TITULACIÓN:	GRADO EN INGENIERÍA INFORMÁTICA Y SISTEMAS DE INFORMACIÓN
CURSO:	2018-2019. CONVOCATORIA ORDINARIA DE ENERO
ASIGNATURA:	BASES DE DATOS AVANZADAS – LABORATORIO

PRACTICA 3: SEGURIDAD, USUARIOS Y TRANSACCIONES.

ALUMNO 1:

Nombre y Apellidos: Marcos Barranquero Fernández

DNI: 51129104N

ALUMNO 2:

Nombre y Apellidos: Eduardo Graván Serrano

DNI: 03212337L

Fecha: 17/12/2018

Profesor Responsable: Iván González Diego

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 práctica como Suspenso – Cero.

PLAZOS

Tarea en laboratorio:	Semana 3 de Diciembre, Semana 10 de Diciembre y semana 17 de Diciembre.
Entrega de práctica:	Día 9 de Enero. Aula Virtual
Documento a entregar:	Este mismo fichero con las respuestas a las cuestiones planteadas, el script de planificación de la seguridad y un script con las pruebas realizadas sobre la seguridad. 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 dos temas:

- la planificación de la seguridad de la base de datos. Deberá de planificarse la seguridad de la base de datos de manera que los usuarios puedan realizar las operaciones permitidas, pero de forma que ningún usuario no autorizado pueda tocar ningún otro dato sensible de operación (cualquier otra tabla/columna a la que no se le ha concedido acceso). Además, la información a la que deberá poder acceder cada tipo de usuario será la mínima necesaria para poder realizar las operaciones correspondientes, de forma que se deberá ocultar al usuario aquella información a la que no deba tener acceso.

- 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.

ACTIVIDADES Y CUESTIONES PARTE 1

CUESTIÓN 1.1: DETERMINAR LOS ROLES DE USUARIOS QUE VAN A PODER ACCEDER A LA BASE DE DATOS **MUSICOS**. RELLENAR LA TABLA QUE SE MUESTRA A CONTINUACIÓN (3 MÍNIMO).

Roles	Características	Comentarios
Administrador	Control total.	Administra la base de datos “Musicos”.
Productor	Puede acceder y modificar información sobre músicos, grupos, discos y canciones.	Solo podrá acceder sin modificar a las tablas relacionadas con los conciertos y entradas.
Organizador	Puede acceder y modificar las tablas que almacenan información sobre los conciertos y las entradas.	Contraparte al productor. No puede modificar la información referente a los músicos.
Músico	Acceso de lectura a casi toda la base de datos, con excepciones.	Solo puede modificar datos referentes a sí mismo.

CUESTIÓN 1.2: PLANIFICAR LA SEGURIDAD DE LA BASE DE DATOS PARA CADA UNO DE LOS ROLES DE USUARIOS. RELLENAR LA TABLA QUE SE MUESTRA A CONTINUACIÓN (ACCIONES SELECT, INSERT, DELETE, UPDATE, ETC)

Roles	Tabla	Acción	Comentarios
Administrador, Productor, Organizador, Músico	Músicos	Select	Todos los usuarios tienen permiso de lectura de esta tabla.
Administrador, Productor	Músicos	Insert, Delete	
Administrador, Productor, Músico	Músicos	Update	Los músicos pueden actualizar información referente a sí mismos.
Administrador, Productor, Organizador, Músico	Grupo	Select	Todos los usuarios tienen permiso de lectura de esta tabla.
Administrador, Productor	Grupo	Insert, Delete	
Administrador, Productor	Grupo	Update	
Administrador, Productor, Organizador, Músico	Conciertos	Select	Todos los usuarios tienen permiso de lectura de esta tabla.
Administrador, Organizador	Conciertos	Insert, Delete	
Administrador, Organizador	Conciertos	Update	
Administrador, Productor, Organizador	Entradas	Select	El músico es el único que no puede leer información sobre las entradas.
Administrador, Organizador	Entradas	Insert, Delete	

Administrador, Organizador	Entradas	Update	
Administrador, Productor, Organizador, Músico	Discos	Select	Todos los usuarios tienen permiso de lectura de esta tabla.
Administrador, Productor	Discos	Insert, Delete	
Administrador, Productor	Discos	Update	
Administrador, Productor, Organizador, Músico	Canciones	Select	Todos los usuarios tienen permiso de lectura de esta tabla.
Administrador, Productor	Canciones	Insert, Delete	
Administrador, Productor	Canciones	Update	
Administrador, Productor, Organizador, Músico	Grupos_Tocan_Conciertos	Select	Todos los usuarios tienen permiso de lectura de esta tabla.
Administrador, Productor, Organizador	Grupos_Tocan_Conciertos	Insert, Delete	
Administrador, Productor, Organizador	Grupos_Tocan_Conciertos	Update	

CUESTIÓN 1.3: IMPLEMENTAR LA SEGURIDAD DE LA BASE DE DATOS PARA CADA UNO DE LOS ROLES DE USUARIOS.

Se crean los roles de usuario con las siguientes sentencias SQL:

```
CREATE ROLE "Administrador" WITH CREATEROLE INHERIT REPLICATION;
CREATE ROLE "Productor;
CREATE ROLE "Organizador";
CREATE ROLE "Musico";
```

Después, con ayuda de la interfaz gráfica del pgadmin, se ejecutan las siguientes consultas SQL para asignarles los permisos que se describen en la tabla anterior a cada uno de los roles:

```
GRANT ALL ON TABLE public."Canciones" TO "Administrador" WITH GRANT OPTION;
GRANT SELECT ON TABLE public."Canciones" TO "Musico";
GRANT SELECT, UPDATE, DELETE, TRIGGER ON TABLE public."Canciones" TO "Productor";
GRANT SELECT ON TABLE public."Canciones" TO "Organizador";
GRANT ALL ON TABLE public."Conciertos" TO "Administrador" WITH GRANT OPTION;
GRANT INSERT, SELECT, UPDATE, DELETE, TRIGGER ON TABLE public."Conciertos" TO "Organizador";
GRANT SELECT ON TABLE public."Conciertos" TO "Productor";
GRANT SELECT ON TABLE public."Conciertos" TO "Musico";
GRANT ALL ON TABLE public."Discos" TO "Administrador" WITH GRANT OPTION;
GRANT SELECT ON TABLE public."Discos" TO "Organizador";
GRANT INSERT, SELECT, UPDATE, DELETE, TRIGGER ON TABLE public."Discos" TO "Productor";
GRANT SELECT ON TABLE public."Discos" TO "Musico";
GRANT ALL ON TABLE public."Entradas" TO "Administrador" WITH GRANT OPTION;
GRANT INSERT, SELECT, UPDATE, DELETE, TRIGGER ON TABLE public."Entradas" TO "Organizador";
GRANT SELECT ON TABLE public."Entradas" TO "Productor";
GRANT ALL ON TABLE public."Grupo" TO "Administrador" WITH GRANT OPTION;
GRANT INSERT, SELECT, UPDATE, DELETE, TRIGGER ON TABLE public."Grupo" TO "Productor";
GRANT SELECT ON TABLE public."Grupo" TO "Organizador";
GRANT SELECT ON TABLE public."Grupo" TO "Musico";
GRANT ALL ON TABLE public."Grupos_Tocan_Conciertos" TO "Administrador" WITH GRANT OPTION;
GRANT INSERT, SELECT, UPDATE, DELETE, TRIGGER ON TABLE public."Grupos_Tocan_Conciertos" TO
"Organizador";
GRANT INSERT, SELECT, UPDATE, DELETE, TRIGGER ON TABLE public."Grupos_Tocan_Conciertos" TO
"Productor";
GRANT SELECT ON TABLE public."Grupos_Tocan_Conciertos" TO "Musico";
GRANT ALL ON TABLE public."Musicos" TO "Administrador" WITH GRANT OPTION;
GRANT INSERT, SELECT, UPDATE, DELETE, TRIGGER ON TABLE public."Musicos" TO "Productor";
GRANT SELECT ON TABLE public."Musicos" TO "Organizador";
GRANT SELECT, UPDATE, TRIGGER ON TABLE public."Musicos" TO "Musico";
```

CUESTIÓN 1.4: SUPONER QUE LA BASE DE DATOS **MUSICOS** TIENE MILES DE MÚSICOS QUE QUIEREN ACCEDER A LA BASE DE DATOS PARA PODER CONSULTAR SUS DATOS Y SUS DISCOS. ¿DE QUÉ MANERAS SE PODRÍA GESTIONAR EL ACCESO DE LOS DIFERENTES USUARIOS A LA BASE DE DATOS? COMENTAR VENTAJAS E INCONVENIENTES.

