a logfile segment  
#                                # placeholders: %p = path of file to archive  
#                                #                 %f = file name only
```

CONFIGURACIÓN FINAL:

```
#-----  
# WRITE-AHEAD LOG  
#-----  
  
# - Settings -  
  
wal_level = replica          # minimal, replica, or logical  
#                                # (change requires restart)  
  
# - Archiving -  
  
archive_mode = on             # enables archiving; off, on, or always  
#                                # (change requires restart)  
archive_command = ''          # command to use to archive a logfile segment  
#                                # placeholders: %p = path of file to archive
```

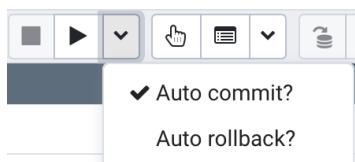
Estos ficheros tienen un nombre único y un tamaño por defecto de 16MB y se generan en el subdirectorio pg_wal que se encuentra en el directorio de datos usado por PostgreSQL. El número de ficheros WAL contenidos en pg_wal dependerá del valor asignado al parámetro *checkpoint_segments* en el fichero de configuración postgresql.conf.

El contenido de los archivos, como podemos ver, es indescriptible. Esto se debe a que el contenido de los ficheros WAL se escribe en binario.

Cuestión 2: Realizar una operación de inserción de una tienda sobre la base de datos TIENDA. Abrir el archivo de diario ¿Se encuentra reflejada la operación en el archivo del sistema? ¿En caso afirmativo, por qué lo hará?

```
INSERT INTO public."Tienda"(  
    "Id_tienda", "Nombre", "Ciudad", "Barrio", "Provincia")  
VALUES (999999999, 'Supeco', 'Madrid', 'Carabanchel', 'Madrid');
```

La operación sale reflejada en el sistema. Esto se debe a que, en PostgreSQL, cuando ejecutamos una consulta, tenemos predefinida la opción de que la consulta que hagamos finalice con un COMMIT, aunque podríamos elegir la opción de ROLLBACK para que lo deshaga.



Por lo cual, la transacción compromete todos los datos, ya que aunque no pongamos manualmente BEGIN, el sistema lo incluye de forma predefinida al inicio de la consulta.

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` sirve para traducir los archivos del WAL de binario al idioma que podemos reconocer los humanos. Podemos encontrar el ejecutable en la carpeta bin.

La sintaxis de pg_waldump es la siguiente:

pg_waldump [option...] [startseg [endseg]]

Usaremos -p como [opción] para que se traduzca el contenido del archivo que especifiquemos, dando la ruta en la que se encuentra dicho archivo y su nombre. En nuestro caso el último fichero WAL es 0000000100000023000000D5, por lo cual el comando que ejecutaremos es el siguiente:

```
Isabels-MacBook-Pro:bin postgres$ ./pg_waldump -p /Library/PostgreSQL/12/data/pg_wal 0000000100000023000000D5
```

Parte del archivo:

```
LOCK_ONLY KEYSHR_LOCK , blkref #0: rel 1663/25113/25114 blk 595 FPW
rmgr: Heap      len (rec/tot):   59/  8231, tx:    1138, lsn: 23/D5A0D050, prev 23/D5A0B010, desc: LOCK off 71: xid 1138: flags 0x00 L
OCK_ONLY KEYSHR_LOCK , blkref #0: rel 1663/25113/25114 blk 1080 FPW
rmgr: Heap      len (rec/tot):   59/  8247, tx:    1138, lsn: 23/D5A0F090, prev 23/D5A0D050, desc: LOCK off 44: xid 1138: flags 0x00 L
OCK_ONLY KEYSHR_LOCK , blkref #0: rel 1663/25113/25114 blk 1589 FPW
rmgr: Heap      len (rec/tot):   59/  8287, tx:    1138, lsn: 23/D5A110E0, prev 23/D5A0F090, desc: LOCK off 10: xid 1138: flags 0x00 L
OCK_ONLY KEYSHR_LOCK , blkref #0: rel 1663/25113/25114 blk 1008 FPW
rmgr: Standby   len (rec/tot):   54/     54, tx:      0, lsn: 23/D5A13108, prev 23/D5A110E0, desc: RUNNING_XACTS nextXid 1376 latestCo
mpletedXid 1375 oldestRunningXid 1138; 1 xacts: 1138
rmgr: Heap      len (rec/tot):   59/  8227, tx:    1138, lsn: 23/D5A13140, prev 23/D5A13108, desc: LOCK off 31: xid 1138: flags 0x00 L
OCK_ONLY KEYSHR_LOCK , blkref #0: rel 1663/25113/25114 blk 1540 FPW
```

Como podemos ver ahora estamos ante un archivo lleno de información que se puede interpretar perfectamente.

Para obtener las estadísticas del archivo usamos como [opción] -z

```
Isabels-MacBook-Pro:bin postgres$ ./pg_waldump -z /Library/PostgreSQL/12/data/pg_wal/0000000100000023000000D5
```

Obtenemos lo siguiente:

Type	N	(%)	Record size	(%)	FPI size	(%)	Combined size	(%)
-	-	-	-	-	-	-	-	-
XLOG	33 (1.19)		2898 (1.37)		960 (0.01)		3858 (0.04)	
Transaction	6 (0.22)		9583 (4.54)		0 (0.00)		9583 (0.09)	
Storage	23 (0.83)		966 (0.46)		0 (0.00)		966 (0.01)	
CLOG	0 (0.00)		0 (0.00)		0 (0.00)		0 (0.00)	
Database	4 (0.14)		144 (0.07)		0 (0.00)		144 (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	99 (3.57)		5042 (2.39)		0 (0.00)		5042 (0.05)	
Heap2	8 (0.29)		472 (0.22)		15020 (0.14)		15492 (0.14)	
Heap	1880 (67.85)		141255 (66.92)		10504648 (99.15)		10645983 (98.52)	
Btree	718 (25.91)		50736 (24.03)		73848 (0.70)		124584 (1.15)	
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	2771		211096 [1.95%]		10594476 [98.05%]		10805572 [100%]	

(Nota: el fatal error que sale en la última línea es la forma habitual en la que finaliza un archivo WAL).

Como podemos ver, se nos muestra el número de registros totales del archivo, el tamaño de cada uno según los tipos, el tamaño de las full-page images (FPI) además del tamaño combinado.

Para cada fila:

- **Type** es rmgr/registro
- **N** es el número de registros de cada tipo
- **%** es el porcentaje del total de registros
- **Record size** es sum(xl_len+SizeOfXLogRecord)

- % es el porcentaje del total de registros
- **FPI size** es el tamaño de full-page images
- % es el porcentaje del total de FPI size
- **Combined size** es sum(xl_tot_len)
- % es el porcentaje del total del tamaño combinado (combined size)

La última línea (“Total”) muestra el número total de registros de todos los tipos, el total de record size (y el total de su porcentaje), el total de full-page image size (y porcentaje), y el total de combined size (donde siempre será 100% por definición).

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.

Para obtener la información referente a la transacción anterior usamos **xmin**.

