

**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:** Luis Alejandro Cabanillas Prudencio

**DNI:** 04236930P

**ALUMNO 2:**

**Nombre y Apellidos:** Álvaro de las Heras Fernández

**DNI:** 03146833L

**Fecha:** 11- enero- 2019

**Profesor Responsable:** Iván González

Mediante la entrega de este fichero los alumnos aseguran que cumplen con la normativa de autoría de trabajos de la Universidad de Alcalá, y declaran éste como un trabajo original y propio.

En caso de ser detectada copia, se puntuará **TODA** la 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	Encargado de gestionar la base de datos	Capaz de realizar cualquier consulta y modificar tablas.
Músico	Usuario registrado en el sistema que estará relacionado con algún grupo y a su vez con el resto de las tablas.	Podrá consultar datos referentes a su grupo, discos, canciones. No podrá modificar, eliminar o insertar datos.
Organizador de conciertos	Usuario de la base de datos que se encargará de los conciertos	Podrá crear, consultar y eliminar conciertos de la base de datos.

<b>Vendedor de entradas</b>	Usuario del sistema que se encargará de la venta de entradas. Podrá insertar entradas y consultar conciertos y entradas.
-----------------------------	--

**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
<b>Administrador</b>	Todas las tablas	Todas las acciones	Restringido a esta base de datos
<b>Musico</b>	Discos, conciertos, grupo, canciones y músicos	Select sobre todas las tablas disponibles. Update y delete en músicos. Insert, update y select sobre discos y canciones.	Restringido a las filas de él mismo en músicos y de su grupo o disco en el resto de las tablas.
<b>Organizador de conciertos</b>	Conciertos, entradas, grupo y grupos_tocan_conciertos	Select sobre todas las tablas disponibles. Insert sobre conciertos, entradas y grupos_tocan_conciertos. Update y delete sobre conciertos, entradas y grupos_tocan_conciertos.	Tiene control sobre toda la gestión organizativa de los conciertos.
<b>Vendedor de entradas</b>	Entradas, conciertos, grupo y grupos_tocan_conciertos	Select sobre las tablas disponibles. Insert, update y delete sobre entradas.	Las acciones que comprometen a la base de datos estarán restringidas a la tabla entradas.

**Cuestión 1.3:** Implementar la seguridad de la base de datos para cada uno de los roles de usuarios.

Para implementar la seguridad lo primero que vamos a hacer es crear cada uno de los roles que hemos definido antes, estos roles tendrán una serie de características generales para la base de datos. Las características que tendrán serán que no pueden ser super-usuarios, que no pueden conceder permisos, que no pueden crear nuevos roles y ninguna acción más relacionada con la gestión de la base de datos como replicación o creación de bases de datos.

Una vez creados, el siguiente paso es concederle los permisos sobre las tablas que tenemos. Para ello emplearemos GRANT. Con GRANT podemos dar privilegios sobre determinadas tablas o incluso bases de datos enteras, estos permisos pueden ser de insertar nuevos datos hasta de crear nuevos schemas o triggers. Otra opción que no hemos elegido sería implementar una seguridad a nivel de fila (Row-Level-Policy), en este caso tendríamos que haber hecho una expresión que devolviese un booleano. Teniendo en cuenta nuestros objetivos no la hemos usado dado que, a pesar de que permite enseñar filas que nos interesan y esconder el resto, el rendimiento se puede ver afectado a peor si hay múltiples usuarios conectados a la base de datos.

A continuación, se muestra cómo se han creado con CREATE ROLE y GRANT:

#### • ROL ADMINISTRADOR:

```
1 CREATE ROLE administrador WITH
2 NOLOGIN
3 NOSUPERUSER
4 NOCREATEDB
5 NOCREATEROLE
6 NOINHERIT
7 REPLICATION
8 CONNECTION LIMIT -1
9 PASSWORD 'xxxxxx';
```

```
1 GRANT ALL PRIVILEGES ON DATABASE musicos to administrador;
2 GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO admin;
```

GRANT

Query returned successfully in 67 msec.

#### • ROL MÚSICO:

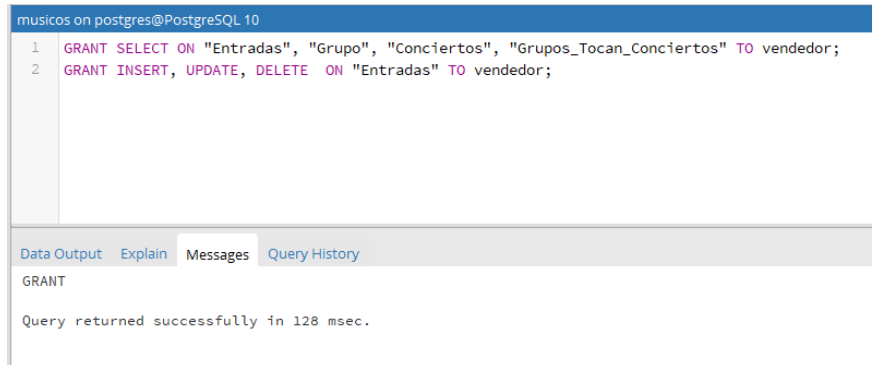
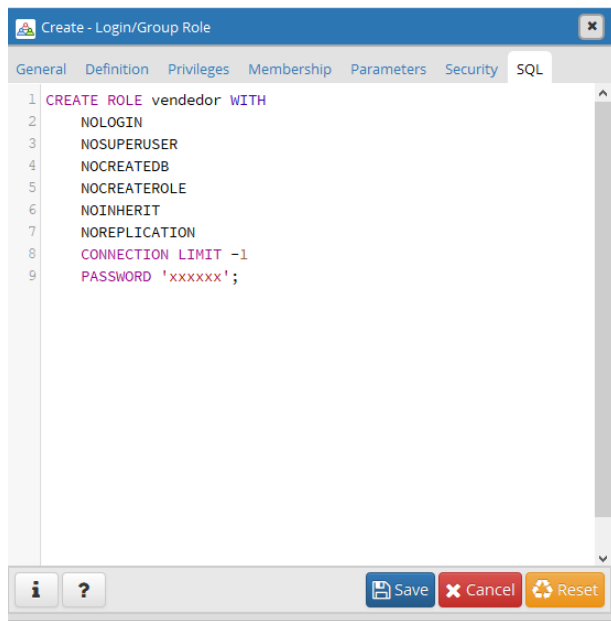
```
1 CREATE ROLE musico WITH
2 NOLOGIN
3 NOSUPERUSER
4 NOCREATEDB
5 NOCREATEROLE
6 NOINHERIT
7 NOREPLICATION
8 CONNECTION LIMIT -1
9 PASSWORD 'xxxxxx';
```

```
1 GRANT SELECT ON "Musicos", "Grupo", "Conciertos", "Discos", "Canciones" TO musico;
2 GRANT UPDATE, DELETE ON "Musicos" TO musico ;
3 GRANT INSERT, UPDATE ON "Discos", "Canciones" TO musico;
```

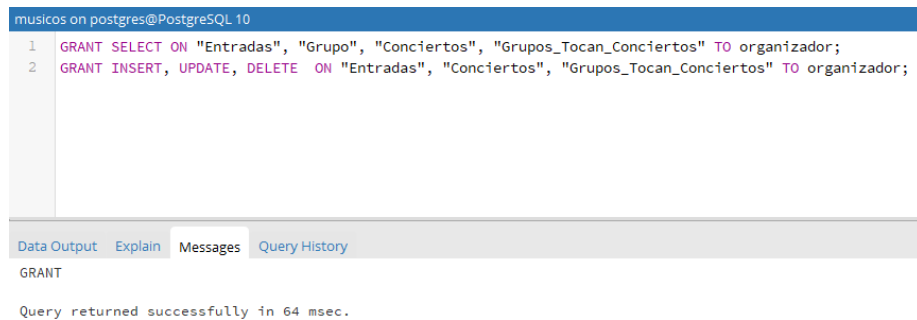
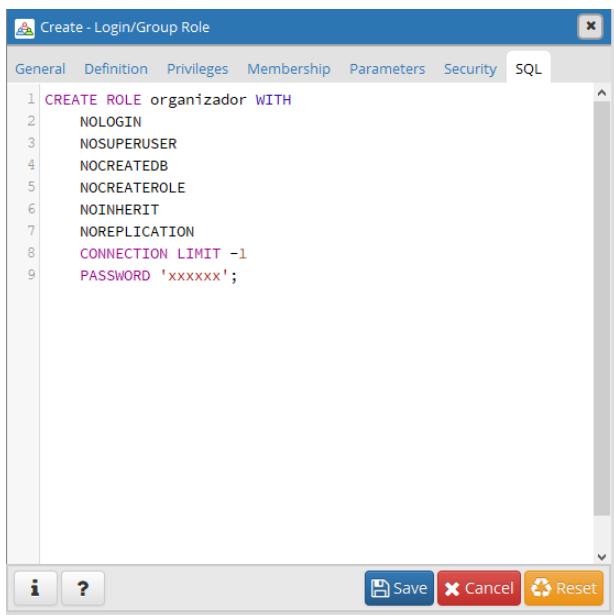
GRANT

Query returned successfully in 65 msec.

## • ROL VENDEDOR:



## • ROL ORGANIZADOR:



**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.

Lo primero de todo sería modificar el número de conexiones permitidas simultáneas de la base de datos por usuario, de esta forma se podría hacer que los usuarios fueran capaces de conectarse. Una vez hecho esto, lo siguiente sería establecer una política de acceso a las tablas. Esta política tendría que garantizar un uso eficiente de estas tablas para las distintas operaciones que puedan realizar. En este caso permitir concurrencia si fuesen lecturas y no se

modificaran datos o si fueran operaciones que alteraran la tabla se bloquease el acceso al resto de usuarios.

A simple vista esta concurrencia controlada puede traer ciertas ventajas como múltiples consultas de lectura a las tablas y una integridad de los datos al bloquear lecturas, básicamente lo que queremos es que nuestra base de datos tenga transacciones ACID de tal forma que los datos no se comprometan en las transacciones. Si enfocamos las ventajas a la seguridad que tenemos con los roles, estos usuarios podrán ser capaces de acceder a todos los datos que necesiten de las tablas, lo cual puede interpretarse como una ventaja para el usuario.

Las desventajas que pueden traer tienen que ver con la pérdida de rendimiento a cambio de ganar datos que son consistentes porque, al bloquear una tabla para escritura, esta no puede ser utilizada. Además en el peor de los casos puede producirse un Deadlock, es decir, que el sistema se quede en un bucle de espera producido por un bucle de tablas bloqueadas al tener una transacción que espera a otra transacción que estará esperando a la primera, esto tiene como solución matar a una transacción que se hará según se haya establecido. El tiempo por defecto de deadlock en PostgreSQL es de 1 segundo por lo que sí que afectaría bastante al sistema. Si al igual que antes nos centramos en la seguridad podemos ver que los datos de los usuarios pueden ser accedidos por todos, lo que hace que no sea muy seguro guardar datos, a no ser que se apliquen Row-Level-Policies.

### **Cuestión 1.5:** Crear un usuario concreto de cada rol de usuario de la base de datos

Para crear un usuario usaremos `CREATE USER` que nos permitirá asignarle un rol de los que teníamos creados con anterioridad al usuario. según PostgreSQL ahora funciona como un alias de `CREATE ROLE` que pone por defecto el valor de `LOGIN`, pero lo obviaremos. Al crear este usuario ya podremos hacer login en la base de datos. Aquí están los distintos usuarios:

```
musicos on postgres@PostgreSQL 10
1 CREATE USER guitarrista WITH IN ROLE musico

Data Output Explain Messages Query History
CREATE ROLE

Query returned successfully in 97 msec.
```

```
musicos on postgres@PostgreSQL 10
1 CREATE USER pedroelentradas WITH IN ROLE vendedor

Data Output Explain Messages Query History
CREATE ROLE

Query returned successfully in 151 msec.
```

```
musicos on postgres@PostgreSQL 10
1 CREATE USER juanorganizador WITH IN ROLE organizador

Data Output Explain Messages Query History
CREATE ROLE

Query returned successfully in 130 msec.
```

```
musicos on postgres@PostgreSQL 10
1 CREATE USER admin WITH IN ROLE administrador

Data Output Explain Messages Query History
CREATE ROLE

Query returned successfully in 99 msec.
```

**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 la planificación comenzamos comprobando que el log funcione y capture todas las consultas del sistema. En nuestro caso ya era así porque lo cambiamos en la práctica anterior. Las pruebas que realizaremos son las siguientes:

#### • PRUEBAS EN ADMINISTRADOR:

```
musicos on admin@PostgreSQL 10
1 SELECT "Codigo_concierto", "Fecha_realizacion", "Pais", "Ciudad", "Recinto"
2 FROM public."Conciertos";
3 DELETE FROM "Discos";
4 INSERT INTO public."Grupo"(
5     "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_web")
6     VALUES (1, 'Prueba','No' , 'Lituania','h.com');
7 CREATE TABLE public.tabla()WITH (OIDS = FALSE);
8 ALTER TABLE public.tabla OWNER to admin;
9
```

Data Output Explain Messages Query History

ALTER TABLE

Query returned successfully in 63 msec.