Haciendo que el rol de músicos en este caso no tenga limite de conexiones concurrentes, o creando un usuario para cada músico en concreto que herede del rol musico principal.

Si se aplican estas soluciones tendremos como ventaja que estos usuarios podrán acceder de forma concurrente a la base de datos, pero como inconveniente se producirá una gran sobrecarga del servidor si todos ellos acceden a la vez.

CUESTIÓN 1.5: CREAR UN USUARIO CONCRETO DE CADA ROL DE USUARIO DE LA BASE DE DATOS

Ejecutando las siguientes consultas SQL para crear los usuarios:

```
CREATE USER admin WITH LOGIN NOSUPERUSER INHERIT NOCREATEDB NOREPLICATION;
GRANT "Administrador" TO admin WITH ADMIN OPTION;

CREATE USER musico1 WITH LOGIN NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT NOREPLICATION CONNECTION LIMIT -1;
GRANT "Musico" TO musico1;

CREATE USER organizador1 WITH LOGIN NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT NOREPLICATION CONNECTION LIMIT -1;
GRANT "Organizador" TO organizador1;

CREATE USER productor1 WITH LOGIN NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT NOREPLICATION CONNECTION LIMIT -1;
GRANT "Productor" TO productor1;
```

Con esto ya se tiene un usuario dentro de cada rol/grupo. Estos usuarios tienen la posibilidad de hacer login y acceder a la base de datos.

CUESTIÓN 1.6: REALIZAR LAS PRUEBAS NECESARIAS PARA COMPROBAR QUE LA PLANIFICACIÓN DE LA SEGURIDAD FUNCIONA CORRECTAMENTE CON EL FICHERO DE LOG ARRANCADO. ¿QUÉ ES LO QUE SE RECOGE EN DICHO FICHERO?

Para comprobar que los permisos funcionan correctamente, vamos a intentar conectarnos desde otro usuario e intentar ejecutar consultas, tanto permitidas para este usuario, como no permitidas.

Por ejemplo, comprobamos que el servidor acepta conexión si especificamos “musico1” como usuario y nos permite acceder a la base de datos. Si intentamos acceder a los datos de la tabla entradas, lo cual en teoría estaba prohibido para este usuario, nos devuelve lo siguiente:

```
ERROR:  permission denied for relation Entradas
SQL state: 42501
```

Si intentamos acceder a una tabla a la que supuestamente si tenemos permisos, este error no sucede y nos devuelve el contenido de las tablas, que en este caso están vacías. Esta sería la respuesta del servidor al intentar acceder a la tabla canciones desde este usuario:

Codigo_cancion [PK] integer	Nombre text	Compositor text	Fecha_grabacion date	Duracion time without time zone	Codigo_disco_Discos integer

Si modificamos el archivo de configuración de postgresql para que registre en el log las conexiones, en el log se verá reflejado las conexiones que realizan los usuarios y desde que dirección se realizan, así como las consultas que realizan estos usuarios sobre la base de datos.

Esto sería lo que el log registra al acceder a la base de datos con el usuario especificado anteriormente:

```
2018-12-29 15:40:07 CET [5648]: [1-1]
user=[unknown],db=[unknown],app=[unknown],client=127.0.0.1 LOG:  connection
received: host=127.0.0.1 port=51664
2018-12-29 15:40:07 CET [5648]: [2-1]
user=musico1,db=Musicos,app=[unknown],client=127.0.0.1 LOG:  connection
authorized: user=musico1 database=Musicos
```

Y esto sería lo que registra el log al intentar acceder a una tabla para la cual no tenemos permiso:

```
2018-12-29 15:37:46 CET [9412]: [9-1] user=musico1,db=Musicos,app=pgAdmin 4 -
CONN:1415218,client=127.0.0.1 LOG:  statement: SELECT * FROM public."Entradas"

2018-12-29 15:37:46 CET [9412]: [10-1] user=musico1,db=Musicos,app=pgAdmin 4 -
CONN:1415218,client=127.0.0.1 ERROR:  permission denied for relation Entradas
2018-12-29 15:37:46 CET [9412]: [11-1] user=musico1,db=Musicos,app=pgAdmin 4 -
CONN:1415218,client=127.0.0.1 STATEMENT:  SELECT * FROM public."Entradas"
```

CUESTIÓN 1.7: SUPONER QUE UN USUARIO EN CONCRETO DE LA BASE DE DATOS ENTRA EN LA MISMA, SÓLO DEBERÍA DE PODER CONSULTAR, MODIFICAR, BORRAR Y ACTUALIZAR SUS DATOS Y NO LOS DE LOS OTROS USUARIOS, DEPENDIENDO DE LA SEGURIDAD IMPLEMENTADA ANTERIORMENTE. ¿CÓMO SE PODRÍA IMPLEMENTAR ESA SEGURIDAD? IMPLEMENTAR ESE TIPO DE SEGURIDAD PARA UN USUARIO DE CADA ROL DEFINIDO ANTERIORMENTE.

Para cubrir esta serie de problemas PostgreSQL nos proporciona una serie de políticas llamadas RLS (Row Level Security). Esta serie de políticas se encarga de a partir de una comprobación que se especifica en código SQL, que se tenga que cumplir una condición para ver una serie de tuplas dentro de una tabla.

Primero hay que activar que estas políticas se apliquen en cada tabla, esto se hace con la sentencia SQL:

```
ALTER TABLE "Musicos" ENABLE ROW LEVEL SECURITY;
```

Esta sentencia se tiene que ejecutar para todas las tablas en las que queramos poner esta capa de protección de información.

En aquellas tablas donde se active esta opción, habrá que definir las políticas de acceso que se tienen que cumplir para todos y cada uno de los roles de usuario de la base de datos.

Se definen las siguientes políticas sobre la tabla “Musicos”:

```
CREATE POLICY admin_todo ON "Musicos" TO “Administrador” USING (true) WITH CHECK (true);
CREATE POLICY productor_todo ON "Musicos" TO "Productor" USING (true) WITH CHECK (true);
CREATE POLICY organizador_lectura ON "Musicos" FOR SELECT TO "Organizador" USING(true);
CREATE POLICY musicos_propio ON "Musicos" TO "Musico" USING(current_user = "Nombre");
```

Para probar esta última política, se ha creado un nuevo usuario que pertenece al rol “Musico” con nombre “nombre1”.

Puesto que no saltan errores cuando estas políticas no se cumplen (en el caso de los select), si no que se limita a mostrar las tablas vacías, se ha utilizado el script generador de datos creado para la PECL2 con números reducidos para poblar las tablas de forma que se pueda comprobar que estas políticas funcionan.

Si hacemos un Select * de la tabla Musicos desde la cuenta de administrador, nos devuelve esto.

	codigo_musico integer	DNI character (10)	Nombre text	Direccion text	Codigo_Postal integer	Ciudad text	Provincia text	telefono integer	Instrumentos text	Codigo_grupo_Grupo integer
1	0	00000001R	nombre0	direccion0	35817	ciudad0	provincia0	913847528	Bongos	0
2	1	00000002W	nombre1	direccion1	80895	ciudad1	provincia1	959688384	Bongos	10
3	2	00000003A	nombre2	direccion2	89525	ciudad2	provincia2	952020149	Flauta	11
4	3	00000004G	nombre3	direccion3	15054	ciudad3	provincia3	917710131	Piano	7
5	4	00000005M	nombre4	direccion4	28398	ciudad4	provincia4	913978768	Trompeta	19
6	5	00000006Y	nombre5	direccion5	39023	ciudad5	provincia5	993115835	Bateria	12
7	6	00000007F	nombre6	direccion6	20709	ciudad6	provincia6	986850012	Guitarra	3
8	7	00000008P	nombre7	direccion7	40257	ciudad7	provincia7	974292071	Trompeta	14
9	8	00000009D	nombre8	direccion8	32479	ciudad8	provincia8	964670751	Triangulo	13
10	9	00000010X	nombre9	direccion9	50074	ciudad9	provincia9	983160426	Trombon	3
11	10	00000011B	nombre10	direccion10	72116	ciudad10	provincia10	937308059	Cantante	11
12	11	00000012N	nombre11	direccion11	29228	ciudad11	provincia11	948476844	Laud medieval	6
13	12	00000013J	nombre12	direccion12	13169	ciudad12	provincia12	950322256	Trompeta	4

Sin embargo, si lo hacemos desde la nueva cuenta “nombre1”, se nos devolverá lo siguiente:

	codigo_musico integer	DNI character (10)	Nombre text	Direccion text	Codigo_Postal integer	Ciudad text	Provincia text	telefono integer	Instrumentos text	Codigo_grupo_Grupo integer
1	1	00000002W	nombre1	direccion1	80895	ciudad1	provincia1	959688384	Bongos	10

Las políticas también funcionan para los updates/deletes.