```
SELECT xmin FROM "Tienda";
```

	xmin	▀
	xid	
1	1385	

Como podemos ver el ID del insert anterior es 1385.

Vamos de nuevo a la terminal, y mostramos los registros creados para esta transacción:

```
Isabels-MacBook-Pro:bin isabelblanco$ ./pg_waldump -x 1385 /Library/PostgreSQL/12/data/pg_wal/00000010000002300000D5
rmgr: Heap    len (rec/tot):   92/   92, tx:     1385, lsn: 23/05A946A8, prev 23/05A94679, desc: INSERT+INIT off 1 flags 0x00, blkref #0: rel 1663/33383/33384 blk 0
rmgr: Btree   len (rec/tot):   94/   94, tx:     1385, lsn: 23/05A94708, prev 23/05A946A8, desc: NEWROOT lev 0, blkref #0: rel 1663/33383/33390 blk 1, blkref #2: rel 1663/33383/33390 blk 0
rmgr: Btree   len (rec/tot):   64/   64, tx:     1385, lsn: 23/05A94768, prev 23/05A94708, desc: INSERT_LEAF off 1 blkref #0: rel 1663/33383/33390 blk 1
rmgr: Transaction len (rec/tot): 34/   34, tx:     1385, lsn: 23/05A947A8, prev 23/05A94768, desc: COMMIT 2020-04-30 20:45:52.382194 CEST
pg_waldump: fatal: error in WAL record at 23/05A94800: invalid record length at 23/05A94838: wanted 24, got 0
```

Como podemos ver, se muestran 4 distintos registros con el ID de la transacción 1385. El primero es la operación de INSERT+INIT (heap), el segundo es el registro NEWROOT del nuevo árbol btree, el tercero es INSERT_LEAF en el que añade una hoja nueva al árbol btree, y por último el COMMIT de la transacción.

Para finalizar se nos muestra la línea de fatal error, lo que indica que hemos llegado al final del documento.

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 realizar el backup base de tienda haremos uso de pg_basebackup.exe

Primero de todo, desde la terminal haremos que el pwd sea la c, el cual se encuentra en la carpeta bin al igual que pg_waldump, el usado en los ejercicios anteriores. Carpeta bin de postgresql.

```
cd /Library/PostgreSQL/12/bin
```

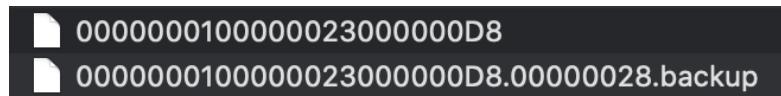
Después, tendremos que iniciar sesión como usuario de postgres en la terminal, mediante el siguiente comando:

```
sudo su - postgres
```

Una vez hecho esto, ya podemos realizar el backup. En nuestro caso lo guardaremos en el propio directorio de pg_wall.

```
[Isabels-MacBook-Pro:bin postgres$ ./pg_basebackup -D /Library/PostgreSQL/12/data/pg_wall/  
[Password:  
Isabels-MacBook-Pro:bin postgres$ ]
```

Si vamos a la carpeta de pg_wall podremos ver el nuevo archivo creado, que tendrá el mismo nombre del último archivo wal necesario para la creación de dicha base de datos, en este caso 0000000100000023000000D8.



0000000100000023000000D8
0000000100000023000000D8.00000028.backup

La segunda parte del nombre del archivo representa una posición exacta dentro del archivo WAL.

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?

PostgreSQL proporciona por defecto la herramienta MVCC (Multiversion Concurrency Control), un modelo de multiversiones. Este sistema permite a varias transacciones concurrentes tener acceso a información consistente, dando aislamiento de transacciones para cada sesión de la base de datos.

PostgreSQL dispone de otras herramientas alternativas a MVCC:

Los bloqueos explícitos son otra de esas medidas que ofrece PostgreSQL para el manejo de concurrencia. Estos bloqueos los tenemos de distintos niveles:

- ❑ **El bloqueo a nivel de tabla** bloquea la tabla por completo cuando se realizan escrituras. Tiene distintos modos de bloqueo dependiendo de las operaciones que se realicen: el del *ACCESS SHARE*, el menos restrictivo y el que deja acceder a todos menos al más restrictivo, y el *ACCESS EXCLUSIVE*, el más restrictivo que no deja acceder a ninguno. Entre medias hay variaciones que permiten distintos accesos a ciertos tipos de bloqueos, en total hay 8 bloqueos.
- ❑ **El bloqueo a nivel de fila** sirve para bloquear el acceso a los datos de las filas. Dentro de este tipo hay 4 modos para distintas operaciones en las filas que son de menos conflictivo a más conflictivo: *FOR KEY SHARE*, *FOR SHARE*, *FOR NO KEY UPDATE* y *FOR UPDATE*.
- ❑ **El bloqueo a nivel de página**, son bloqueos para el acceso de las páginas en memoria compartida.
- ❑ **El bloqueo consultivo** es un bloqueo opcional que se usa para determinadas aplicaciones, que serán las encargadas de su uso siendo bastante eficientes.

Otra herramienta que se ofrece es el aislamiento de transacciones. Hay 4 tipos: *Read uncommitted*, *Read committed*, *Repeatable read* y *Serializable*. Estos tipos van de menos restrictivos a más restrictivos.

La información que PostgreSQL puede mostrar son todos los locks que hay en el sistema, que guarda con *pg_locks*, que puede consultarse con una vista, que proporciona acceso a la información sobre los bloqueos mantenidos por procesos activos dentro del servidor de bases de datos.

Y también tiene la herramienta *pg_stat_activity*, mostrando información relacionada con la actividad actual de ese proceso, como el estado y la consulta actual y de la base de datos en general.

Si realizamos la consulta:

```
SELECT * FROM pg_catalog.pg_locks
```

Podemos ver todos los bloqueos en este momento del sistema:

locktype	database	relation	page	tuple	virtualxid	transactionid	classid	objid	objsubid	virtualtransaction	pid	mode	granted	fastpath
	oid	oid	integer	smallint	text	xid	oid	oid	smallint	text	integer	text	boolean	boolean
1	relation	24654	12143	[null]	[null]	[null]	[null]	[null]	[null]	5/83	1567	Access...	true	true
2	virtualxid	[null]	[null]	[null]	[null]	5/83	[null]	[null]	[null]	5/83	1567	Exclusiv...	true	true

No hay ninguna proceso en marcha y no se producen bloqueos. Esto se debe a que una vez hemos comprometido las transacciones, todos los bloqueos desaparecen. Además la consulta `select * from pg_locks` no puede ser un observador neutral, ya que requiere para su propio uso una transacción y un bloqueo en *pg_locks*. Como consecuencia, esta selección siempre informará al menos dos entradas, las cuales se muestran en la anterior imagen.

Si ejecutamos la consulta:

