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: Patricia Cuesta Ruiz

DNI: 03211093V

ALUMNO 2:

Nombre y Apellidos: Álvaro Golbano Durán

DNI: 03202759D

Fecha:

Profesor Responsable: Óscar Gutiérrez

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?

El directorio donde se encuentra en la carpeta de pg_wall dentro del directorio data de PostgreSQL, el tamaño del archivo es de 16KB de tamaño y al abrirlo con un editor de textos podemos ver que es imposible interpretar la información que contiene ya que esta codificado en hexadecimal. Muestra desde que se ha arrancado la base de datos lo que ha ocurrido.

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?

Al ejecutar la siguiente consulta:

```
insert into "Tienda" values (1,'tienda1','ciudad1','barrio1','provincia1');
```

podremos ver que en el log sale lo siguiente:

```
2020-06-01 17:58:58.969 CEST [13512] SENTENCIA:  insert into Tienda values (1,'tienda1','ciudad1','barrio1','provincia1')
```

Como podemos observar, sí se ve reflejada la operación en el archivo del sistema log, pero no se puede ver nada en el wal ya que está codificado en hexadecimal.

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 sirve para mostrar por pantalla el diario del sistema de una forma entendible, decodificando el archivo. El comando para insertar es:

```
PS D:\Universidad\BBDD Av\bin> .\pg_waldump -z --path="D:\Universidad\BBDD Av\data\pg_wal" 000000010000000000000D5
```

Mostrando por pantalla este resultado:

Type	N	(%)	Record size	(%)	FPI size	(%)	Combined size	(%)
XLOG	144	(0,25)	15552	(0,49)	960	(0,01)	16512	(0,12)
Transaction	12	(0,02)	9787	(0,31)	0	(0,00)	9787	(0,07)
Storage	23	(0,04)	966	(0,03)	0	(0,00)	966	(0,01)
CLOG	0	(0,00)	0	(0,00)	0	(0,00)	0	(0,00)
Database	4	(0,01)	152	(0,00)	0	(0,00)	152	(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	253	(0,44)	12710	(0,40)	0	(0,00)	12710	(0,09)
Heap2	23	(0,04)	1288	(0,04)	0	(0,00)	1288	(0,01)
Heap	55866	(97,93)	3056456	(97,10)	10433212	(99,29)	13489668	(98,79)
Btree	720	(1,26)	50874	(1,62)	73288	(0,70)	124162	(0,91)
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	57045		3147785	[23,05%]	10507460	[76,95%]	13655245	[100%]

Con este resultado podemos contemplar el numero de veces que se han utilizado todos los tipos de instrucciones.

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.

Primero debemos averiguar cual ha sido el último identificador de la transacción:

```
select txid_current();
```

Siendo el resultado:

	txid_current bigint
1	523

Después ejecutamos el siguiente comando:

```
PS D:\Universidad\BBDD Av\bin> .\pg_waldump  --path="D:\Universidad\BBDD Av\data\pg_wal" --xid=523 000000010000000000000D5
rmgr: Transaction len (rec/tot):     34/    34, tx:        523, lsn: 0/D5D2D1D0, prev 0/D5D2D198, desc: COMMIT 2020-06-01 18:28:23.878910 Hora de verano romance
```

Nos devuelve que la última operación que se ha realizado es un COMMIT y la cantidad de registros que se han hecho si se operan lsn-prev en hexadecimal.

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 *"Continous Archiving and point-in-time recovery (PITR)"*.

Creamos un backup llamando a la función pg_start_backup():

```
select pg_start_backup('b1', false, false);
```

Mostrando:

	pg_start_backup pg_lsn
1	0/D6000028

Y luego ejecutamos:

```
select pg_stop_backup(false);
```

Cuestión 6: Qué herramientas disponibles tiene PotsgreSQL 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?