CUESTIÓN 1.8: QUÉ MÉTODOS DE AUTENTIFICACIÓN DE USUARIOS TIENE DISPONIBLE POSTGRESQL? CONFIGURAR EL SERVIDOR POSTGRES PARA QUE UN USUARIO CONECTADO DESDE EL ORDENADOR DE UNO DE LOS MIEMBROS DEL GRUPO PUEDA ACCEDER A LA BASE DE DATOS **MUSICOS** DEL ORDENADOR DEL OTRO COMPAÑERO. ESPECIFICAR TODOS LOS PASOS SEGUIDOS Y LA CONFIGURACIÓN REALIZADA, MOSTRANDO EVIDENCIAS DE QUE SE HA PODIDO REALIZAR LA OPERACIÓN.

PostgreSQL permite los siguientes métodos de autentificación:

- Trust. Este método permite conectar sin ningún tipo de autentificación, es el usado por defecto para conexiones desde la máquina local.
- Password. Se necesita una contraseña para acceder a la base de datos. Según se tenga configurado, esta contraseña puede ir en texto plano o usando el algoritmo de encriptación MD5.
- GSSAPI
- SSPI
- Kerberos
- Ident. Usa el nombre de la cuenta del sistema operativo para intentar logear en la base de datos.
- Peer. Similar al anterior pero solo soporta conexiones locales.
- LDAP. Similar al método de contraseña.
- Radius. Similar al método de contraseña.
- Certificate. Intenta usar certificados de SSL para autentificarse en el servidor de base de datos.
- PAM. Similar al método de contraseña.

Para activar el acceso remoto al servidor de PostgreSQL, primero se tiene que entrar al archivo de configuración de postgre. En el tenemos que asegurarnos de que las siguientes variables están en funcionamiento:

```
listen_addresses = '*'
port = 5432
max_connections = 100
```

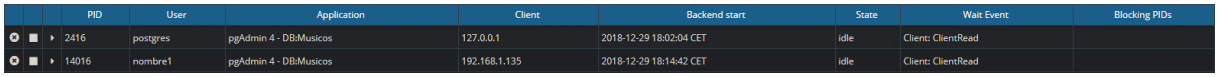
Después, tenemos que ir al archivo de configuración pg_hba.conf. En este archivo tenemos que asegurarnos de que las conexiones están abiertas y se puede configurar el método de autentificación de los usuarios.

```
host    all             all             0.0.0.0/0               md5
```

Esta línea indica que es posible la conexión desde cualquier IP y que se usa md5 para la encriptación de contraseñas.

Es importante entonces que los usuarios con los que queramos acceder en remoto tengan una contraseña establecida, porque si no dará error al intentar logear sin contraseña. Para hacer pruebas accederemos con la cuenta “nombre1”, a la cual le añadiremos como contraseña su propio nombre de usuario.

Desde el segundo ordenador, lanzamos pgAdmin y abrimos una nueva conexión a servidor. Se introduce la IP, el puerto, el usuario y la contraseña. Conseguimos entrar y acceder a la base de datos. Desde el pgAdmin del ordenador que hace de servidor podemos ver lo siguiente:



	PID	User	Application	Client	Backend start	State	Wait Event	Blocking PIDs
	2416	postgres	pgAdmin 4 - DB:Musicos	127.0.0.1	2018-12-29 18:02:04 CET	idle	Client: ClientRead	
	14016	nombre1	pgAdmin 4 - DB:Musicos	192.168.1.135	2018-12-29 18:14:42 CET	idle	Client: ClientRead	

A parte de la sesión en local, tenemos una segunda conexión al servidor desde una IP remota dentro de la red local.

En el log se ve reflejado de la siguiente manera:

```
2018-12-29 18:14:26 CET [10708]: [1-1]
user=[unknown],db=[unknown],app=[unknown],client=192.168.1.135 LOG:  connection
received: host=192.168.1.135 port=58282
2018-12-29 18:14:26 CET [10708]: [2-1]
user=nombre1,db=Musicos,app=[unknown],client=192.168.1.135 LOG:  connection
authorized: user=nombre1 database=Musicos
```

ACTIVIDADES Y CUESTIONES PARTE 2

En esta parte la base de datos **MUSICOS** deberá de ser nueva y no contener datos. Además, consta de 4 actividades:

- Conceptos generales.
- Manejo de transacciones.
- Concurrencia.
- Registro histórico.

CUESTIÓN 2.1: 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 para PostgreSQL se llama WAL (Write-ahead logging) y se encuentra activo por defecto.

Para PostgreSQL 10, los archivos de log generados por el WAL se almacenan en la ruta:

\PostgreSQL\data\pg10\pg_wal

Por defecto, los archivos de log son de 16MB cada uno. Esta es la apariencia del log al abrirlo con un editor de texto:

Este archivo de log es referente a la carga de datos que se hizo para la primera parte de esta práctica. Viendo las partes que se pueden leer como texto se puede deducir que el WAL está recogiendo la inserción de esas tuplas en este caso por si acaso se produjese un error durante su inserción.

CUESTIÓN 2.2: ¿QUÉ PARÁMETROS SE PUEDEN CONFIGURAR DEL DIARIO DEL SISTEMA DE POSTGRES? ¿PARA QUÉ SIRVEN CADA UNO DE ELLOS?

Los parámetros configurables para el WAL son los siguientes:

- Parámetros relacionados con los checkpoints. Van a determinar con que frecuencia se van a realizar los checkpoints, es decir, las escrituras a disco de los cambios registrados en el WAL.
- Parámetros de archivo. Si se activan, permiten que los archivos de log generados por el WAL se archiven en una carpeta aparte.
- Opciones de configuración del WAL. Son parámetros que van a dictaminar desde el tipo de operaciones que queremos que sean registradas en el WAL, como la forma en que se va a realizar la sincronización entre transacciones. También permite modificar la cantidad de información que se va a almacenar en cada uno de los segmentos del WAL, etc.

CUESTIÓN 2.3: REALIZAR UNA OPERACIÓN DE INSERCIÓN DE UN GRUPO MUSICAL SOBRE LA BASE DE DATOS **MUSICOS**. ABRIR EL ARCHIVO DE DIARIO ¿SE ENCUENTRA REFLEJADA LA OPERACIÓN EN EL ARCHIVO DEL SISTEMA? ¿EN CASO AFIRMATIVO, POR QUÉ LO HARÁ?

Se ejecuta la siguiente consulta SQL.

```
INSERT INTO "Grupo" VALUES(123456789,'grupo123456789','blues','albania','www.vivaalbania.com');
```

La operación se registra en el último archivo del WAL. Se pueden ver las tuplas que son de tipo text.

Se registrará la operación en este log por si acaso se produce una caída antes del checkpoint y necesita rehacer la operación.

CUESTIÓN 2.4: ¿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.

Este ejecutable proporcionado por PostgreSQL se encarga de crear una versión de un archivo WAL que sea legible.