#### • PRUEBAS EN MUSICO:

```
musicos on guitarrista@PostgreSQL 10
1 SELECT "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_web"
2 FROM public."Grupo";
3 INSERT INTO public."Discos"(
4     "Codigo_disco", "Titulo", "Fecha_edicion", "Genero", "Formato", "Codigo_grupo_Grupo")
5     VALUES (1, 'Pepe', '1966-3-20', 'Blues', 'sds', 1);
6 UPDATE public."Discos"
7     SET "Titulo"='Adios'
8     WHERE "Codigo_disco"=1;
9 DELETE FROM "Musicos"
10 WHERE "codigo_musico"=1;
```

Data Output Explain Messages Query History

DELETE 0

Query returned successfully in 110 msec.

```
musicos on guitarrista@PostgreSQL 10
1 INSERT INTO public."Musicos"(
2     codigo_musico, "DNI", "Nombre", "Direccion", "Codigo_Postal", "Ciudad", "Provincia", telefono, "Instrumentos", "Codigo_grupo_Grupo")
3     VALUES (0,'0U','Pedro','Calle Azul',22319,'Santiago','Comunidad Valenciana',617850831,'Bateria',1);
```

Data Output Explain Messages Query History

ERROR: permiso denegado a la relación Musicos  
SQL state: 42501

## • PRUEBAS EN ORGANIZADOR:

musicos on juanorganizador@PostgreSQL 10

```

1 SELECT "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_web"
2   FROM public."Grupo";
3 INSERT INTO public."Conciertos"(
4     "Codigo_concierto", "Fecha_realizacion", "Pais", "Ciudad", "Recinto")
5   VALUES (0,'1939-11-6','Corea del norte','Fiji','Recinto ferial');
6 DELETE FROM "Entradas";
7 UPDATE public."Entradas"
8   SET "Localidad"='Madrid';
9 SELECT "Codigo_grupo_Grupo", "Codigo_concierto_Conciertos"
10  FROM public."Grupos_Tocan_Conciertos";

```

Data Output Explain Messages Query History

Codigo_grupo_Grupo	Codigo_concierto_Conciertos
integer	integer

musicos on juanorganizador@PostgreSQL 10

```

1 INSERT INTO public."Grupo"(
2   "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_web")
3   VALUES (2, 'Nombre', 'Musica', 'Islandia', 'n.com');

```

Data Output Explain Messages Query History

ERROR: permiso denegado a la relación Grupo  
SQL state: 42501

## • PRUEBAS EN VENDEDOR:

musicos on pedroelentradas@PostgreSQL 10

```

1 SELECT "Codigo_grupo_Grupo", "Codigo_concierto_Conciertos"
2   FROM public."Grupos_Tocan_Conciertos";
3 SELECT * FROM "Conciertos";
4 INSERT INTO public."Entradas"(
5   "Codigo_entrada", "Localidad", "Precio", "Usuario", "Codigo_concierto_Conciertos")
6   VALUES (1,'Pista central',71.37,'TronLives13',0);
7 UPDATE public."Entradas"
8   SET "Localidad"='Iraq'
9   WHERE "Codigo_entrada"=1;

```

Data Output Explain Messages Query History

UPDATE 1

Query returned successfully in 93 msec.

musicos on pedroelentradas@PostgreSQL 10

```

1 SELECT "Codigo_disco", "Titulo", "Fecha_edicion", "Genero", "Formato", "Codigo_grupo_Grupo"
2   FROM public."Discos";

```

Data Output Explain Messages Query History

ERROR: permiso denegado a la relación Discos  
SQL state: 42501



Una vez realizadas comprobamos el log para ver que contiene y analizarlo:

### • LOG ADMINISTRADOR:

```
2019-01-09 13:57:17.030 CET [5172] LOG:  sentencia: SELECT "Codigo_concierto", "Fecha_realizacion", "Pais", "Ciudad", "Recinto"
FROM public."Conciertos";
DELETE FROM "Discos";
INSERT INTO public."Grupo"(
  "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_web")
VALUES (1, 'Prueba','No' , 'Lituania','h.com');
CREATE TABLE public.tabla()WITH (oids = FALSE);
ALTER TABLE public.tabla OWNER to admin;

2019-01-09 13:57:17.035 CET [5172] LOG:  duraci n: 4.677 ms
```

```
2019-01-09 13:57:56.879 CET [5172] LOG:  sentencia: CREATE DATABASE prueba;
```

```
2019-01-09 13:57:56.881 CET [5172] ERROR:  se ha denegado el permiso para crear la base de datos
```

```
2019-01-09 13:57:56.881 CET [5172] SENTENCIA:  CREATE DATABASE prueba;
```

### • LOG MUSICO:

```
2019-01-09 14:55:24.011 CET [6176] LOG:  sentencia: SELECT "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_web"
FROM public."Grupo";
INSERT INTO public."Discos"(
  "Codigo_disco", "Titulo", "Fecha_edicion", "Genero", "Formato", "Codigo_grupo_grupo")
VALUES (1, 'Pepe', '1966-3-20', 'Blues', 'sds', 1);
UPDATE public."Discos"
SET "Titulo"='Adios'
WHERE "Codigo_disco"=1;
DELETE FROM "MUSICOS"
WHERE "codigo_musico"=1;

2019-01-09 14:55:24.021 CET [6176] LOG:  duraci n: 10.828 ms
```

```
2019-01-09 15:01:19.920 CET [5188] LOG:  sentencia: INSERT INTO public."MUSICOS"(
  codigo_musico, "DNI", "Nombre", "Direccion", "Codigo_Postal", "Ciudad", "Provincia", telefono, "Instrumentos", "Codigo_grupo_grupo")
VALUES (0,'00','Pedro','Calle Azul',22319,'Santiago','Comunidad Valenciana',617850831,'Bateria',1);
2019-01-09 15:01:19.921 CET [5188] ERROR:  permiso denegado a la relaci n Musicos
```

### • LOG VENDEDOR:

```
2019-01-09 15:36:27.003 CET [11172] LOG:  sentencia: SELECT "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_web"
FROM public."Grupo";
INSERT INTO public."Conciertos"(
  "Codigo_concierto", "Fecha_realizacion", "Pais", "Ciudad", "Recinto")
VALUES (0,'1939-11-6','Corea del norte','Fiji','Recinto ferial');
DELETE FROM "Entradas";
UPDATE public."Entradas"
SET "Localidad"='Madrid';
SELECT "Codigo_grupo_grupo", "Codigo_concierto_Conciertos"
FROM public."Grupos_Tocan_Conciertos";

2019-01-09 15:36:27.008 CET [11172] LOG:  duraci n: 5.311 ms
```

```
0 2019-01-09 15:37:38.073 CET [8988] LOG:  sentencia: INSERT INTO public."Grupo"(
1   "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_web")
2   VALUES (2, 'Nombre', 'Musica', 'Islandia', 'n.com');
3 2019-01-09 15:37:38.074 CET [8988] ERROR:  permiso denegado a la relaci n Grupo
```

**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.

En este caso si queremos que únicamente sea capaz de tener acceso a sus datos deberemos implementar una Row-Level-Policy, como antes hemos mencionado. En este caso debemos crear una expresión que únicamente permita validar los datos que han sido creados por ese usuario, devolviendo un booleano cuando así sea. Sin embargo, no va a ser necesario para todos los usuarios que hay en el sistema porque el administrador siempre tendrá que tener acceso a todos los datos de los usuarios. En cuanto al resto solo se les deberá mostrar sus datos y no el del resto.

Por tanto, las políticas que definiremos serán las siguientes:

Activación de las políticas de filas:

```
musicos on postgres@PostgreSQL 10
1 ALTER TABLE "Músicos" ENABLE ROW LEVEL SECURITY;
2 ALTER TABLE "Conciertos" ENABLE ROW LEVEL SECURITY;
3 ALTER TABLE "Canciones" ENABLE ROW LEVEL SECURITY;
4 ALTER TABLE "Discos" ENABLE ROW LEVEL SECURITY;
5 ALTER TABLE "Grupo" ENABLE ROW LEVEL SECURITY;
6 ALTER TABLE "Entradas" ENABLE ROW LEVEL SECURITY;
7 ALTER TABLE "Grupos_Tocan_Conciertos" ENABLE ROW LEVEL SECURITY;

Data Output Explain Messages Query History
ALTER TABLE

Query returned successfully in 62 msec.
```

Como identificador para músico usaremos su DNI, teniendo así cada nombre de usuario como DNI. Por lo que tenemos que cambiar el primero.

```
musicos on postgres@PostgreSQL 10
1 CREATE POLICY admin_all ON "Músicos" TO administrador USING (true);
2 CREATE POLICY admin_all ON "Grupo" TO administrador USING (true);
3 CREATE POLICY admin_all ON "Canciones" TO administrador USING (true);
4 CREATE POLICY admin_all ON "Discos" TO administrador USING (true);
5 CREATE POLICY admin_all ON "Conciertos" TO administrador USING (true);
6 CREATE POLICY admin_all ON "Entradas" TO administrador USING (true);
7 CREATE POLICY admin_all ON "Grupos_Tocan_Conciertos" TO administrador USING (true);
8

Data Output Explain Messages Query History
CREATE POLICY

Query returned successfully in 70 msec.
```

```

musicos on postgres@PostgreSQL 10
1 CREATE POLICY musico ON "Musicos" FOR ALL TO musico
2   USING (current_user = "DNI") ;
3 CREATE POLICY musico ON "Grupo" FOR SELECT TO musico
4   USING ("Codigo_grupo"= (SELECT "Codigo_grupo_Grupo"
5                             FROM "Musicos"
6                             WHERE current_user = "DNI"));
7 CREATE POLICY musico ON "Discos" FOR ALL TO musico
8   USING ("Codigo_grupo_Grupo"= (SELECT "Musicos"."Codigo_grupo_Grupo"
9                                     FROM "Musicos"
10                                    WHERE current_user = "DNI"));
11 CREATE POLICY musico ON "Canciones" FOR ALL TO musico
12   USING ("Codigo_disco_Discos"= (SELECT "Codigo_disco"
13                                     FROM "Discos"
14                                     WHERE "Discos"."Codigo_grupo_Grupo" = (
15                                       SELECT "Musicos"."Codigo_grupo_Grupo"
16                                       FROM "Musicos"
17                                       WHERE current_user = "DNI"))));

```

Data Output Explain Messages Query History

CREATE POLICY

Query returned successfully in 108 msec.

En el caso de las políticas de organizador y vendedor tenemos que modificar las tablas para añadir una nueva columna con el identificador del organizador del concierto en Conciertos o con el vendedor de las entradas en Entradas. Al hacer esto, seremos capaces de saber quién ha creado cada una y así solo mostrar sus datos. En el caso anterior hemos aprovechado que Musico tenía un DNI.

```

musicos on postgres@PostgreSQL 10
1 ALTER TABLE public."Entradas"
2   ADD COLUMN usuario "char"[] NOT NULL;

```

Data Output Explain Messages Query History

ALTER TABLE

Query returned successfully in 59 msec.

```

musicos on postgres@PostgreSQL 10
1 ALTER TABLE public."Conciertos"
2   ADD COLUMN usuario "char"[] NOT NULL;

```

Data Output Explain Messages Query History

ALTER TABLE

Query returned successfully in 56 msec.

```

musicos on postgres@PostgreSQL 10
1 CREATE POLICY organizador ON "Conciertos" FOR ALL TO organizador
2   USING (current_user = usuario);
3 CREATE POLICY organizador ON "Entradas" FOR ALL TO organizador
4   USING ("Codigo_concierto_Conciertos"= (SELECT "Codigo_concierto"
5                                                 FROM "Conciertos"
6                                                 WHERE current_user = "Conciertos"."usuario"));
7 CREATE POLICY organizador ON "Grupos_Tocan_Conciertos" FOR ALL TO organizador
8   USING ("Codigo_concierto_Conciertos"= (SELECT "Codigo_concierto"
9                                                 FROM "Conciertos"
10                                                WHERE current_user = "Conciertos"."usuario"));
11 CREATE POLICY organizador ON "Grupo" FOR SELECT TO organizador
12   USING ("Codigo_grupo"= (SELECT "Codigo_grupo_Grupo"
13                               FROM "Grupos_Tocan_Conciertos"
14                               WHERE "Grupos_Tocan_Conciertos"."Codigo_concierto_Conciertos" = (
15                                 SELECT "Conciertos"."Codigo_concierto"
16                                 FROM "Conciertos"
17                                 WHERE current_user = "Conciertos"."usuario"))));