Internamente, PostgreSQL mantiene la consistencia de los datos a los que acceden las transacciones utilizando MVCC (Multiversion concurrency control). MVCC permite que las instrucciones SQL vean una imagen de los datos, sin contar con su estado actual.

Además, PotsgreSQL nos proporciona 3 herramientas para controlar las transacciones:

- **Aislamiento de Transacción: se definen 4 niveles de aislamiento (Read uncommited, Read commited, Repeatable read, Serializable). Serializable es el nivel más restrictivo. El resto se definen en fenómenos resultantes de la interacción entre distintas transacciones que no debe ocurrir en cada nivel.**
- **Cerrojos Explícitos: PostgreSQL provee distintos modos de bloqueo para controlar el acceso concurrente a los datos por las transacciones. Estos modos**

pueden ser utilizados para usar bloqueos controlados cuando MVCC no se comporte como nosotros deseamos. Además, casi todos los comandos en PostgreSQL adquieren locks de los modos apropiados para asegurarse de que las tablas no se eliminan o modifican.

- **Revisión de la Consistencia de los Datos a Nivel de Aplicación: es muy difícil imponer reglas de negocio en relación a la integridad de los datos utilizando transacciones Read Commited ya que la vista de los datos está cambiando con cada instrucción.**

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

Para crear dos usuarios es necesario poner primero el rol necesario para los usuarios y garantizar a ese rol los permisos de lectura y escritura:

```
create role usuario with login password 'usuario';
grant select, insert, update, delete on all tables in schema public to usuario;
```

Posteriormente creamos a los usuarios 1 y 2:

```
create user usuario1 with role usuario password 'usuario1';
create user usuario2 with role usuario password 'usuario2';
```

Y les garantizamos los permisos:

```
grant select, insert, update, delete on all tables in schema public to usuario1;
grant select, insert, update, delete on all tables in schema public to usuario2;
```

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?

La transacción será la siguiente:

```
begin;
insert into "Tienda" values (2, 'tienda2', 'ciudad2', 'barrio2','provincia2');
select * from "Tienda";
```

	ID_Tienda [PK] integer	Nombre text	Ciudad text	Barrio text	Provincia text
1	1	tienda1	ciudad1	barrio1	provincia1
2	2	tienda2	ciudad2	barrio2	provincia2

No ponemos la palabra la COMMIT para no cerrar la transacción y no comprometer los datos introducidos. Es por eso por lo que no se puede encontrar información nueva en el archivo del sistema ya que la transacción sigue activa.

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.

Habiendo previamente iniciado sesión con usuario1. Abrimos una nueva transacción y operamos:

```
begin;
insert into "Tienda" values (1000, 'tienda1000', 'ciudad1000', 'barrio1000','provincia1000');
insert into "Trabajador" values (1, '1', 'nombre1', 'apellido1', 'puesto1', 1, 1000);
insert into "Ticket" values (54321, 1, '01-01-2001', 1);
```

Como no ponemos COMMIT, la transacción no ha finalizado aún.

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

Si hacemos SELECT dentro de la transacción si nos devuelve la tienda, el trabajador y el ticket que acabamos de insertar, pero si lo ejecutamos en otra consulta distinta no, esto se debe a que no hemos realizado commit en nuestra transacción y por lo tanto los datos no se han escrito en memoria global.

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?

Cuando establecemos una nueva conexión y realizamos la consulta sobre las tablas, nos aparecen sin los cambios realizados por parte del usuario1, esto se debe a que las inserciones que hemos realizado en las transacciones anteriores no han realizado commit, por lo que los datos no se han escrito en las tablas y no quedan reflejados los cambios todavía.

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

Los datos no se encuentran físicamente en la base de datos, sino que se almacenan temporalmente en una memoria exclusiva para la transacción y saldrán a memoria global cuando se realice commit en las transacciones y escriban los datos.

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?

Una vez añadimos commit al final de la transacción T1, cuando realizamos la select desde otra transacción esta vez sí que podemos ver los nuevos datos introducidos en la transacción T1 ya que han comprometido y se han mandado a memoria global, donde la consulta select los ha leído.