Si se llama con el parámetro -z, nos permite ver estadísticas relacionadas con el fichero WAL especificado.

```
C:\PostgreSQL\pg10\bin>pg_waldump.exe -p C:\PostgreSQL\data\pg10\pg_wal -z 000000010000000000000001
Type                                           N      (%)          Record size      (%)             FPI size      (%)        Combined size      (%)
----                                           -      ---          -----------      ---             --------      ---        -------------      ---
XLOG                                         275 (  0.64)               14045 (  0.29)              485188 ( 33.88)              499233 (  7.98)
Transaction                                  573 (  1.32)              157004 (  3.25)                   0 (  0.00)              157004 (  2.51)
Storage                                       48 (  0.11)                2016 (  0.04)                   0 (  0.00)                2016 (  0.03)
CLOG                                           1 (  0.00)                  30 (  0.00)                   0 (  0.00)                  30 (  0.00)
Database                                       3 (  0.01)                 126 (  0.00)                   0 (  0.00)                 126 (  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                                      276 (  0.64)               16268 (  0.34)                   0 (  0.00)               16268 (  0.26)
Heap2                                        766 (  1.77)              265989 (  5.51)              376832 ( 26.31)              642821 ( 10.27)
Heap                                       15766 ( 36.42)             2157334 ( 44.69)              224736 ( 15.69)             2382070 ( 38.05)
Btree                                      25585 ( 59.10)             2214825 ( 45.88)              345496 ( 24.12)             2560321 ( 40.90)
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                                      43293                      4827637 [77.12%]             1432252 [22.88%]             6259889 [100%]
```

Esto es lo que devuelve para nuestro archivo.

CUESTIÓN 2.5: 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.

Analizando cronológicamente la salida de la llamada a pg_walldump.exe y comparando con los oids que devuelve para saber que tablas se están tocando, se ha conseguido saber que el xid o id de transacción para el insert anterior es 575 en nuestro caso.

Si lo llamamos sin el parámetro -z, el programa nos devuelve esto:

```
C:\PostgreSQL\pg10\bin>pg_waldump.exe -p C:\PostgreSQL\data\pg10\pg_wal -x 575 000000010000000000000001
rmgr: Heap        len (rec/tot):    108/   108, tx:        575, lsn: 0/01606218, prev 0/016061E0, desc: INSERT+INIT off 1, blkref #0: rel 1663/16384/16395 blk 0
rmgr: Btree       len (rec/tot):     78/    78, tx:        575, lsn: 0/01606288, prev 0/01606218, desc: NEWROOT lev 0, blkref #0: rel 1663/16384/16401 blk 1, blkref #2: rel 1663/16384/16401 blk 0
rmgr: Btree       len (rec/tot):     64/    64, tx:        575, lsn: 0/016062D8, prev 0/01606288, desc: INSERT_LEAF off 1, blkref #0: rel 1663/16384/16401 blk 1
rmgr: Transaction len (rec/tot):     34/    34, tx:        575, lsn: 0/01606318, prev 0/016062D8, desc: COMMIT 2018-12-30 14:34:07.126997 Romance Standard Time
```

Usando el parámetro -z para obtener estadísticas, obtenemos lo siguiente:

```
C:\PostgreSQL\pg10\bin>pg_waldump.exe -p C:\PostgreSQL\data\pg10\pg_wal -x 575 -z 000000010000000000000001
Type                                           N      (%)          Record size      (%)             FPI size      (%)        Combined size      (%)
----                                           -      ---          -----------      ---             --------      ---        -------------      ---
XLOG                                           0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
Transaction                                    1 ( 25.00)                  34 ( 11.97)                    0 (  0.00)                  34 ( 11.97)
Storage                                        0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
CLOG                                           0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
Database                                       0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
Tablespace                                     0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
MultiXact                                      0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
RelMap                                         0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
Standby                                        0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
Heap2                                          0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
Heap                                           1 ( 25.00)                 108 ( 38.03)                    0 (  0.00)                 108 ( 38.03)
Btree                                          2 ( 50.00)                 142 ( 50.00)                    0 (  0.00)                 142 ( 50.00)
Hash                                           0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
Gin                                            0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
Gist                                           0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
Sequence                                       0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
SPGist                                         0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
BRIN                                           0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
CommitTs                                       0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
ReplicationOrigin                              0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
Generic                                        0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
LogicalMessage                                 0 (  0.00)                   0 (  0.00)                    0 (  0.00)                   0 (  0.00)
                                         --------                     --------                      --------                     --------
Total                                          4                          284 [100.00%]                   0 [0.00%]                  284 [100%]
```

Podemos ver que se ha generado 1 registro de tamaño 34 unidades.

CUESTIÓN 2.6: SE VA A CREAR UN BACKUP (INCLUSO SI SE HABÍA REALIZADO YA) DE LA BASE DE DATOS **MUSICOS**. ESTE BACKUP SERÁ UTILIZADO MÁS ADELANTE. REALIZAR SOLAMENTE EL BACKUP MEDIANTE EL PROCEDIMIENTO DESCRITO EN EL APARTADO 25.3 DEL MANUAL (DEPENDIENDO DE LA VERSION, 10.4 *"CONTINOUS ARCHIVING AND POINT-IN-TIME RECOVERY (PITR)"*.

Para permitir acceso de tipo replicación a la base de datos, hay que añadir las siguientes líneas al archivo de configuración pg_hba.conf:

```
host    replication     postgres        ::1/32                  trust
hostnossl    replication     postgres    ::1/32                  trust
```

Se activa la opción en el archivo de configuración para que se pueden archivar ficheros del WAL. A continuación, se hace la siguiente llamada al programa pg_basebackup:

```
C:\PostgreSQL\pg10\bin>pg_basebackup.exe -D backup -Ft -z -P -U postgres
30140/30140 kB (100%), 1/1 tablespace
```

Se ha creado una carpeta que contiene los siguientes archivos:

CUESTIÓN 2.7: 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?

Para controlar el acceso concurrente a la base de datos a través de transacciones, PostgreSQL establece bloqueos(locks) a diferentes niveles. Puede establecer bloqueos a nivel de tabla, columna y página/bloque con el que se esté trabajando.

Como no todos los accesos necesitan necesariamente bloquear la información, PostgreSQL se va a encargar de administrar estos tipos de bloqueos según el nivel de aislamiento de la transacción.

Para ver los bloqueos activos en la base de datos, y los procesos que están esperando a que estos bloqueos de liberen, PostgreSQL proporciona la vista pg_locks, que nos muestra toda esta información.

A parte, para ver que transacciones se encuentran activas en un determinado momento, PostgreSQL proporciona una serie de funciones de información del sistema que nos proporciona la ID de una transacción en concreto, o de todas las transacciones que se estén ejecutando en un determinado momento. Son las funciones de la familia txid.

Para ver la información no hay más que llamar a las funciones de txid o acceder a los datos que se necesiten de la vista pg_locks.

CUESTIÓN 2.8: CREAR UNA TRANSACCIÓN QUE INSERTE UN GRUPO MUSICAL NUEVO EN LA BASE DE DATOS (NO CIERRE LA TRANSACCIÓN). REALIZAR UNA CONSULTA SQL PARA MOSTRAR TODOS LOS GRUPOS MUSICALES DE LA BASE DE DATOS DENTRO DE ESA TRANSACCIÓN. CONSULTAR LA INFORMACIÓN ANTERIOR SOBRE LO QUE SE ENCUENTRA ACTUALMENTE ACTIVO EN EL SISTEMA. ¿QUÉ CONCLUSIONES SE PUEDEN EXTRAER?

Se ejecutan las siguientes consultas:

```
BEGIN;
INSERT INTO "Grupo" VALUES(2,'grupo2','jazz','france','www.nosgustaeljazz.com');
SELECT * FROM "Grupo";
```

A lo que la base de datos responde con:

	Codigo_grupo integer	Nombre text	Genero_musical text	Pais text	Sitio_web text
1	123456789	grupo123...	blues	albania	www.vivaalbania.com
2	2	grupo2	jazz	france	www.nosgustaeljazz.com

Llamando a la función txid_current_snapshot() desde otra consola, nos debería devolver las transacciones que están activas en este momento. La respuesta nos dice que hay una transacción activa con txid 577. Si ahora llamamos para comprobar el txid desde la consola original, nos debería devolver que su txid es 577. La respuesta es la siguiente:

	txid_current_if_assigned bigint
1	577

Lo cual es congruente.

Como conclusiones se podría sacar que, aunque ese valor no se haya volcado a memoria global de la base de datos aún, desde la propia transacción se puede acceder al valor como si ya se hubiese hecho un commit del insert. A parte se puede ver que las funciones de control de transacciones activas funcionan correctamente.

Se acaba el bloque de transacción con un commit.