```

Data Output Explain Messages Query History

CREATE POLICY

Query returned successfully in 93 msec.

```
musicos on postgres@PostgreSQL 10
1 CREATE POLICY vendedor ON "Entradas" FOR ALL TO vendedor
2 USING (current_user = usuario);
3 CREATE POLICY vendedor ON "Conciertos" FOR SELECT TO vendedor
4 USING ("Codigo_concierto"= (SELECT "Codigo_concierto_Conciertos"
5 FROM "Entradas"
6 WHERE current_user = "Entradas"."usuario"));
7 CREATE POLICY vendedor ON "Grupos_Tocan_Conciertos" FOR ALL TO vendedor
8 USING ("Grupos_Tocan_Conciertos"."Codigo_concierto_Conciertos"= (SELECT "Codigo_concierto_Conciertos"
9 FROM "Entradas"
10 WHERE current_user = "Entradas"."usuario"));
11 CREATE POLICY vendedor ON "Grupo" FOR SELECT TO vendedor
12 USING ("Codigo_grupo"= (SELECT "Codigo_grupo_Grupo"
13 FROM "Grupos_Tocan_Conciertos"
14 WHERE "Grupos_Tocan_Conciertos"."Codigo_concierto_Conciertos" = (
15 SELECT "Entradas"."Codigo_concierto_Conciertos"
16 FROM "Entradas"
17 WHERE current_user = "Entradas"."usuario"))));

Data Output Explain Messages Query History
CREATE POLICY
Query returned successfully in 90 msec.
```

**Cuestión 1.8:** Qué métodos de autenticación 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 actualmente dispone de 11 métodos de autenticación que son los siguientes que se muestran:

**Trust Authentication:** deja a cualquier usuario, con el nombre que quiera, conectarse a la base de datos. Aunque se siguen aplicando las restricciones de nombres de usuario. Es útil para conexiones locales y seguras, pero hay que evitarla para muchos usuarios.

**Password Authentication:** este método ya incluye contraseñas que se almacenarán en un archivo de PostgreSQL con la seguridad adecuada, pueden cambiar los protocolos de unos a otros.

**GSSAPI Authentication:** es un sistema de autenticación automática (sign-on) que es seguro, pero no encripta los mensajes por lo que se recomienda usarlo con SSL.

**SSPI Authentication:** al igual que el anterior es un sistema sign-on basado en Windows.

**Ident Authentication:** este método permite obtener el nombre del sistema operativo del cliente de un servidor de identificación para usarlo como nombre de la base de datos.

**Peer Authentication:** este método es similar al anterior, pero obteniendo el nombre del sistema del kernel, para después emplearlo en la base de datos. Este es exclusivo de conexiones locales.

**LDAP Authentication:** este método es parecido al de la contraseña (password), al comprobar la pareja usuario-contraseña, pero requiere que el usuario ya exista en la base de datos.

**RADIUS Authentication:** es casi idéntico al método anterior con la diferencia de que este emplea servidores RADIUS para la gestión de conexiones.

**Certificate Authentication:** éste emplea certificados SSL para validar las conexiones, permitiendo únicamente el login si el nombre es el correcto del certificado.

**PAM Authentication:** el funcionamiento es casi igual al de password empleando el sistema PAM.

**BSD Authentication:** Al igual que los anteriores es como password empleando BSD, aunque ahora solo se permite OpenBSD.

Para configurar la conexión remota es necesario modificar la configuración de nuestro PostgreSQL para ello cambiaremos el fichero postgresql.conf.

Tendremos que poner **listen\_addresses = '\*'** para que así se puedan atender todas las conexiones y poner el puerto que queremos, en nuestro caso el que usa PostgreSQL por defecto **port = 5432**. Otro añadido más es el de limitar el número de conexiones con **max\_connections**. Aquí la captura de nuestro fichero:

```
# - Connection Settings -  
  
listen_addresses = '*'          # what IP address(es) to listen on;  
                                # comma-separated list of addresses;  
                                # defaults to 'localhost'; use '*' for all  
                                # (change requires restart)  
port = 5432                     # (change requires restart)  
max_connections = 100           # (change requires restart)
```

Una vez cambiado eso, modificaremos el archivo pg\_hba.conf para especificar qué IP's queremos que puedan acceder, cambiando la configuración para el host:

**host all all 0.0.0.0/0 md5**

Con la IP 0.0.0.0/0 permitimos la conexión de cualquier IP, si quisiéramos restringirlo usaríamos la IP del equipo que queramos que acceda. Md5 es el tipo de seguridad que queremos que se emplee para la contraseña.

#	TYPE	DATABASE	USER	ADDRESS	METHOD
# IPv4 local connections:					
host	all	all	all	0.0.0.0/0	md5

Para que los cambios se completen hace falta que reiniciemos el ordenador. Ya solo nos hace falta nuestra IP que, como estamos en una red DHCP, solo será válida para nuestra red, para fuera de ella no.

```
Dirección IPv4. . . . . : 192.168.1.37(Preferido)
Máscara de subred . . . . . : 255.255.255.0
Concesión obtenida. . . . . : miércoles, 9 de enero de 2019 13:47:18
La concesión expira . . . . . : jueves, 10 de enero de 2019 7:47:19
```

Además, tuvimos otro problema relacionado con el firewall, por lo que cuando lo desactivamos pudimos realizarlo sin problema.

### Red de dominio

El firewall está desactivado.

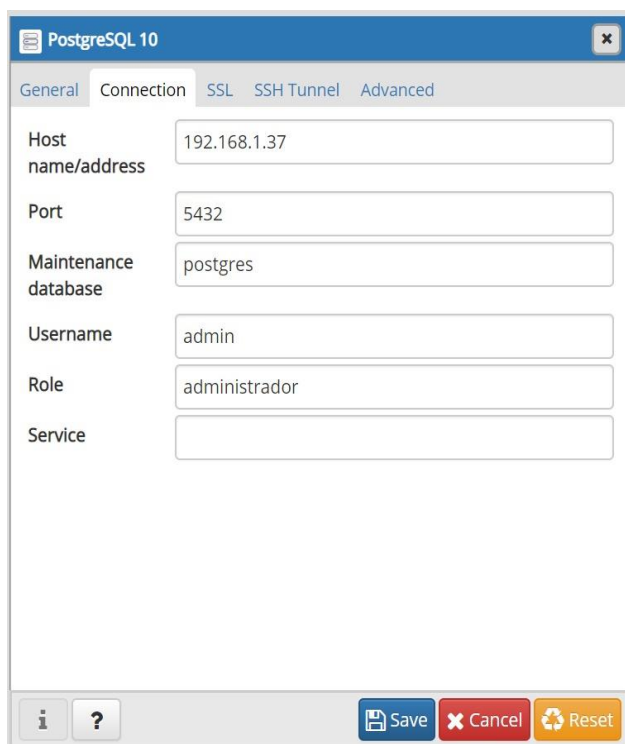
Activar

### Red privada (activa)

El firewall está desactivado.

Activar

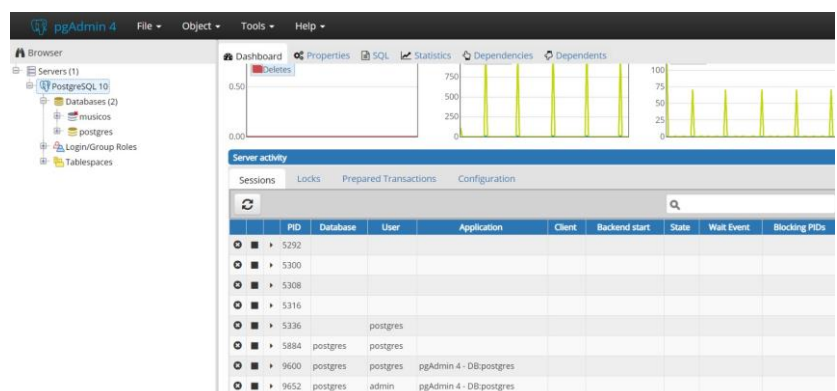
Estas son las capturas desde el otro ordenador para realizar la conexión:



The screenshot shows the 'PostgreSQL 10' connection dialog box. The 'Connection' tab is selected. The fields are filled with the following information:

Field	Value
Host name/address	192.168.1.37
Port	5432
Maintenance database	postgres
Username	admin
Role	administrador
Service	

At the bottom, there are buttons for 'Save', 'Cancel', and 'Reset'.



## 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.
- Concurrency.
- 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?

Para comprobar si el wal está activo podemos hacerlo con los comandos de wal de los que dispone PostgreSQL 10 en este caso:

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

En nuestro caso nos muestra lo siguiente:

musicos on postgres@PostgreSQL 10

1  
2

```
select *  
from pg_ls_waldir()
```

Data Output

Explain

Messages

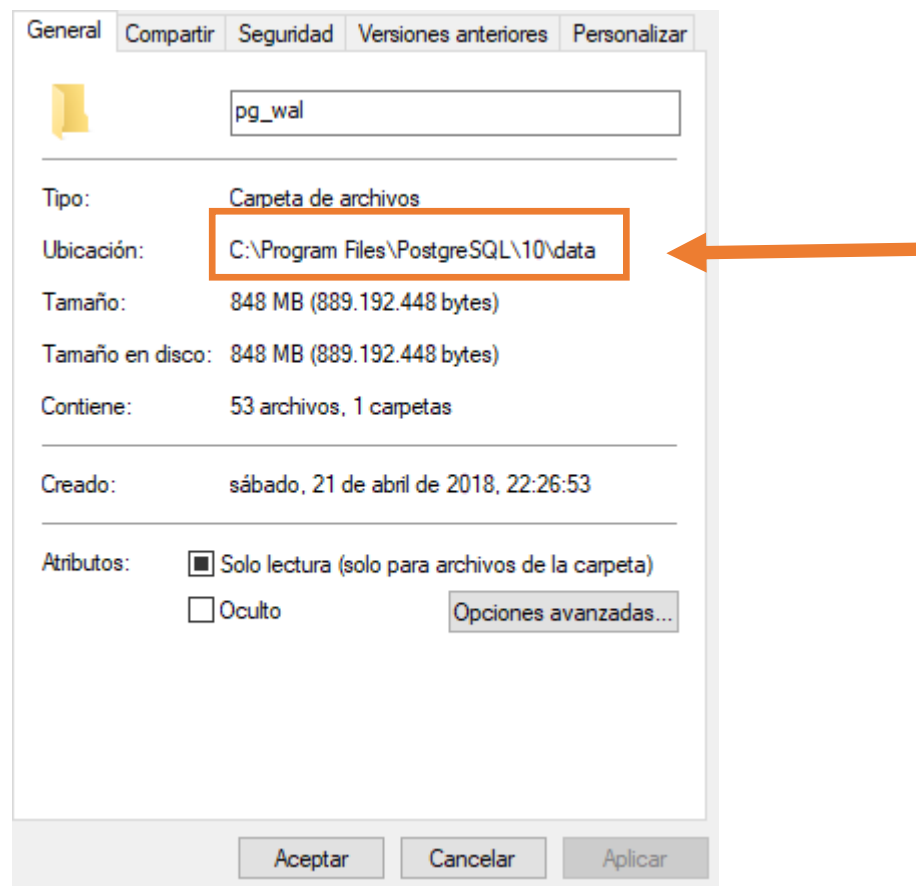
Query History

	name text	size bigint	modification timestamp with time zone
1	00000001000000010000003E	16777216	2019-01-09 20:32:11+01
2	00000001000000010000003F	16777216	2018-10-21 16:51:17+02
3	000000010000000100000040	16777216	2018-10-21 16:51:19+02
4	000000010000000100000041	16777216	2018-10-21 16:51:20+02
5	000000010000000100000042	16777216	2018-10-21 16:51:24+02
6	000000010000000100000043	16777216	2018-10-21 16:51:22+02
7	000000010000000100000044	16777216	2018-10-21 16:51:22+02
8	000000010000000100000045	16777216	2018-10-21 16:51:22+02
9	000000010000000100000046	16777216	2018-10-21 16:51:23+02
10	000000010000000100000047	16777216	2018-10-22 00:45:43+02
11	000000010000000100000048	16777216	2018-10-21 16:51:23+02
12	000000010000000100000049	16777216	2018-10-21 16:51:24+02
13	00000001000000010000004A	16777216	2018-10-21 16:51:24+02
14	00000001000000010000004B	16777216	2018-10-21 16:51:24+02
15	00000001000000010000004C	16777216	2018-10-21 16:53:54+02
16	00000001000000010000004D	16777216	2018-10-21 16:51:25+02
17	00000001000000010000004E	16777216	2018-10-21 16:51:25+02
18	00000001000000010000004F	16777216	2018-10-21 16:51:25+02
19	000000010000000100000050	16777216	2018-10-21 16:51:26+02
20	000000010000000100000051	16777216	2018-10-21 19:26:26+02

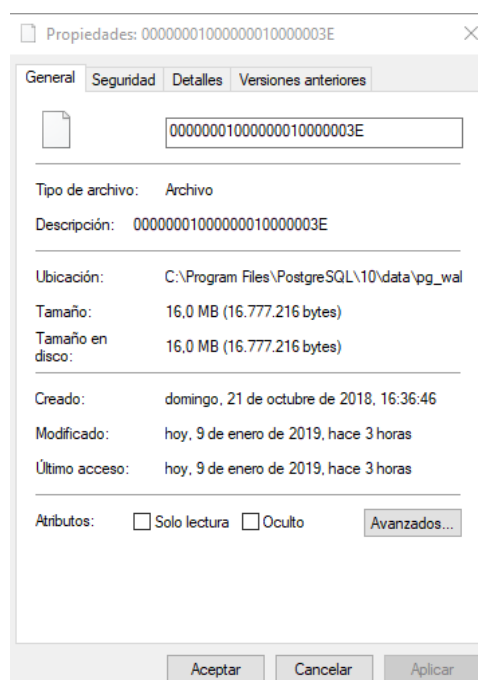


Como podemos ver tiene bastantes archivos generados por el WAL, por tanto, podemos determinar que está activo y que lo que muestra son los distintos archivos WAL que ha ido generando.