Lo mismo pasa con la consulta desde otro usuario, esta vez sí nos aparecen los datos cuando ejecutamos la consulta select, debido a que los datos se han comprometido en memoria global y se han escrito en la base de datos.

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?

La transacción que realizar es la siguiente:

```
begin;
insert into "Tienda" values (2000, 'tienda2000', 'ciudad2000', 'barrio2000','provincia2000');
insert into "Trabajador" values (2, '2', 'nombre2', 'apellido2', 'puesto2', 2, 2000);
insert into "Ticket" values (54300, 1, '02-02-2002', 2);
update "Trabajador" set "ID_Tienda_Tienda" = 1000 where "Codigo_trabajador" = 2;
commit;
```

El estado final de la base de datos será que se han actualizado los datos ya que la transacción ha finalizado correctamente. Ahora ya las inserciones y modificaciones pertinentes se encontrarán en la memoria global de la base de datos.

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?

Abrimos la transacción y realizamos las siguientes consultas:

```
begin;
insert into "Tienda" values (3, 'tienda3', 'ciudad3', 'barrio3','provincia3');
insert into "Trabajador" values (3, '3', 'nombre3', 'apellido3', 'puesto3', 3, 3);
insert into "Ticket" values (3, 3, '03-03-2003', 3);
rollback;
```

Cuando realizamos la transacción, pero al final incluimos la palabra rollback se deshacen los cambios. Rollback se usa para deshacer los cambios hechos durante una transacción, por lo que si ejecutamos rollback al final de la consulta no se comprometerán los datos modificados durante esa transacción. Viéndose la base de datos final, las tablas igual que antes de esta transacción.

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.

Para insertar esa información introducimos estas inserciones:

```
insert into "Tienda" values (31145, 'tienda31145', 'ciudad31145', 'barrio31145','provincia31145');
insert into "Trabajador" values (45678, '45678', 'nombre45678', 'apellido45678', 'puesto45678', 45678, 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?

```
begin;
update "Tienda" set "Nombre" = 'Tienda Alcala' where "ID_Tienda" = 31145;
```

La actividad registrada en la base de datos actualmente es que la transacción del usuario2 está en estado IDLE, ya que no se ha comprometido todavía. La información guardada en la base de datos es la misma que antes de comenzar la transacción porque los datos no se han comprometido todavía.

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?

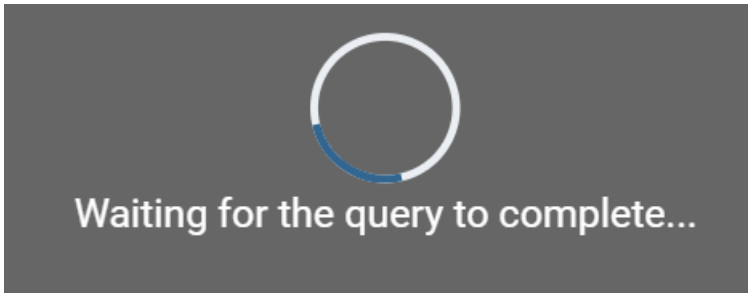
```
begin;
update "Trabajador" set "Salario" = 3000 where "Codigo_trabajador" = 45678;
```

Al igual que en el ejercicio anterior, la transacción está en estado IDLE y la información guardada es la misma que antes de realizar la transacción ya que los datos no se han comprometido.

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?

```
update "Trabajador" set "Salario" = 3000 where "Codigo_trabajador" = 45678;
```

No aparece ninguna actividad registrada ya que la transacción se queda esperando a que T1 termine porque sino no puede realizar el update. Ninguna de esta información se guarda en la base de datos ya que no se realiza ningún commit.



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?