CUESTIÓN 2.9: UTILIZANDO PGADMIN O PSQL, COMENZAR UNA TRANSACCIÓN T1 EN UN USUARIO QUE REALICE LAS SIGUIENTES OPERACIONES SOBRE LA BASE DE DATOS **MUSICOS**. NO TERMINE LA TRANSACCIÓN. SIMPLEMENTE:

- Inserte un grupo musical nuevo.
- Inserte un músico nuevo asociado al grupo anterior.
- Inserte un disco nuevo del grupo anterior.

Se empieza el bloque de transacción con las siguientes consultas:

```
BEGIN;
INSERT INTO "Grupo" VALUES(3,'grupo3','rock','portugal','www.grupo3_portugal.com');
INSERT INTO "Musicos" VALUES(1,'03212337L','Edu','blabla',28564,'Madrid','Madrid',123456,'flauta travesera colombiana',3);
INSERT INTO "Discos" VALUES(1,'melhor album do ano','14/02/2018','rock','mp3',3);
```

CUESTIÓN 2.10: REALIZAR CUALQUIER CONSULTA SQL QUE MUESTRE LOS DATOS DEL GRUPO, MÚSICO Y DISCO QUE SE HAN INSERTADO, PARA VER QUE TODO ESTÁ CORRECTO.

Desde la misma consola, se hace un select para comprobar que los valores han sido introducidos satisfactoriamente. La siguiente captura es de la última inserción:

	Codigo_disco integer	Titulo text	Fecha_edicion date	Genero text	Formato text	Codigo_grupo_Grupo integer
1	1	melhor album do ano	2018-02-14	rock	mp3	3

CUESTIÓN 2.11: ESTABLECER UNA **NUEVA CONEXIÓN** CON PGADMIN O PSQL A LA BASE DE DATOS CON OTRO USUARIO DIFERENTE (ABRIR OTRA SESIÓN DIFERENTE A LA ABIERTA ACTUALMENTE QUE PERTENEZCA A OTRO USUARIO) 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?

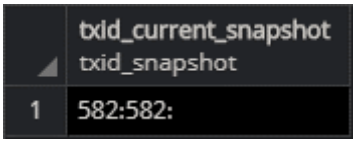
Se ejecutan las siguientes consultas en una ventana nueva:

```
SET ROLE admin;
SELECT * FROM “Discos”
```

A lo que el servidor responde mostrándonos la tabla “Discos” vacía.

Esta diferencia se debe a que la primera transacción aun no ha hecho un COMMIT, por lo que los datos siguen estando en la memoria local asignada a esa transacción y no en la memoria global común a toda la base de datos.

Si llamamos a la función txid_current_snapshot(), el servidor nos devuelve lo siguiente:



	txid_current_snapshot txid_snapshot
1	582:582:

La transacción 582 es la transacción que iniciamos en la pregunta anterior, y es la encargada de insertar estos datos. Esta transacción aparece en esta lista porque aún no se ha hecho el COMMIT/ROLLBACK.

CUESTIÓN 2.12: ¿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.

El resultado que viene desde la misma consola desde la cual se tiene la transacción viene de la memoria local asignada para hacer esa transacción, por lo que el valor ya ha sido modificado para ella.

El resultado que viene de la segunda consola y con el segundo usuario coge los valores almacenados en la memoria global de PostgreSQL. Como la anterior transacción aun no ha hecho el COMMIT, los valores que ha modificado aun no se encuentran en la memoria global, y es por esto que el resto de las sesiones que se abran no podrán ver estos valores modificados.

CUESTIÓN 2.13: 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?

Se hace el COMMIT para la transacción abierta.

Volvemos a hacer el select sobre la tabla Discos y esta vez ambas respuestas son iguales, mostrando ambas los valores insertados correctamente en las tablas.

Esto es debido a que al haberse hecho el COMMIT, los cambios han sido volcados a la memoria global de PostgreSQL, por lo que ya son visibles para el resto de usuarios/sesiones.

CUESTIÓN 2.14: REPETIR LA CUESTIÓN 2.9 CON OTRO GRUPO, MÚSICO Y DISCO. REALIZAR LA MISMA CONSULTA DE LA CUESTIÓN 2.10, PERO AHORA TERMINAR LA TRANSACCIÓN CON UN ROLLBACK Y REPETIR LA CONSULTA CON LOS MISMOS DOS USUARIOS. ¿CUÁL ES EL RESULTADO? ¿POR QUÉ?

Se hace la inserción desde la primera consola:

```
BEGIN;
INSERT INTO "Grupo" VALUES(5,'grupo5','pop latino','colombia','www.colooooombia.com');
```

Si hacemos un select desde esa misma consola, podemos ver los valores introducidos en las tablas:

	Codigo_grupo integer	Nombre text	Genero_musical text	Pais text	Sitio_web text
1	123456789	grupo123...	blues	alba...	www.vivaal...
2	2	grupo2	jazz	france	www.nosgu...
3	3	grupo3	rock	port...	www.grupo...
4	5	grupo5	pop latino	colo...	www.coloo...

Sin embargo, desde el resto de las sesiones no se pueden ver ya que no están en memoria global aún.

Después de hacer el ROLLBACK con la primera consola, al haberse cancelado la inserción, no se pueden ver desde ninguna sesión/usuario, desde la misma primera consola, al hacer el select, nos devuelve esto:

	Codigo_grupo integer	Nombre text	Genero_musical text	Pais text	Sitio_web text
1	123456789	grupo123...	blues	alba...	www.vivaal...
2	2	grupo2	jazz	france	www.nosgu...
3	3	grupo3	rock	port...	www.grupo...

Esto se debe a que al haberse hecho el ROLLBACK, se han cancelado los cambios planeados para la base de datos, por lo que los cambios que estaban en memoria local para esa transacción no son volcados a memoria global.

CUESTIÓN 2.15: SIN NINGUNA TRANSACCIÓN EN CURSO, ABRIR UNA TRANSACCIÓN EN UN USUARIO Y REALIZAR LAS SIGUIENTES OPERACIONES:

- Insertar un músico nuevo con DNI 5643234
- Insertar otro músico con DNI 4578345.
- Modificar en el paciente anterior su DNI a 5643234
- Cerrar la transacción.

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

Se abre la transacción y se hacen las dos inserciones, el estado de la base de datos en este punto es el siguiente:

	codigo_musico integer	DNI character (10)	Nombre text	Direccion text	Codigo_Postal integer	Ciudad text	Provincia text	telefono integer	Instrumentos text	Codigo_grupo_Grupo integer
1	1	03212337L	Edu	blabla	28564	Madrid	Madrid	123456	flauta travesera ...	3
2	10	5643234	Ernesto	calle 123	28643	albacete	castilla-la ...	654746733	piano	2
3	11	4578345	Ricardo	calle 321	27543	valencia	comunidad...	678789096	guitarra electrica	3

Se procede a hacer el update:

```
ERROR:  duplicate key value violates unique constraint "Unique_DNI"
DETAIL:  Key ("DNI")=(5643234   ) already exists.
SQL state: 23505
```

Ahora cerramos la transacción con un COMMIT, y volvemos a hacer el select. Este es el estado final de la base de datos:

	codigo_musico integer	DNI character (10)	Nombre text	Direccion text	Codigo_Postal integer	Ciudad text	Provincia text	telefono integer	Instrumentos text	Codigo_grupo_Grupo integer
1	1	03212337L	Edu	blabla	28564	Madrid	Madrid	123456	flauta travesera ...	3

Esto se debe a que, al haberse producido un error dentro del bloque de transacciones, PostgreSQL fuerza a que se haga un ROLLBACk al terminar el bloque de transacción, por lo que ninguno de los cambios se verán reflejados en el estado final de la base de datos.

CUESTIÓN 2.16: CERRAR TODAS LAS SESIONES ANTERIORES. ABRIR UNA SESIÓN CON UN USUARIO U1 DE LA BASE DE DATOS **MUSICOS**. INSERTAR LA SIGUIENTE INFORMACIÓN EN LA BASE DE DATOS:

- Insertar un grupo musical con código de grupo 20110.
- Insertar un disco que pertenezca al grupo anterior y que tenga código de disco 13560.