El WAL se almacena en una carpeta llamada Pg\_wal dentro de los archivos de PostgreSQL, siendo la ruta completa en nuestro sistema la siguiente:



El tamaño de cada archivo es de 16MB porque así está configurado por defecto por PostgreSQL:





Al abrirlo con el editor de texto nos encontramos con muchos símbolos y caracteres sin sentido junto a palabras entendibles, esto se debe a que está codificado en otro formato; probablemente en hexadecimal:



**Cuestión 2.2:** ¿Qué parámetros se pueden configurar del diario del sistema de postgres? ¿Para qué sirven cada uno de ellos?

Los parámetros que podemos configurar del diario del sistema son los que hay en el archivo postgresql.conf, que se ven en la siguiente imagen:

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

# - Settings -

#wal_level = replica          # minimal, replica, or logical
                              # (change requires restart)
#fsync = on                   # flush data to disk for crash safety
                              # (turning this off can cause
                              # unrecoverable data corruption)
#synchronous_commit = on     # synchronization level;
                              # off, local, remote_write, remote_apply, or on
#wal_sync_method = fsync      # the default is the first option
                              # supported by the operating system:
                              #   open_datasync
                              #   fdatasync (default on Linux)
                              #   fsync
                              #   fsync_writethrough
                              #   open_sync
#full_page_writes = on       # recover from partial page writes
#wal_compression = off       # enable compression of full-page writes
#wal_log_hints = off         # also do full page writes of non-critical updates
                              # (change requires restart)
#wal_buffers = -1            # min 32kB, -1 sets based on shared_buffers
                              # (change requires restart)
#wal_writer_delay = 200ms     # 1-10000 milliseconds
#wal_writer_flush_after = 1MB # measured in pages, 0 disables

#commit_delay = 0            # range 0-100000, in microseconds
#commit_siblings = 5         # range 1-1000

# - Checkpoints -

#checkpoint_timeout = 5min    # range 30s-1d
#max_wal_size = 1GB
#min_wal_size = 80MB
#checkpoint_completion_target = 0.5 # checkpoint target duration, 0.0 - 1.0
#checkpoint_flush_after = 0    # measured in pages, 0 disables
#checkpoint_warning = 30s     # 0 disables

# - Archiving -

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

**wal\_level** indica el nivel de información que se va a escribir. Hay tres opciones: minimal; que hará lo mínimo siendo más eficiente, pero no permite una recuperación completa, replica y logical, que permiten ya una recuperación y son necesarios si se quiere poner como archived.

**fsync** permite las actualizaciones físicas en el disco. Si no está activado el rendimiento mejora, pero esto supone tener información irrecuperable.

**synchronous\_commit** sirve para indicar que espere a que los resultados se guarden en el log. Si está apagado no hay inconsistencia de datos, aunque se abortan más operaciones.

**wal\_sync\_method** sirve para forzar a escribir el WAL, solo tiene sentido si fsync está activo.

**full\_page\_writes:** aunque empeore el rendimiento es necesaria su activación, ya que permite tener datos consistentes. Sirve para almacenar completamente la página de disco, permitiendo después su restauración.

**wal\_compression** su utilidad es comprimir las páginas de discos en el WAL, lo que supone un coste extra de CPU; al comprimir primero la página y después descomprimir las páginas para recuperar datos, pero ahorra bastante memoria de disco.

**wal\_log\_hints** al activarlo se escribe la página de disco en el WAL después de un checkpoint.

**wal\_buffers** es la cantidad de memoria compartida que se le asignará al WAL. La memoria no debe de ser mayor de 16 MB, el tamaño de un WAL, ni menor de 64KB.

**wal\_writer\_delay** sirve para añadir un tiempo de retraso en el inicio de la escritura en el WAL, hacer esto puede permitir que se realicen más operaciones, aunque el sistema tardará más.

**wal\_writer\_flush\_after** se usa para indicar con qué frecuencia se realizará un flush en el WAL.

**commit\_delay** funciona de forma similar al writer\_delay, permitiendo añadir microsegundos para realizar más operaciones antes del flush del WAL.

**commit\_siblings** es el número mínimo de transacciones necesarias que se encuentren de forma concurrente para usar el commit\_delay.

**checkpoint\_timeout** sirve para indicar el tiempo entre WAL de checkpoints. Si el tiempo es muy alto el tiempo de recuperación será mayor.

**checkpoint\_completion\_target** especifica el objetivo de completitud de un checkpoint como una fracción del timeout anterior.

**checkpoint\_flush\_after** sirve para forzar al SO a escribir en el almacenamiento los bytes que está escribiendo el flush mientras realiza un checkpoint. Puede ser útil para reducir la latencia de la transacción, aunque no siempre lo hace.

**checkpoint\_warning** sirve para escribir un mensaje en el log si el tiempo entre checkpoints se acerca mucho.

**max\_wal\_size** es el máximo tamaño entre WALs automáticos de checkpoints aunque este límite se puede sobrepasar en determinadas ocasiones, por ejemplo si hubiera mucha carga. Si se da un valor muy alto, el tiempo de recuperación aumentará bastante.

**min\_wal\_size** como se reciclan los WAL, este valor nos permite asegurar suficiente memoria para picos en los que se use más el WAL.

**archive\_mode** cuando está activo se guardarán en archivos los segmentos de WAL completados, para su uso se requiere tener activado el modo replica o logical del wal\_level.

**archive\_command** sirve para ejecutar comandos de la shell para archivar un WAL completado.

**archive\_timeout** se usa para forzar a realizar WALs periódicamente en el caso de que tuviera poco tráfico.

**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?

Hemos realizado la inserción de un nuevo grupo a la base de datos, siendo la siguiente operación:

```
musicos on postgres@PostgreSQL 10

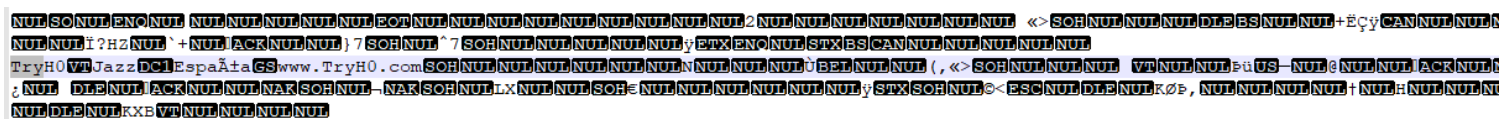
1  INSERT INTO public."Grupo"(
2    "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_web")
3    VALUES (0,'TryH0','Jazz','España','www.TryH0.com');

Data Output  Explain  Messages  Query History

INSERT 0 1

Query returned successfully in 101 msec.
```

Para comprobar que se ha registrado hemos abierto el WAL y hemos buscado los datos que hemos insertado. En este caso hemos comprobado que sí que estaban dentro, siendo además legibles como se ve en esta captura:



Esto ocurre porque se trata de una operación que altera la base de datos, por lo que se tiene que tener constancia de que se han insertado esos datos. Así, después cuando toque una recuperación de datos, se podrá recuperar esta inserción a partir del checkpoint anterior.

**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.

Waldump es una herramienta que ofrece PostgreSQL para entender el volcado del WAL. Al usar esta herramienta con el archivo WAL que tenemos, PostgreSQL hará que el contenido que antes no era entendible lo sea ahora, además podemos pasarle una serie de parámetros para indicarle qué datos queremos, obtener estadísticas entre otras cosas.

Una vez lo aplicamos al último WAL, este irá desglosando todo el contenido que tiene en los distintos tipos de archivos, indicando información relativa al sector en el que se encuentra en el disco: el bloque que ocupa, el tamaño o el tipo de archivo, que es entre los que podemos ver btree, heap, xlog, storage, standby o transaction. En esta última se dan más datos sobre la memoria que emplea con cache. En storage y standby también se pueden ver otros datos como LOCK para bloquear y CREATE para crear elementos. Se puede ver gran parte de lo mencionado en la siguiente captura:

[illegible]

Si queremos obtener las estadísticas tenemos que añadir el parámetro -z al comando para que muestre las estadísticas totales de nuestro archivo WAL. El resultado se muestra a continuación:

Type	N	(%)	Record size	(%)	FPI size	(%)	Combined size	(%)
----	-	---	-----	---	-----	---	-----	---
XLOG	796	( 6,25)	46034	( 3,84)	4115196	( 41,26)	4161230	( 37,24)
Transaction	353	( 2,77)	104128	( 8,69)	0	( 0,00)	104128	( 0,93)
Storage	116	( 0,91)	4880	( 0,41)	0	( 0,00)	4880	( 0,04)
CLOG	0	( 0,00)	0	( 0,00)	0	( 0,00)	0	( 0,00)
Database	6	( 0,05)	228	( 0,02)	0	( 0,00)	228	( 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	618	( 4,85)	29580	( 2,47)	0	( 0,00)	29580	( 0,26)
Heap2	133	( 1,04)	58139	( 4,85)	191048	( 1,92)	249187	( 2,23)
Heap	4640	( 36,44)	519648	( 43,35)	2017820	( 20,23)	2537468	( 22,71)
Btree	6072	( 47,68)	436121	( 36,38)	3650236	( 36,57)	4086357	( 36,57)
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	12734		1198758	[10,73%]	9974300	[89,27%]	11173058	[100%]
pg_waldump: FATAL: error en registro de WAL en 1/3EAB8420: invalid record length at 1/3EAB8458: wanted 24, got 0								

C:\Program Files\PostgreSQL\10\data\pg\_wal>

Analizando las estadísticas podemos ver distintas columnas. La primera, **type**, nos indica el registro o el manejador de recursos. **N** indica el número de registros de ese tipo que hay, **%** indica el porcentaje de transacciones del total. **Record size** es el tamaño de los registros y **%** el porcentaje del total. **FPI** es el tamaño de las imágenes de las páginas completas (Full-page-image), y el **%** indica el porcentaje del total de imágenes. **Combined size** es la combinación de los dos anteriores, y al igual que antes también muestra porcentaje. Si nos centramos en el contenido que tenemos podemos ver que la mayoría del WAL se centra en registros de tipo Btree y Heap, lo que nos puede indicar que actualiza bastante índices de esos tipos o registros de esos tipos. Después les sigue XLOG que se corresponde con los archivos WAL, dado que en la versión 10 de PostgreSQL le cambiaron el nombre por uno más entendible. También



destacan los registros de las transacciones, que son las operaciones que se realizan sobre las tablas.