```
SELECT datid, datname, pid, application_name, wait_event_type, wait_event_state,  
       query FROM pg_stat_activity;
```

Podremos ver la actividad registrada por la base de datos.

	datid oid	datname name	pid integer	application_name text	wait_event_type text	wait_event_state text	state text	query text	
1	[null] [null]		3543		Activity	AutoVacuumMain	[null]	Read-only column	
2	[null] [null]		3546		Activity	LogicalLauncher...	[null]		
3	13637	postgres	3553	pgAdmin 4 - DB:postgres	Client	ClientRead	idle	DROP DATABASE tienda2;	
4	24654	tienda	3555	pgAdmin 4 - DB:tienda	Client	ClientRead	idle	/*pga4dash*/	
5	24654	tienda	8676	pgAdmin 4 - CONN:3476980	[null]	[null]	active	SELECT datid, datname, pid, applic...	
6	24654	tienda	5855	pgAdmin 4 - CONN:2680938	Client	ClientRead	idle		
7	24654	tienda	5857	pgAdmin 4 - CONN:258791	Client	ClientRead	idle	SET SESSION AUTHORIZATION 'us...	
8	[null] [null]		3541		Activity	BgWriterHiberna...	[null]		
9	[null] [null]		3540		Activity	CheckpointerMa...	[null]		
10	[null] [null]		3542		Activity	WalWriterMain	[null]		

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.

Mediante la interfaz de pgAdmin crearemos los usuarios. Creamos dos usuarios: usuario1 (contraseña: usuario1) y usuario2 (contraseña: usuario2).

Nos vamos a la pestaña de privilegios y le damos permiso para hacer login.

 Create - Login/Group Role

General Definition **Privileges** Membership Parameters Security SQL

Can login?	<input checked="" type="checkbox"/> Yes
Superuser?	<input type="checkbox"/> No
Create roles?	<input type="checkbox"/> No
Create databases?	<input type="checkbox"/> No
Update catalog?	<input type="checkbox"/> No
Inherit rights from the parent roles?	<input checked="" type="checkbox"/> Yes
Can initiate streaming replication and backups?	<input type="checkbox"/> No

Una vez hecho esto, tendremos que ir tabla a tabla concediendo permisos a los usuarios que queramos que tengan acceso y especificar lo que queremos que

hagan. Nos vamos a la pestaña de Security en alguna de las tablas y después le damos al (+) en privilegios y especificamos lo siguiente:

Grantee	Privileges	Grantor
<input type="button" value="Delete"/>  usuario1	<input type="checkbox"/> ALL <input type="checkbox"/> WITH GRANT OPTION <input checked="" type="checkbox"/> INSERT <input type="checkbox"/> WITH GRANT OPTION <input checked="" type="checkbox"/> SELECT <input type="checkbox"/> WITH GRANT OPTION <input checked="" type="checkbox"/> UPDATE <input type="checkbox"/> WITH GRANT OPTION <input checked="" type="checkbox"/> DELETE <input type="checkbox"/> WITH GRANT OPTION	 postgres

Ya que queremos que tenga acceso a leer y modificar.

En cada una de las tablas deberían de salirnos así los permisos:

Privileges			+
Grantee	Privileges	Grantor	
<input type="button" value="Delete"/>  postgres	arwdDxt	 postgres	
<input type="button" value="Delete"/>  usuario1	arwd	 postgres	
<input type="button" value="Delete"/>  usuario2	arwd	 postgres	

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?

```
BEGIN;

INSERT INTO public."Tienda"(
    "Id_tienda", "Nombre", "Ciudad", "Barrio", "Provincia")
VALUES (999999998, 'Mercadona', 'Alcalá de Henares', 'Ensanche', 'Madrid');

SELECT * FROM public."Tienda";
```

Si ejecutamos la consulta:

```
SELECT datid, datname, pid, application_name, wait_event_type, wait_event,state,
query FROM pg_stat_activity;
```

	datid oid	datname name	pid integer	application_name text	wait_event_type text	wait_event text	state text	query text
1	[null] [null]		3543		Activity	AutoVacuumMain [null]		
2	[null] [null]		3546		Activity	LogicalLauncherM... [null]		
3	13637	postgres	3553	pgAdmin 4 - DB:postgres	Client	ClientRead	idle	DROP DATABASE tienda2;
4	24654	tienda	3555	pgAdmin 4 - DB:tienda	Client	ClientRead	idle	/*pga4dash*/
5	24654	tienda	8676	pgAdmin 4 - CONN:3476980	[null]	[null]	active	SELECT datid, datname, pid, applicati...
6	24654	tienda	5855	pgAdmin 4 - CONN:2680938	Client	ClientRead	idle	
7	24654	tienda	5857	pgAdmin 4 - CONN:258791	Client	ClientRead	idle in transaction	SELECT n.nspname, r.relname
8	[null] [null]		3541		Activity	BgWriterHibernate [null]		
9	[null] [null]		3540		Activity	CheckpointerMain [null]		
10	[null] [null]		3542		Activity	WalWriterMain [null]		

Podemos ver que ahora en la base de datos nos sale información referente a la transacción que hemos llevado a cabo en el registro 7.

Si realizamos la consulta:

```
SELECT * FROM pg_catalog.pg_locks
```

	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
1	relation	24654	2665	[null]	[null]	[null] [null]	[null]	[null]	[null]	[null]	[null] 5/86	1567	AccessSh...	true
2	relation	24654	2664	[null]	[null]	[null] [null]	[null]	[null]	[null]	[null]	[null] 5/86	1567	AccessSh...	true
3	relation	24654	2579	[null]	[null]	[null] [null]	[null]	[null]	[null]	[null]	[null] 5/86	1567	AccessSh...	true
4	relation	24654	3455	[null]	[null]	[null] [null]	[null]	[null]	[null]	[null]	[null] 5/86	1567	AccessSh...	true
5	relation	24654	2663	[null]	[null]	[null] [null]	[null]	[null]	[null]	[null]	[null] 5/86	1567	AccessSh...	true
6	relation	24654	2662	[null]	[null]	[null] [null]	[null]	[null]	[null]	[null]	[null] 5/86	1567	AccessSh...	true

Donde podemos observar el tipo de bloqueo y que está en granted true y no en false, es decir, que no se produce bloqueo.

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.

Primero realizamos un COMMIT para cerrar la transacción anterior.

```
COMMIT;
```

Ahora, para iniciar una transacción con el usuario1, ejecutamos la siguiente consulta: `SET SESSION AUTHORIZATION 'usuario1';`

Una vez hecho esto, ya iniciamos la conexión y podremos ver que nos aparece lo siguiente al ejecutar SELECT user:

	user	name	lock
1	usuario1		

Así sabemos que lo hemos hecho de forma correcta.

El código de la consulta que se nos pide es el siguiente:

```
BEGIN;

INSERT INTO public."Tienda"(
"Id_tienda", "Nombre", "Ciudad", "Barrio", "Provincia")
VALUES (1000, 'Primaprix', 'Alcalá de Henares', 'Plaza Cervantes', 'Madrid');

INSERT INTO public."Trabajador"(
codigo_trabajador, "DNI", "Nombre", "Apellidos", "Puesto", "Salario", "Id_tienda_Tienda")
VALUES (1111111111, 222222222, 'Pedro', 'Jimenez', 'Reponedor', 700, 1000);

INSERT INTO public."Ticket"(
"N_de_ticket", "Importe", fecha, "codigo_trabajador_Trabajador")
VALUES (54321, 78.32, '03-03-2020', 1111111111);
```

Lo ejecutamos sin COMMIT.

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

Una consulta cualquiera para poder ver los datos recién insertados sería:

```
SELECT "Ticket"."N_de_ticket", "Trabajador"."Nombre", "Tienda"."Nombre"
FROM "Tienda" INNER JOIN "Trabajador" ON "Tienda"."Id_tienda" = "Trabajador"."Id_tienda_Tienda"
INNER JOIN "Ticket" ON "Trabajador".codigo_trabajador = "Ticket".codigo_trabajador_Trabajador"
WHERE "Tienda"."Id_tienda" = 1000;
```

Y podemos ver que los resultados que obtenemos coinciden con los datos que hemos introducido en la cuestión 9.

	N_de_ticket	Nombre	Nombre
	integer	character varying	text
1	54321	Pedro	Primaprix

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?

Ahora, para iniciar una transacción con el usuario2, ejecutamos la siguiente consulta: *SET SESSION AUTHORIZATION 'usuario2';*