```
update "Tienda" set "Barrio" = 'El Ensanche' where "ID_Tienda" = 31145;
```

En este caso, al ejecutar la consulta en la transacción T1 y teniendo en cuenta que T2 antes de realizar esta consulta ya había hecho una modificación de la misma tienda generando así un interbloqueo o deadlock, como se puede observar en la siguiente imagen:

```
ERROR:  se ha detectado un deadlock
DETAIL:  El proceso 3304 espera ShareLock en transacción 567; bloqueado por proceso 14240.
El proceso 14240 espera ShareLock en transacción 568; bloqueado por proceso 3304.
HINT:  Vea el registro del servidor para obtener detalles de las consultas.
CONTEXT:  mientras se actualizaba la tupla (0,11) en la relación «Tienda»
SQL state: 40P01
```

La información guardada en la base de datos sigue sin cambiar.

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?

Tabla Tienda:

5	31145	Tienda Alcala	ciudad311...	barrio311...	provincia31145

Tabla Trabajador:

3	45678	45678	nombre45678	apellido45678	puesto45678	3000	31145

Los valores finales de Tienda son los cambios realizados por la transacción T2, ya que la modificación que se realiza en T1 produjo el deadlock, por lo que al hacer commit se hace rollback hasta antes de que se produzca ese error. Es por eso por lo que en la tabla Trabajador sí vemos reflejados cambios realizados por la transacción T1 porque se producen antes del error que produce el deadlock.

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.

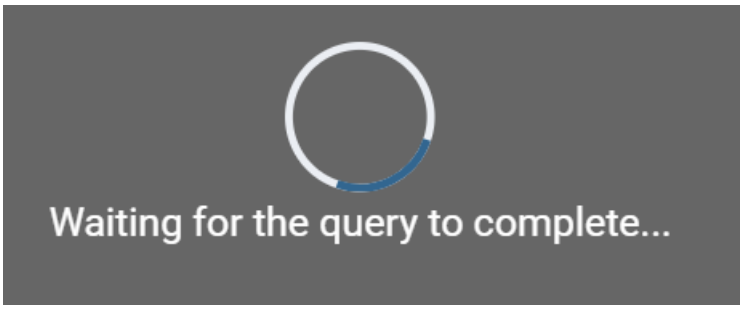
Insertamos la nueva tienda y comenzamos la transacción T1 modificando su nombre sin cerrar dicha transacción:

```
insert into "Tienda" values (6789, 'tienda6789', 'ciudad6789', 'barrio6789','provincia6789');

begin;
update "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?

```
begin;
update "Tienda" set "Nombre" = 'Saturn' where "ID_Tienda" = 6789;
```



En este caso la transacción T2 se queda en espera a que termine T1 ya que espera a que se libere el lock que hay en T1.

Cuando comprobamos la información sobre la actividad de ambas transacciones vemos que ambas están a la espera. T1 está a la espera de ser comprometida y T2 está a la espera de que termine T1 y libere el lock de Tienda y reanudarse.

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?

6	6789	Mediamarkt	ciudad6789	barrio6789	provincia6789

Al comprometer los datos de la transacción T1, se ha cambiado el nombre de la tienda a Mediamarkt y si miramos T2 el mensaje de espera "Waiting for the query to complete…" ya no aparece en pantalla y se queda a la espera de que se realice un commit sobre los datos modificados en T1.

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?

6	6789	Saturn	ciudad6789	barrio6789	provincia6789

Al realizar commit en la transacción T2 se puede observar que el nombre de la tienda ha sido modificado a Saturn ya que los datos de T2 han sido comprometidos.