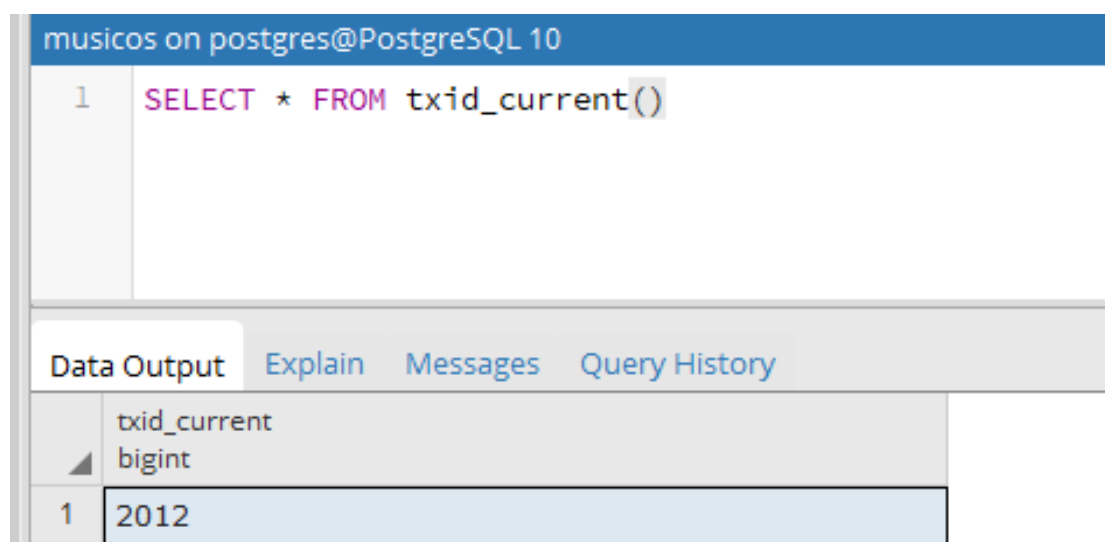
**Cuestión 2.5:** Determinar el identificador de la transacción que realizó la operación anterior. Aplicar el comando anterior al último fichero de WAL que se ha generado y mostrar los registros que se han creado para esa transacción. ¿Qué se puede ver? Interpretar los resultados obtenidos.

Para determinar el identificador de la transacción tenemos que obtener el XID, elemento que cada transacción lleva asociado, esto podemos hacerlo a través de la siguiente consulta:

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

Ésta mostrará el XID de la transacción actual por lo que habrá que restarle hasta llegar a la que nos interesa.

En nuestro caso entre fallo y error creamos 3 transacciones más por lo que nos daba 2012, cuando en realidad era el 2009. Véase la captura:



The screenshot shows a PostgreSQL terminal window with the title 'musicos on postgres@PostgreSQL 10'. The command 'SELECT \* FROM txid\_current();' is entered and executed. Below the command, there are tabs for 'Data Output', 'Explain', 'Messages', and 'Query History'. The 'Data Output' tab is selected, showing a table with one column 'txid\_current' of type 'bigint'. The table contains one row with the value '2012'.

	txid_current
1	2012

Una vez sabemos el XID, solo queda pasarlo como parámetro en el pg\_waldump para así obtener los datos del WAL de la transacción, añadiendo simplemente -x XID:

```
C:\Program Files\PostgreSQL\10\data\pg_wal>pg_waldump -x 2009 00000001000000010000003E
rmgr: Heap      len (rec/tot): 92/ 92, tx: 2009, lsn: 1/3EAB8228, prev 1/3EAB81F0, desc: INSERT+INIT off 1, blkref #0: rel 1663/79741/79752 blk 0
rmgr: Btree     len (rec/tot): 78/ 78, tx: 2009, lsn: 1/3EAB8228, prev 1/3EAB8228, desc: NEWROOT lev 0, blkref #0: rel 1663/79741/79758 blk 1, blkref #2: rel 1663/79741/79758 blk 0
rmgr: Btree     len (rec/tot): 64/ 64, tx: 2009, lsn: 1/3EAB82D8, prev 1/3EAB8288, desc: INSERT_LEAF off 1, blkref #0: rel 1663/79741/79758 blk 1
rmgr: Transaction len (rec/tot): 34/ 34, tx: 2009, lsn: 1/3EAB8318, prev 1/3EAB82D8, desc: COMMIT 2019-01-10 14:49:13.139979 Hora estándar romance
```

Haciendo un análisis de lo que contiene podemos ver en orden de izquierda a derecha: el rgmr, es el manejador de recursos o registro, en nuestro caso todos los que hay asociados son de tipo Btree y Heap, relacionados con PostgreSQL y Transaction que es el de la transacción que hemos realizado. Len nos muestra el número de registros del total, que en este caso coinciden siendo 34 la de la transacción. El tx es el id de la transacción y las operaciones relacionadas, que en nuestro caso es 2009. El parámetro lsn (Log sequence number) es el número de secuencia del log, son 2 números hexadecimales de hasta 8 dígitos cada uno separados por una barra; sirven como punteros a una localización en el log, en nuestro caso

tenemos el actual y el anterior lsn, que como vemos coincide; si cogemos, por ejemplo, el valor de prev del último de todos y lo comparamos con el lsn que tiene el anterior. Finalmente tenemos desc, la descripción de las operaciones realizadas, en HEAP podemos observar como realiza un INSERT+INIT lo que significa que está insertando datos, pero también inicializando la tabla; además se indica después con off el offset que tiene junto a otros datos de los sectores del disco. Si nos centramos en Transaction podemos ver como realiza el COMMIT, es decir, compromete los datos del INSERT y hace que se guarde la operación junto a la hora de su realización

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

Procederemos a realizar el backup siguiendo las indicaciones que ofrece PostgreSQL en su documentación.


Lo primero será activar la opción de archivar WAL:

```
# - Settings -
wal_level = replica          # minimal, replica, or logical
                              # (change requires restart)
#fsync = on                  # flush data to disk for crash safety
                              # (turning this off can cause
                              # unrecoverable data corruption)
#synchronous_commit = on    # synchronization level;
                              # off, local, remote_write, remote_apply, or on
#wal_sync_method = fsync     # the default is the first option
                              # supported by the operating system:
                              #   open_datasync
                              #   fdatasync (default on Linux)
                              #   fsync
                              #   fsync_writethrough
                              #   open_sync
#full_page_writes = on      # recover from partial page writes
#wal_compression = off      # enable compression of full-page writes
#wal_log_hints = off        # also do full page writes of non-critical updates
                              # (change requires restart)
#wal_buffers = -1           # min 32kB, -1 sets based on shared_buffers
                              # (change requires restart)
#wal_writer_delay = 200ms    # 1-10000 milliseconds
#wal_writer_flush_after = 1MB # measured in pages, 0 disables

#commit_delay = 0           # range 0-100000, in microseconds
#commit_siblings = 5        # range 1-1000

# - Checkpoints -
#checkpoint_timeout = 5min   # range 30s-1d
#max_wal_size = 1GB
#min_wal_size = 80MB
#checkpoint_completion_target = 0.5 # checkpoint target duration, 0.0 - 1.0
#checkpoint_flush_after = 0    # measured in pages, 0 disables
#checkpoint_warning = 30s     # 0 disables

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



Después tendremos que lanzar una query en la que crearemos la copia de seguridad. Como label le pondremos la ruta del backup.

```
musicos on postgres@PostgreSQL 10
```

```
1 SELECT pg_start_backup('C:\\Backup\\BackupsMusicos');
```

	pg_start_backup	pg_lsn
1		1/43000028

Solo nos queda comprobar que se ha realizado con éxito, para eso lo examinamos en el árbol de directorios del equipo, dónde comprobamos que se ha hecho correctamente:

Este equipo > Disco local (C:) > Backup					
	Nombre	Fecha de modifica...	Tipo	Tamaño	
ido	00000001000000010000003E	10/01/2019 17:07	Archivo	16.384 KB	
	00000001000000010000003F	10/01/2019 17:07	Archivo	16.384 KB	
	000000010000000100000040	10/01/2019 17:07	Archivo	16.384 KB	
tos	000000010000000100000041	21/10/2018 16:51	Archivo	16.384 KB	
	000000010000000100000041.00000028.backup	10/01/2019 17:17	Archivo BACKUP	1 KB	
	000000010000000100000042	21/10/2018 16:51	Archivo	16.384 KB	

**Cuestión 2.7:** Qué herramientas disponibles tiene PostgreSQL para controlar la actividad de la base de datos en cuanto a la concurrencia y transacciones? ¿Qué información es capaz de mostrar? ¿Dónde se guarda dicha información? ¿Cómo se puede mostrar?

PostgreSQL proporciona por defecto la herramienta MVCC (Multiversion Concurrency Control), un modelo de multiversiones. Este sistema permite a varias transacciones concurrentes tener acceso a información consistente, dando aislamiento de transacciones para cada sesión de la base de datos. Funciona mediante instancias o instantáneas de la base de datos, que tienen los datos que había en la base de datos en el momento de la transacción, esto permite que las transacciones puedan trabajar independientemente solucionando los problemas de la concurrencia. Generalmente, si está bien configurado, tendrá un mejor rendimiento que los bloqueos, ya que permite lectura y escritura, una de cada a la vez, sin tener conflicto alguno.



Además de esta herramienta PostgreSQL dispone de otras herramientas alternativas a MVCC.

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

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

Otra herramienta que se ofrece es el aislamiento de transacciones. Tenemos 4 tipos Read uncommitted, Read committed, Repeatable read y Serializable. Estos tipos van de menos restrictivos a más restrictivos, aunque solo se usan los últimos tres siendo el último casi de forma serial.

La información que PostgreSQL puede mostrar son todos los locks que hay en el sistema, que guarda con pg\_locks, que puede consultarse con una vista, también están las transacciones bloqueadas. Para el aislamiento no se guardan datos porque se ha de especificar en la transacción o sesión, aunque sí es posible exportarlos para un uso posterior.

Aquí se puede ver el contenido de pg\_locks, que puede ser ampliado si se junta con pg\_stat\_activity:

PostgreSQL 10 - musicos - pg\_locks.pg\_locks

```
1 SELECT * FROM pg_catalog.pg_locks
2
```

Data Output Explain Messages Query History

locktype text	database oid	relation oid	page integer	tuple smallint	virtualxid oid	transactionid xid	classid oid	objid oid	objsubid smallint	virtualtransaction text	pid integer	mode text	granted boolean	fastpath boolean
1 relation	79741	11577	[null]	[null]	[null]	[null]			[null]	7/841	4640	AccessShareLock	true	true
2 virtualxid			[null]	[null]	7/841	[null]			[null]	7/841	4640	ExclusiveLock	true	true

musicos on postgres@PostgreSQL 10

```
1 SELECT * FROM pg_locks pl LEFT JOIN pg_stat_activity psa
2 ON pl.pid = psa.pid;
```

Data Output Explain Messages Query History

locktype text	database oid	relation oid	page integer	tuple smallint	virtualxid oid	tran xid	clas oid	objid oid	objsubid smallint	virtualtransaction text	pid integer	mode text	granted boolean	fastpath boolean	datid oid	datname name	pid integer	usesysid oid	username name	application_name	client_addr inet
1 relation	79741	11682	[null]	[null]	[null]	[null]		[null]		8/13	11012	AccessShareLock	true	true	79741	musicos	11012	10	postgres	pgAdmin 4 - CONN25...	192.168.1.37
2 relation	79741	11577	[null]	[null]	[null]	[null]		[null]		8/13	11012	AccessShareLock	true	true	79741	musicos	11012	10	postgres	pgAdmin 4 - CONN25...	192.168.1.37
3 virtualxid			[null]	[null]	8/13	[null]		[null]		8/13	11012	ExclusiveLock	true	true	79741	musicos	11012	10	postgres	pgAdmin 4 - CONN25...	192.168.1.37
4 relation	0	2676	[null]	[null]	[null]	[null]		[null]		8/13	11012	AccessShareLock	true	false	79741	musicos	11012	10	postgres	pgAdmin 4 - CONN25...	192.168.1.37
5 relation	0	2671	[null]	[null]	[null]	[null]		[null]		8/13	11012	AccessShareLock	true	false	79741	musicos	11012	10	postgres	pgAdmin 4 - CONN25...	192.168.1.37
6 relation	0	1262	[null]	[null]	[null]	[null]		[null]		8/13	11012	AccessShareLock	true	false	79741	musicos	11012	10	postgres	pgAdmin 4 - CONN25...	192.168.1.37
7 relation	0	1260	[null]	[null]	[null]	[null]		[null]		8/13	11012	AccessShareLock	true	false	79741	musicos	11012	10	postgres	pgAdmin 4 - CONN25...	192.168.1.37
8 relation	0	2677	[null]	[null]	[null]	[null]		[null]		8/13	11012	AccessShareLock	true	false	79741	musicos	11012	10	postgres	pgAdmin 4 - CONN25...	192.168.1.37
9 relation	0	2672	[null]	[null]	[null]	[null]		[null]		8/13	11012	AccessShareLock	true	false	79741	musicos	11012	10	postgres	pgAdmin 4 - CONN25...	192.168.1.37

**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?

Para realizar una transacción basta con indicarle begin, pero como no hay que cerrarla no es necesario hacer commit, es decir, comprometer los datos con el sistema.

musicos on postgres@PostgreSQL 10