Desde el usuario “postgres”, se ejecutan las siguientes inserciones:

```
INSERT INTO "Grupo" VALUES(20110,'grupo20110','flamenco','japan','www.nihongo-very-ole.jp');
INSERT INTO "Discos" VALUES(13560,'homenaje a el cigala','14/02/2016','flamenco','mp4',20110);
```

CUESTIÓN 2.17: ABRIR UNA SESIÓN CON UN USUARIO DIFERENTE U2 DEL ANTERIOR DE LA BASE DE DATOS **MUSICOS**. ABRIR UNA TRANSACCIÓN T2 EN ESTE USUARIO U2 Y REALIZAR UNA MODIFICACIÓN DEL GRUPO CON CÓDIGO 20110 PARA CAMBIAR EL NOMBRE A “LA GUARDIA”. ¿QUÉ ACTIVIDAD HAY REGISTRADA EN LA BASE DE DATOS? ¿CUÁL ES LA INFORMACIÓN GUARDADA EN LA BASE DE DATOS? ¿POR QUÉ?

Se utiliza el usuario “admin”. Se abre la transacción y se ejecutan las siguientes líneas:

```
BEGIN;
UPDATE "Grupo" SET "Nombre"='la guardia' WHERE "Codigo_grupo"=20110;
```

Accediendo a la vista pg_stat_activity, nos encontramos con esta información:

64819	2018-12-31 14:23:56.806165+01	2018-12-31 14:27:01.791679+01	2018-12-31 14:27:01.791679+01	2018-12-31 14:27:01.792644+01	Client	ClientRead	idle in...	589	[null]	-- U2 T2	client backend
64936	2018-12-31 14:24:16.090584+01	2018-12-31 14:27:18.480903+01	2018-12-31 14:27:18.480903+01	2018-12-31 14:27:18.480903+01	[null]	[null]	active	[null]	589	select ...	client backend

La segunda línea que se enseña corresponde a una tercera consola desde la cual hemos hecho esta consulta.

El valor 589 se corresponde con el ID de transacción que como podemos ver está en estado de espera. Esto corresponde con T2, la transacción en la que estamos haciendo el update.

En cuanto al estado de la base de datos, haciendo un select desde la consola abierta con U1, podemos ver que el valor no se ha modificado aún en memoria global:

4	20110	grupo20110	flamenco	japan	www.nihon...

Desde dentro de T2, el valor sí se ve modificado:

4	20110	la guardia	flamenco	japan	www.nihon...

Esto se debe a que aún no se ha hecho el COMMIT.

CUESTIÓN 2.18. ABRA UNA TRANSACCIÓN T1 EN EL USUARIO U1. HAGA UNA ACTUALIZACIÓN DEL DISCO CON CÓDIGO 13560 PARA CAMBIAR EL GÉNERO Y PONER 'ROCK'. ¿QUÉ ACTIVIDAD HAY REGISTRADA EN LA BASE DE DATOS? ¿CUÁL ES LA INFORMACIÓN GUARDADA EN LA BASE DE DATOS? ¿POR QUÉ?

U1 ejecuta:

```
BEGIN;
UPDATE "Discos" SET "Genero"='rock' WHERE "Codigo_disco"=13560;
```

Accediendo a la vista de pg_locks, podemos ver lo siguiente en cuanto a actividad:

9	transactionid			[null]	[null]	[null]	590		[null]	8/123	3412	ExclusiveLock	true	false
10	transactionid			[null]	[null]	[null]	589		[null]	5/32	14756	ExclusiveLock	true	false

Siendo 589 y 590 los ids de transacciones correspondientes a T2 y T1 respectivamente.

Desde una tercera consola se hace un select sobre la relación "Discos" para comprobar que los datos aún no se han modificado:

	Codigo_disco integer	Titulo text	Fecha_edicion date	Genero text	Formato text	Codigo_grupo_Grupo integer
1	1	melhor...	2018-02-14	rock	mp3	3
2	13560	homen...	2016-02-14	flamenco	mp4	20110

Lo cual es congruente, ya que aun no se ha hecho el COMMIT.

CUESTIÓN 2.19: EN LA TRANSACCIÓN T2, REALICE UNA MODIFICACIÓN DEL DISCO CON CÓDIGO 13560 PARA CAMBIAR LA FECHA DE EDICIÓN Y PONER LA ACTUAL. ¿QUÉ ACTIVIDAD HAY REGISTRADA EN LA BASE DE DATOS? ¿CUÁL ES LA INFORMACIÓN GUARDADA EN LA BASE DE DATOS? ¿POR QUÉ?

Desde la consola abierta por U2, se ejecutan la siguiente instruccion:

```
UPDATE "Discos" SET "Fecha_edicion"='31/12/2018' WHERE "Codigo_disco"=13560;
```

La transacción T2 ha quedado bloqueada, ya que T1 tenía un lock sobre la tabla "Discos". Los valores no se han modificado en memoria global ya que no se han podido si quiera acceder a ellos.

En cuanto a actividad, accediendo a la vista pg_stat_activity:

2018-12-31 14:23:56.806165+01	2018-12-31 14:27:01.791679+01	2018-12-31 14:32:39.330924+01	2018-12-31 14:32:39.330924+01	Lock	transactionid	active	589	589	-- U2 T2	client backend
2018-12-31 14:24:16.090584+01	2018-12-31 14:34:42.402704+01	2018-12-31 14:34:42.402704+01	2018-12-31 14:34:42.402704+01	[null]	[null]	active	[null]	589	select ...	client backend
2018-12-31 14:23:45.107781+01	2018-12-31 14:30:49.347569+01	2018-12-31 14:30:49.347569+01	2018-12-31 14:30:49.348539+01	Client	ClientRead	idle in...	590	[null]	-- U1 T1	client backend

La segunda línea corresponde a la tercera consola desde la cual se ha accedido a la vista.

Como podemos ver, la transacción 589 (T2), se encuentra en estado activo, ya que está intentando hacer el update pero aun no ha conseguido acceso a la tabla. La columna transactionid hace referencia a qué está esperando la transacción, lo cual indica que está esperando a otra transacción. La columna que tiene "Lock", nos muestra el tipo de espera, es decir, la razón por la que esta esperando en este caso, y esto se debe a que está bloqueada por T1.

CUESTIÓN 2.20: EN LA TRANSACCIÓN T1, REALICE UNA MODIFICACIÓN DEL GRUPO MUSICAL CON CÓDIGO 20110 PARA MODIFICAR EL SITIO WEB Y PONER WWW.LAGUARDIA.ES. ¿QUÉ ACTIVIDAD HAY REGISTRADA EN LA BASE DE DATOS? ¿CUÁL ES LA INFORMACIÓN GUARDADA EN LA BASE DE DATOS? ¿POR QUÉ?

Desde T1 se ejecuta la siguiente instrucción:

```
UPDATE "Grupo" SET "Sitio_web" = 'www.laguardia.es' WHERE "Codigo_grupo"=20110;
```

A lo que PostgreSQL nos contesta con el siguiente error:

```
ERROR:  deadlock detected
DETAIL:  Process 3412 waits for ShareLock on transaction 589; blocked by process 14756.
Process 14756 waits for ShareLock on transaction 590; blocked by process 3412.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,7) in relation "Grupo"
SQL state: 40P01
```

Se ha producido un interbloqueo y PostgreSQL ha decidido matar a T1. Si ahora vamos a T2, vemos que la transacción del update de la pregunta anterior se ha podido ejecutar con éxito.

En cuanto a actividad, volvemos a acceder a la vista pg_stat_activity:

64819	2018-12-31 14:23:56.806165+01	2018-12-31 14:27:01.791679+01	2018-12-31 14:32:39.330924+01	2018-12-31 14:41:35.950553+01	Client	ClientRead	idle in...	589	[null]	-- U2 T2
64936	2018-12-31 14:24:16.090584+01	2018-12-31 14:43:24.766663+01	2018-12-31 14:43:24.766663+01	2018-12-31 14:43:24.766663+01	[null]	[null]	active	[null]	589	select ...
64736	2018-12-31 14:23:45.107781+01	[null]	2018-12-31 14:41:34.949087+01	2018-12-31 14:41:35.949555+01	Client	ClientRead	idle in...	[null]	[null]	-- U1 T1

La segunda línea corresponde a la tercera terminal que accede a la vista.

Como podemos ver, ambas transacciones siguen estando registradas en esta vista, pero solo la transacción con id 589 (T2) sigue teniendo un identificador de transacción.

Aún no se han hecho los COMMIT, por lo que la información guardada en la base de datos sigue siendo la inicial.