```

1  SELECT "Ticket"."N_de_ticket", "Trabajador"."Nombre", "Tienda"."Nombre"
2  FROM "Tienda" INNER JOIN "Trabajador" ON "Tienda"."Id_tienda" = "Trabajador"."Id_tienda_Tienda"
3  INNER JOIN "Ticket" ON "Trabajador".codigo_trabajador = "Ticket"."codigo_trabajador_Trabajador"
4  WHERE "Tienda"."Id_tienda" = 1000;

```

Sin embargo, esta vez el output nos sale vacío.

	N_de_ticket integer	Nombre character varying	Nombre text

Esto se debe a que como no hemos realizado un COMMIT desde usuario1 para T1, la transacción no se ha comprometido y no puede ser accedida la información de dentro de esa consulta T1 por otro usuario.

Si ejecutamos la consulta:

```
SELECT datid, datname, pid, application_name, wait_event_type, wait_event_state,
       query FROM pg_stat_activity;
```

oid	datid	datname	pid	application_name	wait_event_type	wait_event_state	query
1	[null]	[null]	3543		Activity	AutoVacuumMain	[null]
2	[null]	[null]	3546		Activity	LogicalLauncherMain	[null]
3	13637	postgres	3553	pgAdmin 4 - DB:postgres	Client	ClientRead	idle
4	24654	tienda	3555	pgAdmin 4 - DB:tienda	Client	ClientRead	idle /*pga4dash*/
5	24654	tienda	8676	pgAdmin 4 - CONN:3476980	[null]	[null]	active
6	24654	tienda	5855	pgAdmin 4 - CONN:2680938	Client	ClientRead	idle in transaction
7	24654	tienda	5857	pgAdmin 4 - CONN:258791	Client	ClientRead	idle
8	[null]	[null]	3541		Activity	BgWriterHibernate	[null]
9	[null]	[null]	3540		Activity	CheckpointerMain	[null]
10	[null]	[null]	3542		Activity	WalWriterMain	[null]

Como podemos ver ahora nos salen dos consultas en las que estamos ejecutando el SELECT del ejercicio 10. Sin embargo, solo podemos ver datos en la consulta realizada desde la transacción, es decir en el registro 6. Ya que desde el registro 7 no podemos acceder a la información de la transacción.

Si realizamos la consulta:

```
SELECT * FROM pg_catalog.pg_locks
```

Vemos que no se producen bloqueos.

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 se encuentran los datos físicamente en las tablas de la base de datos.

Estamos ante un caso en el que tratamos de leer información parcialmente comprometida (*READ UNCOMMITTED*).

Por lo cual, dichos datos temporales se encuentran almacenados en memoria. PostgreSQL tiene modificación diferida, por lo que en cuanto se realice un COMMIT los datos guardados en memoria pasarán al disco y por tanto aparecerán físicamente en las tablas.

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?

Ahora podemos ver que realizando la consulta desde cada una de las conexiones podremos ver la información correctamente desde ambas. Esto se debe a que una vez realizamos el COMMIT de una transacción, pasa a estar almacenado en disco (los nuevos datos ya están físicamente en las tablas de la base de datos) y cualquier usuario puede ver la información actualizada.

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?

Realizamos las siguientes operaciones en usuario1, por ejemplo.

```

BEGIN;

INSERT INTO public."Tienda"(
"Id_tienda", "Nombre", "Ciudad", "Barrio", "Provincia")
VALUES (2000, 'Dia', 'Madrid', 'Sol', 'Madrid');

INSERT INTO public."Trabajador"(
codigo_trabajador, "DNI", "Nombre", "Apellidos", "Puesto", "Salario", "Id_tienda_Tienda")
VALUES (111111112, 333333333, 'Marta', 'Fernández', 'Cajero', 980, 2000);

INSERT INTO public."Ticket"(
"N_de_ticket", "Importe", fecha, "codigo_trabajador_Trabajador")
VALUES (54300, 23.92, '04-04-2020', 111111112);

UPDATE public."Trabajador"
SET "Id_tienda_Tienda"=1000
WHERE codigo_trabajador=111111112;

COMMIT;

```

Si ejecutamos de nuevo la consulta:

```

SELECT datid, datname, pid, application_name, wait_event_type, wait_event,state,
query FROM pg_stat_activity;

```

datid oid	datname name	pid integer	application_name text	wait_event_type text	wait_event text	state text	query text
1	[null] [null]	3543		Activity	AutoVacuumMain	[null]	
2	[null] [null]	3546		Activity	LogicalLauncherMain	[null]	
3	13637 postgres	3553	pgAdmin 4 - DB:postgres	Client	ClientRead	idle	DROP DATABASE tienda2;
4	24654 tienda	3555	pgAdmin 4 - DB:tienda	Client	ClientRead	idle	/*pga4dash*/
5	24654 tienda	8676	pgAdmin 4 - CONN:3476980	[null]	[null]	active	SELECT datid, datname, pid, applicatio...
6	24654 tienda	5855	pgAdmin 4 - CONN:2680938	Client	ClientRead	idle	COMMIT;
7	24654 tienda	5857	pgAdmin 4 - CONN:258791	Client	ClientRead	idle	COMMIT;
8	[null] [null]	3541		Activity	BgWriterHibernate	[null]	
9	[null] [null]	3540		Activity	CheckpointerMain	[null]	
10	[null] [null]	3542		Activity	WalWriterMain	[null]	

Podemos ver que ahora no hay ninguna transacción activa y nos aparece que la última acción realizada en la base de datos es un COMMIT.

Si realizamos la consulta:

```

SELECT * FROM pg_catalog.pg_locks

```

Vemos que no se producen bloqueos ya que tenemos una única transacción en marcha que además ha realizado un COMMIT.

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?

La consulta resultante es la siguiente:

```
BEGIN;

INSERT INTO public."Tienda"(
"Id_tienda", "Nombre", "Ciudad", "Barrio", "Provincia")
VALUES (3000, 'Carrefur', 'Alcalá de Henares', 'Avenida Complutense', 'Madrid');

INSERT INTO public."Trabajador"(
codigo_trabajador, "DNI", "Nombre", "Apellidos", "Puesto", "Salario", "Id_tienda_Tienda")
VALUES (111111113, 888888888, 'Ivan', 'Lopez', 'Reponedor', 710, 3000);

INSERT INTO public."Ticket"(
"N_de_ticket", "Importe", fecha, "codigo_trabajador_Trabajador")
VALUES (54200, 10.01, '02-03-2020', 111111113);

ROLLBACK;
```

Lo ejecutamos primero en **usuario1**. La consulta de la cuestión 10 cambiando el condición del WHERE por "Tienda"."Id_tienda" = 3000 quedaría así:

```
SELECT "Ticket"."N_de_ticket", "Trabajador"."Nombre", "Tienda"."Nombre"
FROM "Tienda" INNER JOIN "Trabajador" ON "Tienda"."Id_tienda" = "Trabajador"."Id_tienda_Tienda"
INNER JOIN "Ticket" ON "Trabajador".codigo_trabajador = "Ticket".codigo_trabajador_Trabajador"
WHERE "Tienda"."Id_tienda" = 3000;
```

El resultado de la consulta está vacío:

	N_de_ticket integer	Nombre character varying	Nombre text

Ahora ejecutaremos las dos anteriores consultas desde **usuario2**. Y el resultado es el mismo, la consulta está vacía después de hacer la transacción con el ROLLBACK. Esto se debe a que *ROLLBACK* es un comando que causa que todos los cambios de datos desde la última sentencia *BEGIN* sean descartados por el sistema de gestión de base de datos relacional para que el estado de los datos sea revertida a la forma en que estaba antes de que aquellos cambios tuvieran lugar.

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.

Ahora, para insertar los datos con el usuario1, ejecutamos la siguiente consulta: *SET SESSION AUTHORIZATION 'usuario1';*

Ejecutamos la siguiente consulta:

```
INSERT INTO public."Tienda"(  
"Id_tienda", "Nombre", "Ciudad", "Barrio", "Provincia")  
VALUES (31145, 'El Corte Inglés', 'Guadalajara', 'Plaza Uno', 'Castilla-La Mancha');  
  
INSERT INTO public."Trabajador"(  
codigo_trabajador, "DNI", "Nombre", "Apellidos", "Puesto", "Salario", "Id_tienda_Tienda")  
VALUES (45678, 777777777, 'Gerardo', 'Díaz', 'Reponedor', 600, 31145);
```

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?

Ahora, para insertar los datos con el usuario2, ejecutamos la siguiente consulta: *SET SESSION AUTHORIZATION 'usuario2';*

La consulta para actualizar el nombre de la tienda es la siguiente:

```
BEGIN;  
  
UPDATE public."Tienda"  
SET "Nombre"='Tienda Alcalá'  
WHERE "Id_tienda"=31145;
```

Si realizamos la consulta:

```
SELECT * FROM pg_catalog.pg_locks
```

Vemos que no se producen bloqueos y que se le concede *granted* en *True* a T2 de tipo exclusivo.

Para ver la actividad de la base de datos volvemos a ejecutar

```
SELECT datid, datname, pid, application_name, wait_event_type, wait_event_state,  
query FROM pg_stat_activity;
```

	datid	datname	pid	application_name	wait_event_type	wait_event	state	query
	oid	name	integer	text	text	text	text	text
1		[null] [null]	3543		Activity	AutoVacuumMain	[null]	
2		[null] [null]	3546		Activity	LogicalLauncherMain	[null]	
3	13637	postgres	3553	pgAdmin 4 - DB:postgres	Client	ClientRead	idle	
4	24654	tienda	3555	pgAdmin 4 - DB:tienda	Client	ClientRead	idle	SELECT oid, format_type(oid, NULL) AS typname FROM pg_type WH...
5	24812	tienda2	3574	pgAdmin 4 - DB:tienda2	Client	ClientRead	idle	/*pga4dash*/
6	24812	tienda2	8473	pgAdmin 4 - CONN:1626772	[null]	[null]	active	SELECT datid, datname, pid, application_name, wait_event_type, wait...
7	24654	tienda	5855	pgAdmin 4 - CONN:2680938	Client	ClientRead	idle	SET SESSION AUTHORIZATION 'usuario1';
8	24654	tienda	5857	pgAdmin 4 - CONN:258791	Client	ClientRead	idle in transaction	UPDATE public."Tienda"
9		[null] [null]	3541		Activity	BgWriterHibernate	[null]	
10		[null] [null]	3540		Activity	CheckpointerMain	[null]	
11		[null] [null]	3542		Activity	WalWriterMain	[null]	

Como podemos ver ahora en la actividad de la base de datos tienda, vemos que actualmente está activa la query UPDATE public.Tienda, ya que es la primera en acceder a ese bloque de datos.

Entre la información guardada en la base de datos no se incluye la última modificación, ya que como no hemos comprometido todavía la transacción la modificación no se refleja en físicamente en las tablas de la base de datos, pero sí en la memoria local de la transacción y en el archivo WAL.

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?

La consulta para actualizar el salario del trabajador es la siguiente:

```
BEGIN;

UPDATE public."Trabajador"
SET "Salario"=3000
WHERE codigo_trabajador=45678;
```

Para ver la actividad de la base de datos volvemos a ejecutar

SELECT datid, datname, pid, application_name, wait_event_type, wait_event,state, query FROM pg_stat_activity;

	datid	datname	pid	application_name	wait_event_type	wait_event	state	query
	oid	name	integer	text	text	text	text	text
1		[null] [null]	3543		Activity	AutoVacuumMain	[null]	
2		[null] [null]	3546		Activity	LogicalLauncherMain	[null]	
3	13637	postgres	3553	pgAdmin 4 - DB:postgres	Client	ClientRead	idle	
4	24654	tienda	3555	pgAdmin 4 - DB:tienda	Client	ClientRead	idle	/*pga4dash*/
5	24812	tienda2	3574	pgAdmin 4 - DB:tienda2	Client	ClientRead	idle	/*pga4dash*/
6	24812	tienda2	8473	pgAdmin 4 - CONN:1626772	[null]	[null]	active	SELECT datid, datname, pid, application_name, wait_event_type, wait...
7	24654	tienda	5855	pgAdmin 4 - CONN:2680938	Client	ClientRead	idle in transaction	UPDATE public."Trabajador"
8	24654	tienda	5857	pgAdmin 4 - CONN:258791	Client	ClientRead	idle in transaction	UPDATE public."Tienda"
9		[null] [null]	3541		Activity	BgWriterHibernate	[null]	
10		[null] [null]	3540		Activity	CheckpointerMain	[null]	
11		[null] [null]	3542		Activity	WalWriterMain	[null]	

Como podemos ver ahora en la actividad de la base de datos tienda, vemos que actualmente está activa la query UPDATE public.Trabajador, ya que es la primera en acceder a ese bloque de datos. También nos sale como activa la transacción realizada previamente en T2.

Si realizamos la consulta:

```
SELECT * FROM pg_catalog.pg_locks
```

Vemos que no se producen bloqueos y que se le concede granted en True a T1 de tipo exclusivo.

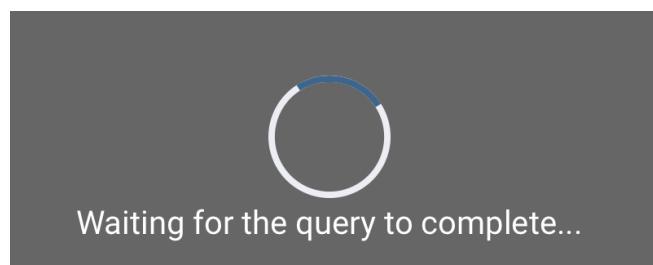
Al igual que antes, en la información guardada en la base de datos no se incluye la última modificación, ya que como no hemos comprometido todavía la transacción la modificación no se refleja en físicamente en las tablas de la base de datos, pero sí en la memoria local de la transacción y en el archivo WAL.

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?

Para modificar el puesto de capataz, la transacción T2 quedaría así:

```
BEGIN;  
  
UPDATE public."Tienda"  
SET "Nombre"='Tienda Alcalá'  
WHERE "Id_tienda"=31145;  
  