```
1 BEGIN;
2 INSERT INTO public."Grupo"(
3     "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_web")
4     VALUES (1,'SingY1','Pop','España','www.SingY1.com');
5 SELECT * FROM "Grupo"
```

Data Output Explain Messages Query History

	Codigo_grupo integer	Nombre text	Genero_musical text	Pais text	Sitio_web text
1	0	TryH0	Jazz	Espa...	www.TryH0...
2	1	SingY1	Pop	Espa...	www.SingY...

Al realizar la transacción, primero inserta el dato y después muestra los datos que contiene en memoria para esta transacción; lo que no implica que los datos se hallan comprometido. Esto se puede comprobar mostrando los datos de grupo para las dos operaciones, como hemos hecho. Por tanto, al comparar las capturas vemos que al no hacer COMMIT ningún dato ha sido insertado, sino que sigue a la espera de COMMIT.

musicos on postgres@PostgreSQL 10

```
1 SELECT * FROM public."Grupo";
```

Data Output Explain Messages Query History

	Codigo_grupo integer	Nombre text	Genero_musical text	Pais text	Sitio_web text
1	0	TryH0	Jazz	Espa...	www.TryH0...

Esto se puede ver consultando en `pg_locks`, que nos indica todos los bloqueos que hay en el sistema y las transacciones que se encuentran bloqueadas. En este caso esto es lo que muestra:

```
musicos on postgres@PostgreSQL 10

1  SELECT locktype, transactionid, mode, granted , query,wait_event, wait_event_type FROM pg_locks pl LEFT JOIN pg_stat_activity psa
2  ON pl.pid = psa.pid;
```

	locktype text	transactionid xid	mode text	granted boolean	query text	wait_event text	wait_event_type text
1	relation	[null]	AccessShareLock	true	SELECT locktype, transactionid, mode, granted , query,wait_event, ...	[null]	[null]
2	relation	[null]	AccessShareLock	true	SELECT locktype, transactionid, mode, granted , query,wait_event, ...	[null]	[null]
3	virtualxid	[null]	ExclusiveLock	true	SELECT locktype, transactionid, mode, granted , query,wait_event, ...	[null]	[null]
4	relation	[null]	AccessShareLock	true	BEGIN;	ClientRead	Client
5	relation	[null]	AccessShareLock	true	BEGIN;	ClientRead	Client
6	relation	[null]	RowExclusiveLock	true	BEGIN;	ClientRead	Client
7	virtualxid	[null]	ExclusiveLock	true	BEGIN;	ClientRead	Client
8	relation	[null]	AccessShareLock	true	SELECT locktype, transactionid, mode, granted , query,wait_event, ...	[null]	[null]
9	relation	[null]	AccessShareLock	true	SELECT locktype, transactionid, mode, granted , query,wait_event, ...	[null]	[null]
10	relation	[null]	AccessShareLock	true	SELECT locktype, transactionid, mode, granted , query,wait_event, ...	[null]	[null]
11	relation	[null]	AccessShareLock	true	SELECT locktype, transactionid, mode, granted , query,wait_event, ...	[null]	[null]
12	relation	[null]	AccessShareLock	true	SELECT locktype, transactionid, mode, granted , query,wait_event, ...	[null]	[null]
13	relation	[null]	AccessShareLock	true	SELECT locktype, transactionid, mode, granted , query,wait_event, ...	[null]	[null]
14	transactionid	2031	ExclusiveLock	true	BEGIN;	ClientRead	Client

En esta consulta hemos seleccionado los campos más interesantes para ver que la transacción se corresponde con el BEGIN, que posee un ExclusiveLock, este se debe a que iba a realizar una escritura en la base de datos. Además, este Lock se lo han concedido por lo que no habrá conflictos para realizar la transacción. Otro dato que muestra que es interesante es que se encuentra esperando al cliente, esto significa que está esperando a algo que es ajeno a la base de datos, que depende exclusivamente del cliente, que en nuestro caso sería el COMMIT.

Por tanto, si hubiésemos hecho COMMIT hubiera sido una transacción de escritura satisfactoria, al otorgarse el ExclusiveLock sin ningún conflicto. Aunque al no hacerlo estos datos no se han escrito como hemos comprobado con los SELECT.

**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.

Ahora tenemos que volver a aplicar el mismo concepto que antes sin cerrar la transacción por lo que la consulta sería la siguiente:

```

1 BEGIN;
2 INSERT INTO public."Grupo"(
3     "Codigo_grupo", "Nombre", "Genero_musical", "País", "Sitio_web")
4     VALUES (1,'SingY1','Pop','España','www.SingY1.com');
5 INSERT INTO public."Músicos"(
6     codigo_musico, "DNI", "Nombre", "Direccion", "Codigo_Postal", "Ciudad", "Provincia", telefono, "Instrumentos", "Codigo_grupo_Grupo")
7     VALUES (1,'00','Pedro','Calle Azul',22319,'Santiago','Comunidad Valenciana',617850831,'Bateria',1);
8 INSERT INTO public."Discos"(
9     "Codigo_disco", "Titulo", "Fecha_edicion", "Genero", "Formato", "Codigo_grupo_Grupo")
10    VALUES (1,'VillancicosJ0','1966-3-20','jazz','Vinilo',1);

```

Data Output Explain Messages Query History

INSERT 0 1

Query returned successfully in 114 msec.

**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.

Para realizar la consulta hemos hecho varios SELECT.

musicos on postgres@PostgreSQL 10

1

2

SELECT \* FROM "Grupo";

Data Output

Explain

Messages

Query History

	Codigo_grupo integer	Nombre text	Genero_musical text	Pais text	Sitio_web text
1	0	TryH0	Jazz	Espa...	www.TryH0...

musicos on postgres@PostgreSQL 10

1

2

SELECT \* FROM "Discos";

Data Output

Explain

Messages

Query History

Codigo\_disco  
integer

Titulo  
text

Fecha\_edicion  
date

Genero  
text

Formato  
text

Codigo\_grupo\_Grupo  
integer

musicos on postgres@PostgreSQL 10

1

SELECT \* FROM "Músicos"

Data Output

Explain

Messages

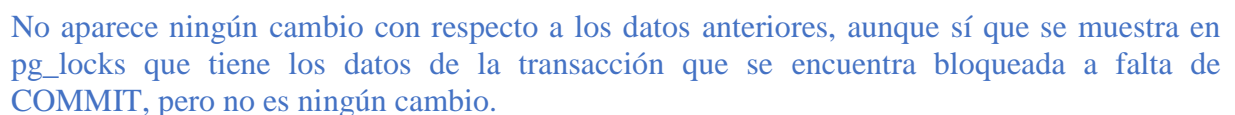
Query History

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
--------------------------	-----------------------	----------------	-------------------	--------------------------	----------------	-------------------	---------------------	----------------------	-------------------------------

Sin embargo, no se muestra ningún dato nuevo, esto es correcto porque al igual que hemos explicado antes, al no hacer COMMIT no se comprometen los datos por lo que únicamente estarán vivos en memoria local de la transacción y no en la memoria compartida por todos.

Por tanto, no mostrará ningún dato nuevo de las inserciones al no haberse comprometido con COMMIT.

Para establecer la conexión empleamos el mismo método que usamos en la parte 1, usando otro equipo. Como se puede ver en la imagen:

[illegible]

**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?

Estos datos no se encuentran en las tablas de las bases de datos que tienen en común toda la base de datos, sino que se encuentran en la memoria local. Al estar en la memoria local, sólo es accesible para la propia transacción y no para el resto. Los resultados de las operaciones anteriores se obtienen de la memoria compartida en la que los datos no se han comprometido teniendo como resultado en ambas los valores de la memoria.

Por tanto, todos los datos los saca de la memoria que tienen en común que en nuestro caso no se ha actualizado al no finalizar la transacción. Los datos que hay en la transacción se almacenan en la memoria local.

**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?

Para finalizar con éxito la transacción T1 hay que añadir COMMIT al final para que se comprometan los datos. La consulta a realizar es la siguiente:

```
musicos on admin@PostgreSQL 10
1 BEGIN;
2 INSERT INTO public."Grupo"(
3     "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_web")
4     VALUES (1,'SingY1','Pop','España','www.SingY1.com');
5 INSERT INTO public."Musicos"(
6     codigo_musico, "DNI", "Nombre", "Direccion", "Codigo_Postal", "Ciudad", "Provincia", telefono, "Instrumentos", "Codigo_grupo_grupo")
7     VALUES (1,'0U','Pedro','Calle Azul',22319,'Santiago','Comunidad Valenciana',617850831,'Bateria',1);
8 INSERT INTO public."Discos"(
9     "Codigo_disco", "Titulo", "Fecha_edicion", "Genero", "Formato", "Codigo_grupo_grupo")
10    VALUES (1,'VillancicosJ0','1966-3-20','jazz','Vinilo',1);
11 COMMIT;
```

COMMIT

Query returned successfully in 114 msec.

Ahora, si miramos pg\_locks veremos que ya no hay ninguna transacción bloqueada esperando. Por lo que sabemos que ya si se ha realizado la operación.

Para comprobarlo haremos SELECT como antes para ver qué datos muestra ahora nuestra base de datos. Primero en la que teníamos abierta y después con la otra conexión.

- **SESIONES:**

			PID	User	Application	Client	Backend start	State	Wait Event	Blocking PID
✖	■	▶	1496	admin	pgAdmin 4 - CONN:4844105	192.168.1.37	2019-01-10 22:36:52 CET	idle	Client: ClientRead	
✖	■	▶	4660	admin	pgAdmin 4 - CONN:7207385	192.168.1.37	2019-01-10 22:37:33 CET	idle	Client: ClientRead	
✖	■	▶	7176	admin	pgAdmin 4 - CONN:6151506	192.168.1.37	2019-01-10 21:55:42 CET	idle	Client: ClientRead	
✖	■	▶	10680	postgres	pgAdmin 4 - DB:musicos	192.168.1.40	2019-01-10 22:40:20 CET	idle	Client: ClientRead	
✖	■	▶	14324	admin	pgAdmin 4 - DB:musicos	192.168.1.37	2019-01-10 21:55:22 CET	idle	Client: ClientRead	

- **SELECTS PARA COMPROBAR DATOS:**

PostgreSQL 10 - músicos - public.Grupo

```

1  SELECT * FROM public."Grupo"
2  ORDER BY "Codigo_grupo" ASC

```

Data Output Explain Messages Query History

	Codigo_grupo [PK] integer	Nombre text	Genero_musical text	Pais text	Sitio_web text
1	0	TryH0	Jazz	Espa...	www.TryH0...
2	1	SingY1	Pop	Espa...	www.SingY...

músicos on postgres@PostgreSQL 10

```

1  SELECT * FROM public."Músicos"
2  ORDER BY codigo_musico ASC

```

Data Output Explain Messages Notifications Query History

	codigo_musico [PK] integer	DNI character (10)	Nombre text	Direccion text	Codigo_Postal integer	Ciudad text	Provincia text	telefono integer	Instrumentos text
1	1	0U	Pedro	Calle Azul	22319	Santiago	Comunida...	617850831	Bateria

**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?

Hay que repetir los INSERT de antes, pero añadiendo ROLLBACK al final, quedando así la consulta:

músicos on admin@PostgreSQL 10

```

1  BEGIN;
2  INSERT INTO public."Grupo"(
3    "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_web")
4    VALUES (2,'SingX2','Rock','España','www.SingX2.com');
5  INSERT INTO public."Músicos"(
6    codigo_musico, "DNI", "Nombre", "Direccion", "Codigo_Postal", "Ciudad", "Provincia", telefono, "Instrumentos",
7    VALUES (2,'1E','Ivan','Calle Mayor',25391,'Guadalajara','Castilla la mancha',666699656,'Guitarra',2);
8  INSERT INTO public."Discos"(
9    "Codigo_disco", "Titulo", "Fecha_edicion", "Genero", "Formato", "Codigo_grupo_grupo")
10   VALUES (2,'VillancicosQ2','1975-2-19','rock','CD',2);
11  ROLLBACK;

```

Data Output Explain Messages Query History

ROLLBACK

Query returned successfully in 103 msec.



Para comprobar los datos vamos a hacer los SELECT de antes con las dos sesiones que tenemos para los usuarios postgres y admin.

musicos on admin@PostgreSQL 10

1

2

```
SELECT "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_w
FROM public."Grupo";
```

Data Output

Explain

Messages

Query History

	Codigo_grupo integer	Nombre text	Genero_musical text	Pais text	Sitio_web text
1	0	TryH0	Jazz	Espa...	www.TryH0...
2	1	SingY1	Pop	Espa...	www.SingY...



musicos on postgres@PostgreSQL 10

1

2