CUESTIÓN 2.21: COMPROMETA AMBAS TRANSACCIONES T1 Y T2. ¿CUÁL ES EL VALOR FINAL DE LA INFORMACIÓN MODIFICADA EN LA BASE DE DATOS PARA GRUPOS MUSICALES Y DISCOS? ¿POR QUÉ?

Al hacer el COMMIT en T1, PostgreSQL ha forzado a hacer un ROLLBACK debido al error causado por el interbloqueo.

T2 ha contestado como que ha hecho el COMMIT satisfactoriamente.

La información final de la tabla “Grupo” es la siguiente:

	Codigo_grupo integer	Nombre text	Genero_musical text	Pais text	Sitio_web text
1	123456789	grupo123...	blues	alba...	www.vivaalbania.com
2	2	grupo2	jazz	france	www.nosgustaeljazz.com
3	3	grupo3	rock	port...	www.grupo3_portugal.com
4	20110	la guardia	flamenco	japan	www.nihongo-very-ole.jp

La información final de la tabla “Discos” es la siguiente:

	Codigo_disco integer	Titulo text	Fecha_edicion date	Genero text	Formato text	Codigo_grupo_Grupo integer
1	1	melhor al...	2018-02-14	rock	mp3	3
2	13560	homenaje...	2018-12-31	flamenco	mp4	20110

Esto nos indica que todos los cambios que se habían producido desde T1 se han obviado. Esto se debe a que PostgreSQL ha decidido matarla cuando ha detectado interbloqueo, haciendo un rollback a todos sus cambios.

Los cambios producidos por T2 se ven reflejados en la memoria común y se han llevado a cabo satisfactoriamente.

CUESTIÓN 2.22: CERRAR TODAS LAS SESIONES ANTERIORES. ABRIR UNA SESIÓN CON UN USUARIO DE LA BASE DE DATOS **MUSICOS**. INSERTAR EN LA TABLA GRUPO UN NUEVO GRUPO MUSICAL QUE TENGA UN CÓDIGO DE GRUPO DE 50800. ABRIR UNA TRANSACCIÓN T1 EN ESTE USUARIO Y REALIZAR UNA MODIFICACIÓN DEL GRUPO MUSICAL CON CÓDIGO 50800Y ACTUALIZAR EL GÉNERO MUSICAL A ‘POP’. NO CIERRE LA TRANSACCIÓN.

Se abre una primera sesión con el usuario “postgres”. Se ejecutan las siguientes consultas SQL:

```
INSERT INTO "Grupo" VALUES(50800,'grupo50800','rock','uganda','www.grupo50800.com');
BEGIN;
UPDATE "Grupo" SET "Genero_musical"='pop' WHERE "Codigo_grupo"=50800;
```

CUESTIÓN 2.23: ABRIR UNA SESIÓN CON UN USUARIO DIFERENTE DEL ANTERIOR DE LA BASE DE DATOS **MUSICOS**. ABRIR UNA TRANSACCIÓN T2 EN ESTE USUARIO Y REALIZAR UNA MODIFICACIÓN DEL GRUPO MUSICAL CON CÓDIGO 50800 Y CAMBIAR EL GÉNERO MUSICAL A ‘HEAVY METAL’. 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?

Se abre una segunda sesión con el usuario “admin”. Se ejecutan las siguientes consultas SQL:

```
BEGIN;
UPDATE "Grupo" SET "Genero_musical"='heavy metal' WHERE "Codigo_grupo"=50800;
```

La transacción queda bloqueada, ya que T1 tiene un lock sobre esta tabla.

Accediendo a la vista pg_stat_activity desde una tercera terminal, tenemos lo siguiente:

61678	2018-12-31 15:00:23.250461+01	2018-12-31 15:02:09.867698+01	2018-12-31 15:03:10.061795+01	2018-12-31 15:03:10.062791+01	Client	ClientRead	idle in...	592	[null]	UPDAT...
63251	2018-12-31 15:04:23.597715+01	2018-12-31 15:05:11.33508+01	2018-12-31 15:05:11.33508+01	2018-12-31 15:05:11.33508+01	Lock	transactionid	active	593	592	BEGIN;
64103	2018-12-31 15:06:24.748531+01	2018-12-31 15:06:35.778944+01	2018-12-31 15:06:35.778944+01	2018-12-31 15:06:35.778944+01	[null]	[null]	active	[null]	592	select ...

La tercera línea corresponde a la terminal que llama para acceder a la vista.

Como ya ocurría con el bloque de transacciones anterior, una de nuestras transacciones ha sido bloqueada ya que ha intentado acceder a una tabla que ya tenía un lock hecho por otra transacción. En este caso, la transacción con id 593 (T2) está esperando a que se suelte el bloqueo causado por la transacción 592 (T1).

CUESTIÓN 2.24: COMPROMETA LA TRANSACCIÓN T1, ¿QUÉ ES LO QUE OCURRE? ¿POR QUÉ? ¿CUÁL ES EL ESTADO FINAL DE LA INFORMACIÓN DEL GRUPO MUSICAL CON CÓDIGO 50800 PARA AMBOS USUARIOS? ¿POR QUÉ?

La transacción T1 hace el COMMIT satisfactoriamente, el estado final de esa tupla de la tabla "Grupo" para "postgres" es la siguiente:

5	50800	grupo508...	pop	ugan...	www.grupo...

Mientras que para la transacción T2, el cual ya ha podido ejecutar su sentencia ya que se ha levantado el lock, el estado final de esa tupla para el usuario "admin" es el siguiente:

5	50800	grupo508...	heavy metal	ugan...	www.grupo...

Esto se debe a que T2 esta leyendo de memoria local el dato, pero este valor no se encuentra en memoria global.

El valor que se encuentra en memoria global en este momento es el valor que ha sido cambiado por T1, es decir, "Genero_musical"='pop'.

CUESTIÓN 2.25: COMPROMETA LA TRANSACCIÓN T2, ¿QUÉ ES LO QUE OCURRE? ¿POR QUÉ? ¿CUÁL ES EL ESTADO FINAL DE LA INFORMACIÓN DEL GRUPO MUSICAL CON CÓDIGO 50800? ¿POR QUÉ?

El COMMIT se produce satisfactoriamente, ya que en este caso no se ha producido interbloqueo y se ha podido ejecutar todo el bloque de transacciones correctamente.

Haciendo el select desde otra cuenta diferente a la de "admin" para poder leer la memoria global y estar seguros, nos devuelve la siguiente tupla:

5	50800	grupo508...	heavy metal	ugan...	www.grupo...

El valor final es el valor que ha actualizado T2, ya que es la última transacción en comprometerse, lo cual tiene sentido.

CUESTIÓN 2.26: CERRAR TODAS LAS SESIONES ANTERIORES. ABRIR UNA SESIÓN CON UN USUARIO DE LA BASE DE DATOS **MUSICOS**. ABRIR UNA TRANSACCIÓN T1 EN ESTE USUARIO Y REALIZAR UNA MODIFICACIÓN DEL DISCO CON CÓDIGO 13560 PARA CAMBIAR SU CÓDIGO A 23560. ABRA OTRO USUARIO DIFERENTE DEL ANTERIOR Y REALICE UNA TRANSACCIÓN T2 QUE CAMBIE EL FORMATO DEL DISCO CON CÓDIGO 13560 A 'MP3'. NO CIERRE LA TRANSACCIÓN.

Se abre la sesión con el usuario "postgres" y se ejecutan las siguientes consultas:

```
BEGIN;
UPDATE "Discos" SET "Codigo_disco"=23560 WHERE "Codigo_disco"=13560;
```

Se abre otra sesión con el usuario “admin” y se ejecutan las siguientes consultas:

```
BEGIN;
UPDATE "Discos" SET "Formato"='mp3' WHERE "Codigo_disco"=13560;
```

La transacción T2 se queda en estado de espera por el bloqueo causado por la transacción T1.

CUESTIÓN 2.27: COMPROMETA LA TRANSACCIÓN T1, ¿QUÉ ES LO QUE OCURRE? ¿POR QUÉ? ¿CUÁL ES EL ESTADO DE LA INFORMACIÓN DEL DISCO CON CÓDIGO 13560 PARA AMBOS USUARIOS? ¿POR QUÉ?