UPDATE public."Trabajador"  
SET "Puesto"='Capataz'  
WHERE codigo_trabajador=45678;
```

La consulta se queda esperando:



Esto pasa porque **T1** ya había solicitado el acceso al bloque de datos de *Trabajador*, por lo cual **T1** tiene el lock de acceso. En cuanto **T1** haga un **COMMIT** o un **ROLLBACK** se liberará el lock y **T2** podrá continuar.

Para ver la actividad de la base de datos volvemos a ejecutar

```
SELECT datid, datname, pid, application_name, wait_event_type, wait_event, state, query
FROM pg_stat_activity;
```

datid oid	datname name	pid integer	application_name text	wait_event_type text	wait_event text	state text	query text	Read-only column
1	[null] [null]	3543		Activity	AutoVacuumMain	[null]		
2	[null] [null]	3546		Activity	LogicalLauncherMain	[null]		
3	13637 postgres	3553	pgAdmin 4 - DB:postgres	Client	ClientRead	idle		
4	24654 tienda	3555	pgAdmin 4 - DB:tienda	Client	ClientRead	idle	/*pga4dash*/	
5	24812 tienda2	3574	pgAdmin 4 - DB:tienda2	Client	ClientRead	idle	SELECT oid, format_type(oid, NULL) AS typname FROM pg_type WHERE	
6	24812 tienda2	8473	pgAdmin 4 - CONN:1626772	[null]	[null]	active	SELECT datid, datname, pid, application_name, wait_event_type, wait_	
7	24654 tienda	5855	pgAdmin 4 - CONN:2680938	Client	ClientRead	idle in transaction	UPDATE public."Trabajador"	
8	24654 tienda	5857	pgAdmin 4 - CONN:258791	Lock	transactionid	active	UPDATE public."Trabajador"	
9	[null] [null]	3541		Activity	BgWriterHibernate	[null]		
10	[null] [null]	3540		Activity	CheckpointerMain	[null]		
11	[null] [null]	3542		Activity	WalWriterMain	[null]		

Esta vez podemos ver como aparece dos veces que se está tratando de acceder a UPDATE public."Trabajador" y como bien sabemos, cuando el bloqueo es de tipo exclusivo no se puede conceder acceso. Por lo cual, como podemos ver en el registro 8 de esta última ejecución, el evento es de tipo Lock (es decir, bloqueo). Mientras que el registro 7, que es de la transacción **T1** sigue como activo. Hasta que no se haga un COMMIT o un ROLLBACK en **T1** el bloqueo sobre **T2** se mantendrá activo.

Usando pg_locks también podemos ver que nos sale una consulta con granted en false, ya que es la que tiene que esperar a que se libere ese bloque de datos. Su acceso es de tipo exclusivo.

Como no hemos realizado COMMIT la información sigue sin guardarse en disco.

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?

Para modificar el barrio, la transacción T1 quedaría así:

```
BEGIN;

UPDATE public."Trabajador"
SET "Salario"=3000
WHERE codigo_trabajador=31145;

UPDATE public."Tienda"
SET "Barrio"='El Ensanche'
WHERE "Id_tienda"=31145;
```

Al igual que en el ejercicio anterior nos sale el aviso de que la consulta debe esperar. Sin embargo, como ambas transacciones se encuentran en espera se produce un interbloqueo (deadlock).

```

ERROR: deadlock detected
DETAIL: Process 5855 waits for ShareLock on transaction 585; blocked by process 5857.
Process 5857 waits for ShareLock on transaction 586; blocked by process 5855.
HINT: See server log for query details.
CONTEXT: while updating tuple (0,20) in relation "Tienda"
SQL state: 40P01

```

Por lo cual, PostgreSQL automáticamente matará una de las dos consultas para dejar el camino libre a alguna de ellas. En este caso, la transacción que PostgreSQL ha decidido matar ha sido **T1**, mientras que en **T2** ahora podemos ver el siguiente mensaje:

```
UPDATE 1
```

```
Query returned successfully in 9 min 1 secs.
```

Y podemos saber que los datos se han actualizado correctamente para esta transacción (**T2**).

Para ver la actividad de la base de datos volvemos a ejecutar

```
SELECT datid, datname, pid, application_name, wait_event_type, wait_event, state, query FROM pg_stat_activity;
```

datid oid	datname name	pid integer	application_name text	wait_event_type text	wait_event text	state text	query text
1	[null] [null]	3543		Activity	AutoVacuumMain	[null]	
2	[null] [null]	3546		Activity	LogicalLauncherMain	[null]	
3	13637 postgres	3553	pgAdmin 4 - DB:postgres	Client	ClientRead	idle	
4	24654 tienda	3555	pgAdmin 4 - DB:tienda	Client	ClientRead	idle	/*pg4dash*/
5	24812 tienda2	3574	pgAdmin 4 - DB:tienda2	Client	ClientRead	idle	SELECT oid, format_type(oid, NULL) AS typename FROM pg_type'
6	24812 tienda2	8473	pgAdmin 4 - CONN:1626772	[null]	[null]	active	SELECT datid, datname, pid, application_name, wait_event_type,'
7	24654 tienda	5855	pgAdmin 4 - CONN:2680938	Client	ClientRead	idle in transaction (aborte...	UPDATE public."Tienda"
8	24654 tienda	5857	pgAdmin 4 - CONN:258791	Client	ClientRead	idle in transaction	UPDATE public."Trabajador"
9	[null] [null]	3541		Activity	BgWriterHibernate	[null]	
10	[null] [null]	3540		Activity	CheckpointerMain	[null]	
11	[null] [null]	3542		Activity	WalWriterMain	[null]	

Ahora, podemos ver que el registro 7 el cual pertenecía a **T1**, sale su estado como abortado. Por lo cual, ahora desaparece el bloqueo (lock) que aparecía en el registro 8, que pertenecía a **T2** y tiene vía libre.

Usando pg_locks podemos ver que no hay ningún registro con granted en false. Ya que al producirse el interbloqueo automáticamente se elimina una y aparecen como granted en true. Su acceso es de tipo exclusivo.

En este momento, PostgreSQL mata/aborta **T1**, por lo que es como si todos los datos que estaban guardados en memoria desaparecieran (Undo).

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?

- T1 queda así:

```
BEGIN;

UPDATE public."Trabajador"
SET "Salario"=3000
WHERE codigo_trabajador=45678;

UPDATE public."Tienda"
SET "Barrio"='El Ensanche'
WHERE "Id_tienda"=31145;

COMMIT;
```

Una vez ejecutamos el COMMIT podemos ver que el mensaje que nos devuelve es el siguiente:

```
ROLLBACK
```

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

Es decir, que como habíamos predecido, se realiza un ROLLBACK de la transacción. Por lo cual no se grabarán los datos modificados en la consulta al disco.

- T2 queda así:

```
BEGIN;

UPDATE public."Tienda"
SET "Nombre"='Tienda Alcalá'
WHERE "Id_tienda"=31145;

UPDATE public."Trabajador"
SET "Puesto"='Capataz'
WHERE codigo_trabajador=45678;