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.

```
/* T1 */
begin;
update "Ticket" set "No_de_tickect" = 223560 where "No_de_tickect" = 54321;

/* T2 */
begin;
update "Ticket" set "Fecha" = '02-06-2020' where "No_de_tickect" = 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?

Tabla Ticket:

2	223560	1	2001-01-01	1

Al comprometer la transacción T1, el valor de No_de_tickect se ve comprometido y se sobrescribe el valor anterior en memoria global. Para el usuario2 que está ejecutando T2 la transacción deja de esperar a T1 y se queda en espera para realizar el commit.

Si comprobamos los datos para T2, se puede observar que el ticket con código 54321 ya no existe.

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?

Al haber modificado en T1 el código de ticket y al haberle dado otro valor, T2 a pesar de que haga commit, ningún dato se verá modificado ya que el gestor de la base de datos al no encontrar la tupla a modificar no la tiene en cuenta.

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?

```
begin;
update "Ticket" set "Importe" = 666 where "No_de_tickect" = 223560;
    begin;
    delete from "Ticket" where "No_de_tickect" = 223560;
```

Si realizásemos esto recibiríamos un aviso WARNING, ya hay una transacción en curso, por lo que no se iniciaría una nueva transacción porque PostgreSQL no soporta sub-transacciones. Debido a esto, se obviaría el segundo BEGIN y se trataría todo como una sola transacción, dando como resultado que con un solo comando COMMIT.

Se comprometen ambas consultas por lo que primero modificaría el Importe del ticket y luego borraría dicha tupla.

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) *"Continous Archiving and point-in-time recovery (PITR)*. ¿Cuál es el estado final de la base de datos? ¿Por qué?

Para recuperar los datos primero deberemos detener el servidor de PostgreSQL.

A continuación, borraremos los archivos y subdirectorios existentes bajo el directorio de datos del cluster y bajo los directorios raíz. También eliminaremos los archivos del pg_wal, ya que quedarán obsoletos si recuperamos una versión anterior de la base de datos.

Deberemos crear un archivo recovery.conf, que al terminar el proceso el servidor renombrará a recovery.done.

Lo más importante de dicho archivo recovery.conf es el siguiente comando:

restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"'

Donde especificaremos la dirección de nuestro archivo de backup.

Una vez realizado esto, iniciaremos el servidor PostgreSQL y observaremos que los archivos se van recuperando del WAL, dando lugar a la recuperación de la base de datos tal y como estaba en la cuestión 5.

Si consultamos la información de cualquier tabla de la base de datos, la única que existe es la tienda con código = 1 que se creó en la cuestión 2. Nada de lo hecho posteriormente al backup aparecerá.

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.

Para la recuperación PostgreSQL tiene implementado el WAL, el cual utiliza para registrar las acciones de las consultas y transacciones y poderse recuperar a partir de dicho archivo en caso de algún tipo de fallo.

PostgreSQL cuenta con Modificacion Diferida, pues solo manda los cambios a memoria global cuando hace COMMIT una transacción, por lo que solo ejecuta REDO en caso de error sobre las transacciones cuyo begin y commit se encuentre en el WAL.

En cuanto a la gestión de concurrencia, PostgreSQL utiliza MVCC lo cual permite que cada transacción observe una imagen de los datos, no el estado actual de estos. Además, cuenta con otras 3 herramientas como el Aislamiento de Transacción, los Cerrojos Explícitos y la Revisión de la Consistencia. Todo esto permite que las transacciones se ejecuten de forma concurrente y de forma segura sin formar inconsistencias en la base de datos.

Cuando hablamos de la secuenciabilidad, en PostgreSQL no siempre son secuenciables las transacciones. Como hemos podido leer en la documentación, existen 4 niveles de aislamiento de las transacciones. Uno de ellos es Serializable Isolation Level, que lo que hace es ejecutar las transacciones de forma totalmente secuencial, una tras otra emulando una ejecución secuencial.

En PostgreSQL no se produce rollback en cascada cuando una transacción falla, pues como hemos visto en algunas preguntas, al ocurrir un fallo en una transacción, dicha transacción falla y se deshacen sus cambios, pero no se deshacen los cambios del resto de transacciones, estas continúan. Al no tener rollback en cascada, las planificaciones son recuperables.

Bibliografía

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