Se compromete T1 y con esto se consigue que T2 consiga acceder a la tabla y ejecutar su consulta. Para el usuario postgres, el estado de la tabla en este momento es el siguiente:

	Codigo_disco integer	Titulo text	Fecha_edicion date	Genero text	Formato text	Codigo_grupo_Grupo integer
1	1	melhor...	2018-02-14	rock	mp3	3
2	23560	homen...	2018-12-31	flamenco	mp4	20110

Para el usuario admin, el estado es el siguiente:

	Codigo_disco integer	Titulo text	Fecha_edicion date	Genero text	Formato text	Codigo_grupo_Grupo integer
1	1	melhor...	2018-02-14	rock	mp3	3
2	23560	homen...	2018-12-31	flamenco	mp4	20110

Como se puede ver el resultado es el mismo. Esto se debe a la transacción T2 no se le permite leer la tabla hasta que T1 no haya hecho el commit, por lo que una vez lea la tabla, ya no habrá ninguna tupla cuyo código de disco sea 13560, por lo que no tendrá que actualizar nada.

CUESTIÓN 2.28: COMPROMETA LA TRANSACCIÓN T2, ¿QUÉ ES LO QUE OCURRE? ¿POR QUÉ? ¿CUÁL ES EL ESTADO FINAL DE LA INFORMACIÓN DEL DISCO CON CÓDIGO 13560 PARA AMBOS USUARIOS? ¿POR QUÉ?

El estado final de la información es el siguiente, teniendo en cuenta que ambas transacciones han tenido un commit satisfactorio.

	Codigo_disco integer	Titulo text	Fecha_edicion date	Genero text	Formato text	Codigo_grupo_Grupo integer
1	1	melhor...	2018-02-14	rock	mp3	3
2	23560	homen...	2018-12-31	flamenco	mp4	20110

Esto es, la transacción T2 no ha realizado ningún cambio. Esto se debe a lo explicado en el punto anterior, aunque se haga el commit, debido a que no hay ningún valor cuyo código de disco sea 13560, no ha modificado nada, aunque la transacción se haya comprometido satisfactoriamente.

CUESTIÓN 2.29: ¿QUÉ ES LO QUE OCURRE EN EL SISTEMA GESTOR DE BASE DE DATOS SI DENTRO DE UNA TRANSACCIÓN QUE CAMBIA EL NOMBRE DEL GRUPO MUSICAL CON CÓDIGO 50800 SE ABRE OTRA TRANSACCIÓN QUE CAMBIE EL PAÍS DE RESIDENCIA? ¿POR QUÉ?

Se cierran las sesiones actuales y se abre una sesión con el usuario “postgres”. Se ejecutan las siguientes líneas:

```
BEGIN;
UPDATE "Grupo" SET "Nombre"='nombre modificado' WHERE "Codigo_grupo"=50800;
BEGIN;
```

PostgreSQL nos contesta con lo siguiente:

```
WARNING:  there is already a transaction in progress
BEGIN

Query returned successfully in 41 msec.
```

En referencia al último BEGIN.

Seguimos con la consulta:

```
UPDATE "Grupo" SET "Pais"='china' WHERE "Codigo_grupo"=50800;
```

Si accedemos a la vista pg_stat_activity, vemos lo siguiente:

57514	2018-12-31 15:57:22.249755+01	2018-12-31 15:58:33.600641+01	2018-12-31 16:04:20.818162+01	2018-12-31 16:04:20.81913+01	Client	ClientRead	idle in...	596	[null]	UPDAT...	client backend
59311	2018-12-31 16:02:04.808487+01	2018-12-31 16:04:47.302424+01	2018-12-31 16:04:47.302424+01	2018-12-31 16:04:47.302424+01	[null]	[null]	active	[null]	596	select ...	client backend

La segunda línea corresponde a la consola que accede a la vista.

Como se puede ver tenemos una sola transacción en ejecución, la segunda transacción lanzada desde dentro de T1 no ha creado un proceso diferente.

Si hacemos el COMMIT, se sale de la transacción completamente, y el estado final para esa tupla es lo siguiente:

5	50800	nombre modificado	heavy metal	china	www.grupo...

A efectos prácticos la transacción anidada no se ha producido, si no que se ha tratado a los dos updates como si estuviesen dentro del mismo bloque de transacción a un mismo nivel.

Mirando en la documentación de PostgreSQL se ha llegado a la conclusión de que PostgreSQL no soporta transacciones anidadas como tal, pero si se pueden hacer “puntos de guardado” para que si se produce un error solo se haga el ROLLBACK hasta ese punto. Esto se implementa con los comandos “SAVEPOINT”, “RELEASE SAVEPOINT” y “ROLLBACK TO SAVEPOINT”, estas sentencias SQL se introducen dentro de un bloque de transacción convencional que empieza con BEGIN y acaba con COMMIT/ROLLBACK. Si se hace un ROLLBACK para el bloque de transacción, se hace el ROLLBACK a las transacciones anidadas también.

CUESTIÓN 2.30: SUPONER QUE SE PRODUCE UNA PÉRDIDA DEL CLUSTER DE DATOS Y SE PROCEDE A RESTAURAR LA INSTANCIA DE LA BASE DE DATOS DEL PUNTO 2.6. REALIZAR SOLAMENTE LA RESTAURACIÓN (RECOVERY) MEDIANTE EL PROCEDIMIENTO DESCRITO EN EL APARTADO 25.3 DEL MANUAL (DEPENDIENDO DE LA VERSION, 10.4) *"CONTINOUS ARCHIVING AND POINT-IN-TIME RECOVERY (PITR)*. ¿CUÁL ES EL ESTADO FINAL DE LA BASE DE DATOS? ¿POR QUÉ?

Se detiene el servidor de PostgreSQL. Se copia el directorio de data y se borra el directorio original. Se descomprime el archivo base de recuperación generado cuando se hizo el backup. Se descomprime el archivo referente al backup del WAL y se mete en la carpeta pg_wall. Se copia el resto de los archivos del WAL que hemos copiado en el segundo paso. Se crea el archivo recovery.conf en la carpeta data/pg10 y se escribe la siguiente línea:

```
restore_command = ‘copy "C:\\PostgreSQL\\data\\pgarchive\\pg10\\%f" "%p"’
```

Después de esto se reinicia el servidor de PostgreSQL desde la terminal de Windows. PostgreSQL entra en modo recovery y empieza a leer y a restaurar todo lo que pueda desde el WAL. PostgreSQL va listando los archivos que va copiando. Cuando acaba le devuelve el control a la terminal. Al entrar al log, podemos ver esto entre las últimas líneas.

LOG: archive recovery complete

LOG: database system is ready to accept connections

Después de un reinicio total del ordenador, el servidor PostgreSQL se inicia normalmente y permite acceder a él.

La base de datos se encuentra en el estado en el que estaba antes de empezar con el proceso de recovery. Esto se debe a que al haber conservado los archivos WAL, PostgreSQL es capaz de rehacer las operaciones que sucedieron después del punto del recovery, restaurando la base de datos a su punto más avanzado.

Al entrar en el directorio data de PostgreSQL podemos observar que se ha cambiado la extensión de los archivos necesarios para hacer el recovery para evitar que PostgreSQL vuelva a entrar en modo recovery cuando se inicie de nuevo.

CUESTIÓN 2.31: 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.

PostgreSQL tiene implementando un sistema de recuperación REDO, ya que no vuelca los datos a memoria global hasta que no se hace el COMMIT, por lo que al producirse un error o necesitarse hacer un recovery desde una copia de seguridad, solo tiene que preocuparse de rehacer las operaciones oportunas sin tener que deshacer nada.

En cuanto a la concurrencia, PostgreSQL permite el acceso concurrente a la base de datos e implementa un sistema de bloqueos a nivel de tabla, si se intenta modificar una tabla desde dos transacciones a la vez, la segunda transacción quedará bloqueada hasta que la primera haya terminado y hecho COMMIT/ROLLBACK. Las planificaciones no siempre son secuenciables, pero si recuperables ya que no se escribe a memoria común hasta que no se hace el commit. Es por esto mismo que PostgreSQL tampoco tiene rollback en cascada, ya que los posibles fallos de una transacción no afectan al resto.

BIBLIOGRAFÍA

- Capítulo 5: Data Definition (System Columns, Privileges, Row Security Policies).
- Capítulo 13: Concurrency Control.
- Capítulo 20: Client Authentication.
- Capítulo 25: Backup and Restore.
- Capítulo 28: Monitoring Database Activity.
- Capítulo 30: Reliability and the Write-Ahead log.
- Capítulo 40: The Rule System.