COMMIT;
```

Una vez realizamos el COMMIT de T2, podemos ver que se ejecuta sin problema. Sabemos que los datos por lo cual se trasladan al disco y las modificaciones realizadas en dicha transacción se aplican correctamente. Por lo tanto, la tienda con **ID 31145** pasa a llamarse *Tienda Alcalá* y el trabajador con **código 45678** pasa a tener puesto *Capataz*.

A diferencia de las preguntas anteriores, ahora la información de la base de datos se refleja físicamente en la base de datos. Esto es porque una vez realizado el COMMIT, todos los datos de la memoria pasan a almacenarse en disco.

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.

Cerramos sesiones anteriores y abrimos sesión en usuario1 con

```
SET SESSION AUTHORIZATION 'usuario1';
```

La consulta resultante de la transacción T1 sería la siguiente:

```
INSERT INTO public."Tienda"(  
    "Id_tienda", "Nombre", "Ciudad", "Barrio", "Provincia")  
VALUES (6789, 'Consum', 'Alcalá de Henares', 'Avenida Guadalajara', 'Madrid');
```

```
BEGIN;  
UPDATE public."Tienda"  
    SET "Nombre"='Mediamarkt'  
    WHERE "Id_tienda"=6789;
```

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?

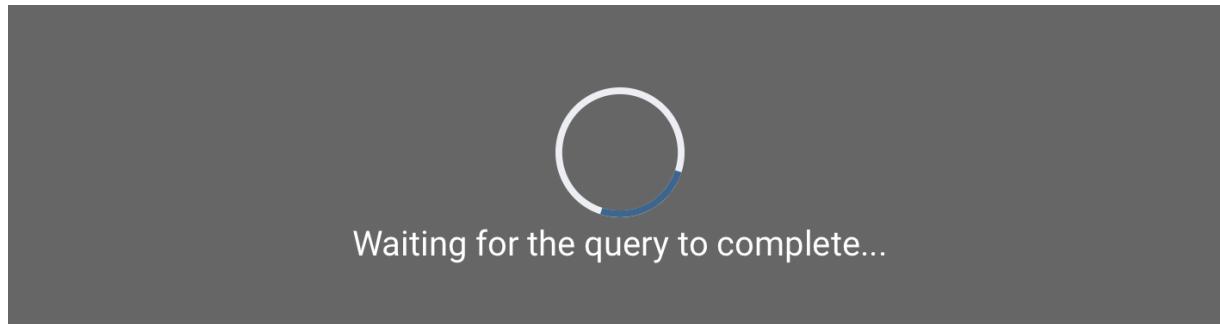
Abrimos sesión en usuario2 con

```
SET SESSION AUTHORIZATION 'usuario2';
```

La consiguiente consulta de la transacción T2 es:

```
BEGIN;  
UPDATE public."Tienda"  
    SET "Nombre"='Saturn'  
    WHERE "Id_tienda"=6789;
```

Como podemos ver, nos sale un mensaje puesto que **T2** no puede realizar la consulta hasta que termine previamente **T1**, ya que están intentando acceder al mismo bloque de datos *Tienda*.



Para ver la actividad de la base de datos volvemos a ejecutar

```
SELECT datid, datname, pid, application_name, wait_event_type, wait_event, state,
query FROM pg_stat_activity;
```

datid oid	datname name	pid integer	application_name text	wait_event_type text	wait_event text	state text	query text
1	[null] [null]	3543		Activity	AutoVacuumMain [null]		
2	[null] [null]	3546		Activity	LogicalLauncherMain [null]		
3	13637 postgres	3553	pgAdmin 4 - DB:postgres	Client	ClientRead	idle	DROP DATABASE tienda2;
4	24654 tienda	3555	pgAdmin 4 - DB:tienda	Client	ClientRead	idle	/*pga4dash*/
5	24654 tienda	8676	pgAdmin 4 - CONN:3476980	[null]	[null]	active	SELECT datid, datname, pid, applicatio...
6	24654 tienda	5855	pgAdmin 4 - CONN:2680938	Client	ClientRead	idle in transaction	UPDATE public."Tienda"
7	24654 tienda	5857	pgAdmin 4 - CONN:258791	Lock	transactionid	active	UPDATE public."Tienda"
8	[null] [null]	3541		Activity	BgWriterHibernate [null]		
9	[null] [null]	3540		Activity	CheckpointerMain [null]		
10	[null] [null]	3542		Activity	WalWriterMain [null]		

Como podemos ver, la consulta nos muestra en el registro 7, la cual corresponde a la transacción **T2** y vemos que su estado es Lock (bloqueado). Esto ocurre por lo anteriormente mencionado, ambas tratan de hacer un UPDATE public.Tienda y al ser exclusivo, hasta que no termine la transacción **T1**, **T2** seguirá bloqueada.

Si realizamos la consulta:

```
SELECT * FROM pg_catalog.pg_locks
```

Podemos ver como nos sale un registro que representa a la transacción T2 y aparece como granted en false, porque solicita bloqueo exclusivo y ya lo tenía concedido **T1**.

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?

Realizamos el COMMIT de T1: **COMMIT**;

Tras realizar esto podemos ver que la espera de T2 ha terminado y ya ha completado el UPDATE de la consulta que antes estaba esperando a que finalizara T1.

El estado de la información de la tienda con id 6789 para el usuario1 pasa a llamarse Mediamarkt, mientras que para el usuario2, sigue siendo Saturn, ya que leyó la información antes del COMMIT de T1 y es la información que se encuentra en memoria. En caso de que se hiciera un SELECT desde usuario2, leería Mediamarkt ya que es el valor que está en disco.

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: **COMMIT;**

Ahora sí que se ha cambiado la información de la tienda Id_tienda=6789: su nombre ha pasado de ser 'Mediamarkt' a 'Saturn' ya que, al comprometer T2, esa información se ha grabado en disco y se sustituye al valor escrito por T1 ya que este COMMIT es el último,

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.

En el usuario1 ejecutamos la siguiente consulta para T1:

```
BEGIN;  
UPDATE public."Ticket"  
SET "N_de_ticket"=223560  
WHERE "N_de_ticket"=54321;
```

En el usuario2 ejecutamos la siguiente consulta para T2:

```
BEGIN;  
UPDATE public."Ticket"  
SET fecha=(SELECT current_date)  
WHERE "N_de_ticket"=54321;
```

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?

Realizamos el COMMIT de T1:

COMMIT;

Antes de hacer el COMMIT, T2 estaba en espera ya que estaba intentando acceder a *Ticket*, y como ya tenía T1 permisos sobre *Ticket*.

Una vez comprometida la transacción, podemos ver que T2 tiene vía libre y realiza la actualización. Sin embargo, no devuelve ninguna actualización:

```
UPDATE 0
```

```
Query returned successfully in 3 min 41 secs.
```

Esto pasa porque el COMMIT de **T1** pasa antes de que **T2** llegue a ejecutar el código para actualizar Ticket. Entonces, el id del ticket se cambia por 223560 antes de que **T2** haga su UPDATE. Entonces, no encuentra ningún ticket con id 54321 ya que es el valor desactualizado de ticket.

Se ha comprometido **T1**, por lo que se ha grabado en disco la nueva información que ha aportado esta transacción, es decir, ahora:

“N_de_ticket”=223560.

Sin embargo, **T2** ha leído esta tienda antes de que se realizase la modificación y se comprometiera, por lo que en **T2** al no poder aplicar la actualización, sigue siendo la fecha antigua.

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?

Realizamos el COMMIT de **T2**:

```
COMMIT;
```

Como hemos explicado antes, ningún valor se ha actualizado en la transacción T2, por lo cual aunque el COMMIT se realiza de forma correcta, en realidad ningún cambio se ha producido en la transacción.

Para el antiguo ticket de id 54321, para todos los usuarios pasa a ser el ticket con id 223560. Además, ya que no se ha llegado a realizar la modificación de T2, la fecha del ticket sigue siendo la antigua.

Lo comprobamos ejecutando la siguiente consulta:

```
SELECT "N_de_ticket", "fecha"  
FROM "Ticket"  
WHERE "N_de_ticket" = 223560;
```

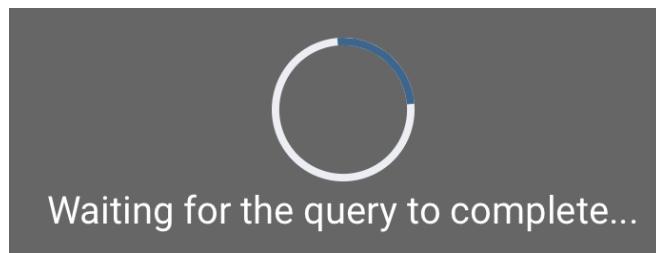
Como podemos comprobar por el output de la consulta estábamos en lo cierto.

	N_de_ticket [PK] integer	fecha date
1	223560	2020-03-03

La fecha sigue siendo la antigua (no la actual, que debería ser 2020-05-30). Y el Nº del ticket se ha actualizado.

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?

El sistema gestor de base de datos se paraliza ya que está en espera. Nos sale el siguiente mensaje:



Lo que significa que hasta que no realicemos el COMMIT o un ROLLBACK en la transacción que trata de modificar los datos del ticket, la transacción que trata de eliminar dicho ticket quedará bloqueada en espera de que termine la primera transacción.

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?

Antes que todo nos vamos al archivo de postgresql.conf y nos dirigimos a la sección de -Archive Recovery-. Cambiamos el restore_command y pondremos la ruta '`/Users/isabelblanco/Desktop/wal_archive/%f %p`', ya que hemos puesto nuestro archivo de backup en una carpeta llamada wal_archive.

Configuración previa:

```
# - Archive Recovery -
# These are only used in recovery mode.

#restore_command = ''          # 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)
#archive_cleanup_command = ''   # command to execute at every restartpoint
#recovery_end_command = ''     # command to execute at completion of recovery
```

Configuración final:

```
# - Archive Recovery -
# These are only used in recovery mode.

restore_command = 'cp /Users/isabelblanco/Desktop/wal_archive/%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)
# command to execute at every restartpoint
# command to execute at completion of recovery
```

Para restaurar la copia de seguridad desde el backup realizado en la cuestión 6 debemos seguir los siguientes pasos.

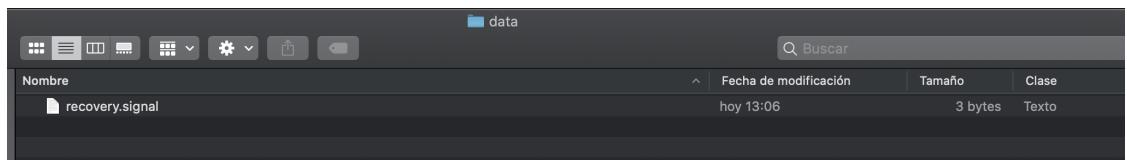
1. Detenemos el servidor en caso de que esté ejecutado.

```
Isabels-MacBook-Pro:~ postgres$ cd /Library/PostgreSQL/12/bin
Isabels-MacBook-Pro:bin postgres$ ./pg_ctl stop -D /Library/PostgreSQL/12/data/
waiting for server to shut down.... done
server stopped
Isabels-MacBook-Pro:bin postgres$
```

2. Copiamos la carpeta actual de /Library/PostgreSQL/12/data en un sitio seguro por si algo sale mal. (En /Users/isabelblanco/Desktop/data)
3. Eliminamos el contenido del clúster (carpeta data) para asegurarnos también de que la carpeta pg_wal queda vacía.

```
Isabels-MacBook-Pro:bin postgres$ rm -rf /Library/PostgreSQL/12/data/*
```

4. Comprobamos que los permisos de la carpeta son correctos y creamos un archivo llamado `recovery.signal` en la carpeta actualmente vacía de data.



5. Iniciamos el servidor. Este entrará en modo recuperación y leerá los archivos de WAL que necesite. Cuando el proceso de recuperación termine, el archivo que habíamos creado llamado `recovery.signal` habrá sido eliminado por el servidor.

Para comprobar que los pasos anteriormente mencionados funcionaron, vamos a ver la información que podemos ver en la base de datos. Si es correcto, deberíamos poder ver lo que hicimos antes del ejercicio 6.

Puesto que al principio introducimos una tienda, ejecutaremos una consulta para ver las tiendas que tiene nuestra base de datos:

```
SELECT * FROM public."Tienda";
```

El output de la consulta es el siguiente:

	Id_tienda [PK] integer	Nombre text	Ciudad text	Barrio text	Provincia text
1	999999999	Supeco	Madrid	Carabanchel	Madrid

Como podemos ver, la base de datos solo tiene el registro que le insertamos en la cuestión 2, por lo que el backup se ha hecho de forma correcta.

Además, si miramos el log podemos ver el siguiente mensaje:

```
2020-30-05 13:25:56.362 UTC [8576] LOG: restored log file  
"000000010000000000000002" from archive
```

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 secuenciales? ¿Genera siempre planificaciones recuperables? ¿Tiene rollbacks en cascada? Justificar las respuestas.

PostgreSQL tiene recuperación basada en log del tipo **modificación diferida**, como hemos podido comprobar observando cómo los datos sólo se reflejaban físicamente en la base de datos al realizar el COMMIT de la transacción.

Como protocolo de gestión de concurrencia utiliza los **bloqueos**. Esto lo sabemos ya que hemos visto como PostgreSQL bloquea una transacción y hace que pase a tener *granted* a *true* solo cuando la transacción que llega primero al bloque de datos libere el bloqueo.

Siempre genera **planificaciones secuenciales** porque tenemos que comprometer los datos de cada transacción en un orden determinado para ser capaces de acceder al mismo bloque de información en cada una de las transacciones.

Siempre genera **planificaciones recuperables** ya que cuando hay una transacción (**T_i**) en marcha accediendo a un bloque de datos determinado bloquea cualquier transacción (**T_j**) que venga detrás siempre y cuando el tipo de transacción de ambas no solicite lectura, ya que entonces no se produciría ningún bloqueo. Por lo cual, podemos demostrar que el COMMIT de la que accede primero al bloque de datos (**T_i**) tiene que comprometer sus datos antes que **T_j**, ya que **T_j** se bloqueará y no se liberará hasta entonces.

Que tenga **rollbacks en cascada** o no depende de la planificación en cuestión. Imaginemos la siguiente situación: Abrimos una transacción **T_i** que escribe el bloque de datos A. Abrimos otra transacción **T_j** que lee el bloque de datos A. Justo después realizamos el commit de **T_i**. En esta situación, sí que tendría rollback en cascada. Sin embargo, imaginemos que el COMMIT de **T_i** fuese justo delante de la lectura de **T_j**. En este caso, no tendría 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.**