```
SELECT "Codigo_disco", "Titulo", "Fecha_edicion", "Genero", "Formato", "Cc
FROM public."Discos";
```

Data Output

Explain

Messages

Notifications

Query History

	Codigo_disco integer	Titulo text	Fecha_edicion date	Genero text	Formato text	Codigo_grupo_Grupo integer
1	1	Villanci...	1966-03-20	jazz	Vinilo	1

El resultado que hemos obtenido es que ningún dato de los que había en la base de datos ha sido modificado.

Esto se debe a que ROLLBACK sirve para deshacer los cambios que se han hecho en una transacción, en este caso ha sido deshacer las inserciones por lo que el resultado ha sido lo que se esperaba: que las tablas no fuesen modificadas. Solo se modificó la memoria local, y en este caso no se bloqueó ninguna transacción, sino que fue deshecha.

En definitiva, el aplicar ROLLBACK sirve para deshacer todos los cambios que una transacción ha hecho hasta dejarlo en el estado inicial.



**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?

La query para realizar lo que se pide en el enunciado es la siguiente:

```
musicos on admin@PostgreSQL 10
1 BEGIN;
2 INSERT INTO public."Musicos" (
3     codigo_musico, "DNI", "Nombre", "Direccion", "Codigo_Postal", "Ciudad", "Provincia", telefono, "Instrumentos", "Codigo_grupo_Grupo")
4     VALUES (2, '5643234', 'Ivan', 'Calle Mayor', 25391, 'Guadalajara', 'Castilla la mancha', 666699656, 'Guitarra', 1);
5 INSERT INTO public."Musicos" (
6     codigo_musico, "DNI", "Nombre", "Direccion", "Codigo_Postal", "Ciudad", "Provincia", telefono, "Instrumentos", "Codigo_grupo_Grupo")
7     VALUES (3, '4578345', 'Tomas', 'Calle Cervantes', 10985, 'Guadalajara', 'Andalucia', 658665698, 'Guitarra', 1);
8 UPDATE public."Musicos"
9     SET "DNI" = '5643234'
10    WHERE "DNI" = '4578345';
11 COMMIT;
```

Data Output Explain Messages Query History

ERROR: llave duplicada viola restricción de unicidad «Unique\_DNI»  
DETAIL: Ya existe la llave ("DNI")=(5643234 ).  
SQL state: 23505

El estado de la base de datos es el mismo que el inicial, es decir, no se ha añadido ni modificado ningún dato. Esto se debe a que hemos intentado añadir un atributo UNIQUE que ya estaba en la memoria local, que era el DNI que habíamos insertado previamente. Al no verificar que sea UNIQUE se produce un error y por tanto no se realiza ningún cambio en la base de datos y abortará.

Para volver a ganar el control es necesario realizar un ROLLBACK para deshacer por completo la transacción que abortó.

Por tanto, al producirse un error, al ser DNI UNIQUE se aborta la transacción sin provocar cambio alguno en la base de datos, pero obligando a realizar un ROLLBACK para recuperarse.

**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.

Para cerrar las sesiones, nos hemos desconectado del servidor y después hemos entrado con un usuario. En este caso se ve en la captura que solo queda la del usuario actual, admin, en MUSICOS y la de postgres pero esta sesión está para gestionar la base de postgres.

musicos on admin@PostgreSQL 10

```

1 SELECT *
2 FROM pg_stat_activity;

```

	datid oid	datname name	pid integer	usesysid oid	username name	application_name text	client_addr inet	client_hostname text	client_port integer
1		[null]	5208		[null]		[null]	[null]	[null]
2		[null]	5248	10	postgres		[null]	[null]	[null]
3	12938	postgres	5816	10	postgres		[null]	[null]	[null]
4	12938	postgres	3752	79678	admin	pgAdmin 4 - DB:postg...	192.168.1.37	[null]	62844
5	79741	musicos	5572	79678	admin	pgAdmin 4 - DB:musi...	192.168.1.37	[null]	63020
6	79741	musicos	3672	79678	admin	pgAdmin 4 - CONN:48...	192.168.1.37	[null]	63946
7	79741	musicos	10368	79678	admin	pgAdmin 4 - CONN:52...	192.168.1.37	[null]	64663
8		[null]	5176		[null]		[null]	[null]	[null]
9		[null]	5168		[null]		[null]	[null]	[null]
10		[null]	5184		[null]		[null]	[null]	[null]

Después de esto, realizamos la consulta:

musicos on admin@PostgreSQL 10

```

1 INSERT INTO public."Grupo"(
2     "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_web")
3 VALUES (20110,'NiceK5','Soul','España','www.NiceK5.com');
4 INSERT INTO public."Discos"(
5     "Codigo_disco", "Titulo", "Fecha_edicion", "Genero", "Formato", "Codigo_grupo_Grupo")
6 VALUES (13560,'CantantesM5','1925-7-2','rock&roll','Vinilo',20110);

```

INSERT 0 1

Query returned successfully in 100 msec.

**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?

Una vez conectados podemos comprobar como ahora tenemos dos sesiones activas, una para cada usuario.

musicos on admin@PostgreSQL 10

1  
2

SELECT \*  
FROM pg\_stat\_activity;

Data Output

Explain

Messages

Query History

	datid oid	datname name	pid integer	usesysid oid	username name	application_name text	client_addr inet	client_hostname text	client_port integer	
1		[null]	5208		[null]		[null]	[null]	[null]	
2		[null]	5248	10	postgres		[null]	[null]	[null]	
3	12938	postgres	5816	10	postgres		[null]	[null]	[null]	
4	12938	postgres	3752	79678	admin	pgAdmin 4 - DB:postgres	192.168.1.37	[null]	62844	
5	79741	musicos	5572	79678	admin	pgAdmin 4 - DB:musicos	192.168.1.37	[null]	63020	
6	79741	musicos	3672	79678	admin	pgAdmin 4 - CONN:48661	192.168.1.37	[null]	63946	
7	79741	musicos	10368	79678	admin	pgAdmin 4 - CONN:5242777	192.168.1.37	[null]	64663	
8	12938	postgres	8420	10	postgres	pgAdmin 4 - DB:postgres	[null]	[null]	[null]	
9	79741	musicos	5056	10	postgres	pgAdmin 4 - DB:musicos	[null]	[null]	[null]	
10		[null]	5176		[null]		[null]	[null]	[null]	
11		[null]	5168		[null]		[null]	[null]	[null]	
12		[null]	5184		[null]		[null]	[null]	[null]	

La consulta a realizar será:


**musicos on postgres@PostgreSQL 10**

1 **BEGIN;**  
2 **UPDATE public."Grupo"**  
3 **SET "Nombre"='La guardia'**  
4 **WHERE "Codigo\_grupo"=20110;**

**Data Output** Explain Messages Notifications Query History

UPDATE 1

Query returned successfully in 59 msec.

Una vez hecha la consulta para ver la actividad, nos vamos a las estadísticas de la base de datos, consultando pg\_stat\_activity. Solamente cogeremos los datos que nos interesen, en este caso son los que se muestran en la consulta:

musicos on postgres@PostgreSQL 10

1

SELECT datname,usesysid,client\_addr,state,query FROM pg\_stat\_activity

Data Output

Explain

Messages

Notifications

Query History

	datname	usesysid	client_addr	state	query
	name	oid	inet	text	text
4	postgres	79678	192.168.1.37	idle	SELECT
5	musicos	79678	192.168.1.37	idle	SELECT oid, format_type(oid,null) as typname FROM ...
6	musicos	79678	192.168.1.37	idle	INSERT INTO public."Grupo"({
7	musicos	79678	192.168.1.37	idle	SELECT *
8	postgres	10	192.168.1.40	idle	SELECT
9	musicos	10	192.168.1.40	idle	SELECT oid, format_type(oid, NULL) AS typname FRO...
10	musicos	10	192.168.1.40	idle	
11	musicos	10	192.168.1.40	idle in transaction	BEGIN;
12	musicos	10	192.168.1.40	active	SELECT datname,usesysid,client_addr,state,query FR...

De esta imagen podemos destacar los dos usuarios que hay, que emplean distintas IP al estar en distintos ordenadores, y que la transacción se encuentra en el estado idle\_transaction que significa que se encuentra esperando dentro de un BEGIN, como bien indica la query.

La información que se guarda en la base de datos, al igual que en los casos anteriores, es la misma que la inicial porque no se ha hecho COMMIT, aunque para la transacción sí que estará actualizada en memoria local.



musicos on postgres@PostgreSQL 10

1

```
SELECT * FROM "Grupo";
```

Data Output

Explain

Messages

Notifications

Query History

	Codigo_grupo integer	Nombre text	Genero_musical text	Pais text	Sitio_web text
1	0	TryH0	Jazz	España	www.TryH0...
2	1	SingY1	Pop	España	www.SingY...
3	20110	NiceK5	Soul	España	www.NiceK...

**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?

La consulta que hemos realizado es la siguiente:

musicos on admin@PostgreSQL 10

1

BEGIN;

2

UPDATE public."Discos"

3

SET "Genero"='Rock'

4

WHERE "Codigo\_disco"=13560;

Data Output

Explain

Messages

Query History

UPDATE 1

Query returned successfully in 64 msec.

La actividad que tiene la base de datos, obtenida del mismo modo que en la cuestión anterior, es esta:

musicos on admin@PostgreSQL 10

1

SELECT datname, username, client\_addr, query, state

2

FROM pg\_stat\_activity;

	datname	username	client_addr	query	state
	name	name	inet	text	text
1	[null]	[null]	[null]	<insufficient privilege>	[null]
2	[null]	postgres	[null]	<insufficient privilege>	[null]
3	postgres	postgres	[null]	<insufficient privilege>	[null]
4	postgres	admin	192.168.1.37	SELECT	idle
5	musicos	admin	192.168.1.37	SELECT oid, format_type(oid,null) as typna...	idle
6	musicos	admin	192.168.1.37	BEGIN;	idle in transaction
7	musicos	admin	192.168.1.37	SELECT datname, username, client_addr, q...	active
8	postgres	postgres	[null]	<insufficient privilege>	[null]
9	musicos	postgres	[null]	<insufficient privilege>	[null]
10	musicos	postgres	[null]	<insufficient privilege>	[null]
11	musicos	postgres	[null]	<insufficient privilege>	[null]
12	musicos	postgres	[null]	<insufficient privilege>	[null]
13	[null]	[null]	[null]	<insufficient privilege>	[null]
14	[null]	[null]	[null]	<insufficient privilege>	[null]
15	[null]	[null]	[null]	<insufficient privilege>	[null]

En este caso, al ser un usuario normal que no es superuser, no tiene permiso para ver todas las queries que hay en ejecución, pero podemos ver las que ha creado al realizarlo. En este caso otra más con BEGIN y por tanto en idle in transaction.

Al igual que antes, no se ha guardado nada en la base de datos, está todo en memoria local porque no ha habido COMMIT y no se han comprometido los datos:

musicos on admin@PostgreSQL 10

```
1 SELECT * FROM "Discos"
```

	Codigo_disco integer	Titulo text	Fecha_edicion date	Genero text	Formato text	Codigo_grupo_Grupo integer
1	1	VillancicosJ0	1966-03-20	jazz	Vinilo	1
2	13560	CantantesM5	1925-07-02	rock&roll	Vinilo	20110

**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?

La query que se realiza es la siguiente:

Statistics Dependencies Dependents Query - musicos ... Update script \* Update script \*

musicos on postgres@PostgreSQL 10

```
1 UPDATE public."Discos"  
2     SET "Fecha_edicion"='11-1-2019'  
3     WHERE "Codigo_disco"=13560;  
4  
5
```

Data Output Explain Messages Notifications Query History

Waiting for the query execution to complete...

En este caso surge un problema: la consulta se queda en espera por la tabla de discos, ya que quiere modificarla y la está usando T1. Esto se puede ver en la siguiente captura en la que el estado es Lock, es decir, está bloqueada porque requiere el lock de la tabla. También se ve que es un transactionid y que se encuentra activo, o sea, que está bloqueado. El resto de la actividad sigue prácticamente igual:

musicos on postgres@PostgreSQL 10

1

SELECT datname,username,client\_addr,wait\_event\_type,wait\_event,state,query FROM pg\_stat\_activity

Data Output

[Explain](#)
[Messages](#)
[Notifications](#)
[Query History](#)

	datname name	username name	client_addr inet	wait_event_type text	wait_event text	state text	query text
3	postgres	postgres	::1	Client	ClientRead	idle	SELECT J.jobid FROM pgagent.pga_j...
4	postgres	admin	192.168.1.37	Client	ClientRead	idle	SELECT
5	musicos	admin	192.168.1.37	Client	ClientRead	idle	SELECT oid, format_type(oid,null) as ...
6	musicos	admin	192.168.1.37	Client	ClientRead	idle in transaction	BEGIN;
7	musicos	admin	192.168.1.37	Client	ClientRead	idle	SELECT * FROM "Discos"
8	postgres	postgres	192.168.1.40	Client	ClientRead	idle	
9	musicos	postgres	192.168.1.40	Client	ClientRead	idle	SELECT oid, format_type(oid, NULL) ...
10	musicos	postgres	192.168.1.40	Client	ClientRead	idle	
11	musicos	postgres	192.168.1.40	Lock	transactionid	active	UPDATE public."Discos"
12	musicos	admin	192.168.1.37	Client	ClientRead	idle	SELECT * FROM pg_stat_activity
13	musicos	postgres	192.168.1.40	[null]	[null]	active	SELECT datname,username,client_ad...

En la base de datos, al igual que en todos los casos anteriores, se encuentra la información inicial como se ve en estas capturas:



musicos on postgres@PostgreSQL 10

1

**SELECT \* FROM "Grupo"**

Data Output

Explain

Messages

Notifications

Query History

	Codigo_grupo integer	Nombre text	Genero_musical text	Pais text	Sitio_web text
1	0	TryH0	Jazz	Espa...	www.TryH0...
2	1	SingY1	Pop	Espa...	www.SingY...
3	20110	NiceK5	Soul	Espa...	www.NiceK...



musicos on postgres@PostgreSQL 10

1

SELECT \* FROM "Discos"

Data Output

[Explain](#)

[Messages](#)

[Notifications](#)

[Query History](#)

	Codigo_disco integer	Titulo text	Fecha_edicion date	Genero text	Formato text	Codigo_grupo_Grupo integer
1	1	VillancicosJ0	1966-03-20	jazz	Vinilo	1
2	13560	CantantesM5	1925-07-02	rock&roll	Vinilo	20110

**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](http://www.laguardia.es). ¿Qué actividad hay registrada en la base de datos? ¿Cuál es la información guardada en la base de datos? ¿Por qué?

Cuando realizamos la consulta salta un error porque se ha producido un deadlock, esto se debe a que hay un ciclo de espera de permisos de bloqueo. Para solucionarlo PostgreSQL mata una transacción como se ve en el mensaje de la captura:

```
musicos on admin@PostgreSQL 10
1 BEGIN;
2 UPDATE public."Grupo"
3     SET "Sitio_web"='www.laguardia.es'
4     WHERE "Codigo_grupo"=20110;
```

Data Output Explain Messages Query History

ERROR: se ha detectado un deadlock  
DETAIL: El proceso 3672 espera ShareLock en transacción 2056; bloqueado por proceso 10608.  
El proceso 10608 espera ShareLock en transacción 2054; bloqueado por proceso 3672.  
HINT: Vea el registro del servidor para obtener detalles de las consultas.  
CONTEXT: mientras se actualizaba la tupla (0,13) en la relación «Grupo»  
SQL state: 40P01

Para verlo con más detalle accedemos al log del servidor, en el que podemos ver qué transacción ha sido matada y cuál ha sido elegida para continuar. En nuestro caso se ha elegido la transacción T2, matando así a T1. También finaliza T2 como se puede ver en la segunda imagen.

```
2019-01-11 02:31:53.958 CET [3672] ERROR: se ha detectado un deadlock
2019-01-11 02:31:53.958 CET [3672] DETALLE: El proceso 3672 espera ShareLock en transacción 2056; bloqueado por proceso 10608.
El proceso 10608 espera ShareLock en transacción 2054; bloqueado por proceso 3672.
Proceso 3672: BEGIN;
UPDATE public."Grupo"
    SET "Sitio_web"='www.laguardia.es'
    WHERE "Codigo_grupo"=20110;
Proceso 10608: UPDATE public."Discos"
    SET "Fecha_edicion"='11-1-2019'
    WHERE "Codigo_disco"=13560;

2019-01-11 02:31:53.958 CET [3672] HINT: Vea el registro del servidor para obtener detalles de las consultas.
2019-01-11 02:31:53.958 CET [3672] CONTEXTO: mientras se actualizaba la tupla (0,13) en la relación «Grupo»
2019-01-11 02:31:53.958 CET [3672] SENTENCIA: BEGIN;
UPDATE public."Grupo"
    SET "Sitio_web"='www.laguardia.es'
    WHERE "Codigo_grupo"=20110;
2019-01-11 02:31:53.958 CET [10608] LOG: duración: 561821.920 ms
```



- **FINALIZACIÓN DE T2:**

```

musicos on postgres@PostgreSQL 10
1  UPDATE public."Discos"
2      SET "Fecha_edicion"='11-1-2019'
3      WHERE "Codigo_disco"=13560;
4
5
Data Output Explain Messages Notifications Query History
UPDATE 1

Query returned successfully in 9 min 21 secs.

```

En la actividad de la base de datos se encuentra lo mismo que antes con dos BEGIN, aunque su contenido hay cambiado al hacer el COMMIT; ya que, al abortar uno, este tendrá que deshacer todo.

En la base de datos sigue sin guardarse información porque al igual que antes no hay COMMIT y por tanto los datos de la transacción que sigue viva están en memoria local.

**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 comprometer cada transacción con COMMIT obtenemos un resultado diferente. En la que ha sido detenida obtenemos un ROLLBACK. Esto se debe a que, al ser eliminada, todos los cambios que tenga hechos se deshacen, por lo tanto, lo deja como estaba. En el caso de T2 podemos ver como se realiza el COMMIT con éxito por lo que los datos se comprometen, siendo indicativo de que los datos han pasado de la memoria local a la compartida. Estos datos ahora sí están disponibles para todos los usuarios.

```

musicos on admin@PostgreSQL 10
1  COMMIT;

Data Output Explain Messages Query History
ROLLBACK

Query returned successfully in 62 msec.

```

**T1**

```

musicos on postgres@PostgreSQL 10
1  COMMIT;
2
3
4

Data Output Explain Messages Notifications
COMMIT

Query returned successfully in 61 msec.

```

**T2**

El valor final de las actividades son los iniciales con los cambios de T2, que sí que han sido comprometidos correctamente como se puede ver en las siguientes capturas:

musicos on admin@PostgreSQL 10

1 SELECT \* FROM "Discos"

Data Output Explain Messages Query History

	Codigo_disco integer	Titulo text	Fecha_edicion date	Genero text	Formato text	Codigo_grupo_Grupo integer
1	1	VillancicosJ0	1966-03-20	jazz	Vinilo	1
2	13560	CantantesM5	2019-01-11	rock&roll	Vinilo	20110

musicos on admin@PostgreSQL 10

1 SELECT \* FROM "Grupo"

Data Output Explain Messages Query History

	Codigo_grupo integer	Nombre text	Genero_musical text	Pais text	Sitio_web text
1	0	TryH0	Jazz	Espa...	www.TryH0...
2	1	SingY1	Pop	Espa...	www.SingY...
3	20110	La guardia	Soul	Espa...	www.NiceK...

Esto se debe a que, como consecuencia del deadlock producido, se ha tenido que elegir una operación para que se realice y no tener una espera infinita. En este caso se ha elegido a T2 para que continúe, mientras que T1 deshacerá todos los cambios para que pueda a volver a hacer la consulta si quiere, aunque ya sería sin este deadlock.

**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.

Realizamos primero la inserción de forma normal:

```
musicos on admin@PostgreSQL 10
1  INSERT INTO public."Grupo"(
2      "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_web")
3      VALUES (50800, 'CoolR8', 'Rock', 'España', 'www.CoolR8.com');
4
5
```

Data Output Explain Messages Query History

INSERT 0 1

Query returned successfully in 102 msec.

Abrimos la nueva transacción sin cerrarla:

```
musicos on admin@PostgreSQL 10
1  BEGIN;
2  UPDATE public."Grupo"
3      SET "Genero_musical"='Pop'
4      WHERE "Codigo_grupo"=50800;
```

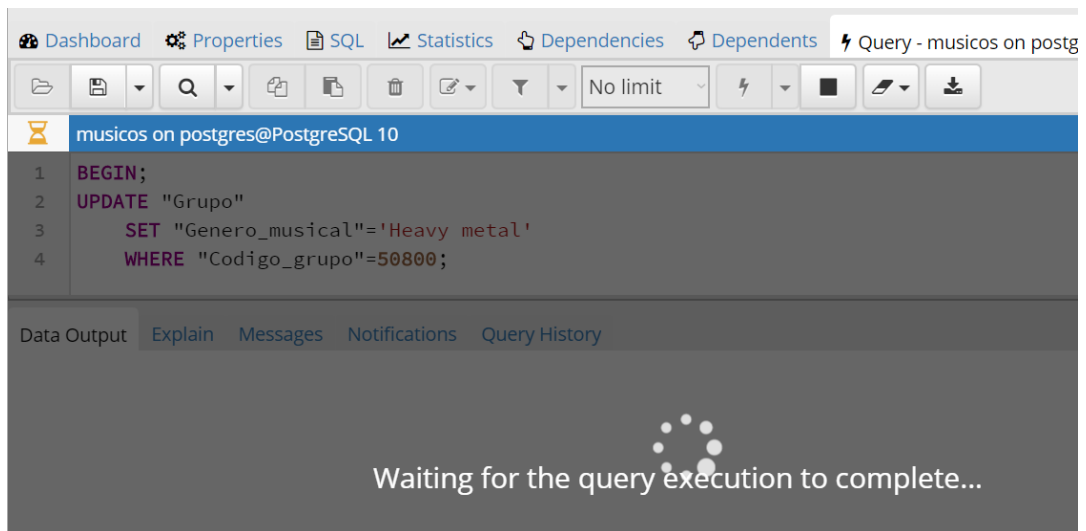
Data Output Explain Messages Query History

UPDATE 1

Query returned successfully in 119 msec.

**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?

Lo primero es abrir la nueva sesión con otro usuario y realizar la consulta:



Lo que ocurre es que la transacción se queda esperando para ejecutarse, al no acabar la otra transacción. Esto se debe a que T1 ha tomado con un lock la tabla, de tal forma que, cuando T2 quiere usarla, no puede porque ya tiene un lock. Por lo que a T2 no le queda otra que quedarse en espera hasta que acabe la transacción T1.

La información que podemos obtener de las dos transacciones en el sistema es la siguiente:

musicos on postgres@PostgreSQL 10

```
1 SELECT DISTINCT datname,username,client_addr, query,wait_event, wait_event_type,state FROM pg_stat_statements
```

Data Output

[Explain](#)
[Messages](#)
[Notifications](#)
[Query History](#)

	datname name	username name	client_addr inet	query text	wait_event text	wait_event_type text	state text
1	[null]	[null]	[null]		CheckpointerMain	Activity	[null]
2	musicos	postgres	192.168.1.40	BEGIN;	transactionid	Lock	active
3	musicos	postgres	192.168.1.40	SELECT oid, for...	ClientRead	Client	idle
4	[null]	[null]	[null]		AutoVacuumMain	Activity	[null]
5	[null]	postgres	[null]		LogicalLauncherMain	Activity	[null]
6	musicos	postgres	192.168.1.40	SELECT DISTIN...	[null]	[null]	active
7	postgres	postgres	192.168.1.40	/*pga4dash*/	ClientRead	Client	idle
8	musicos	admin	192.168.1.37	SELECT	ClientRead	Client	idle
9	postgres	postgres	::1	SELECT l.jobid ...	ClientRead	Client	idle
10	musicos	admin	192.168.1.37	BEGIN;	ClientRead	Client	idle in transaction
11	[null]	[null]	[null]		BgWriterHibernate	Activity	[null]

En esta imagen podemos ver que se encuentra T2 en Lock, es decir, está bloqueada como indica con active. Mientras que T1 se encuentra en estado idle on transaction, que indica que todavía sigue dentro del BEGIN porque no se ha hecho COMMIT.

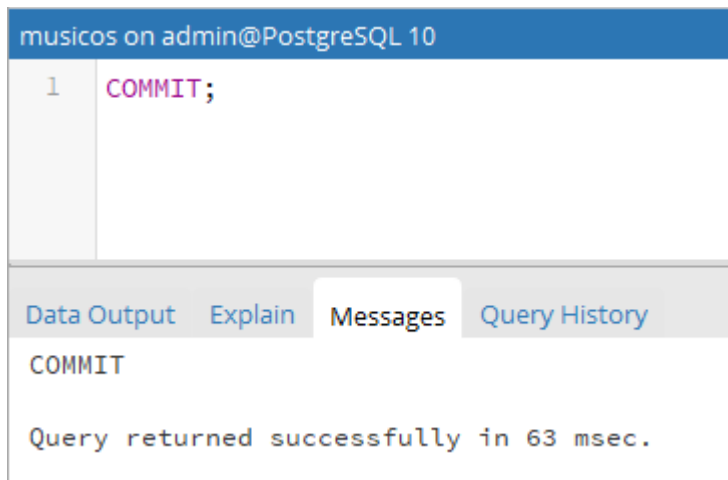
La razón de esto se debe al sistema de bloqueos de PostgreSQL, porque ambas transacciones requerían los mismos recursos: primero un Lock para la lectura y después un Lock que se subía de nivel para la escritura al actualizar. Pero como no ha acabado ninguna de las dos, al querer acceder T2, se ha visto obligada a esperar a que T1 acabase para así poder leer los datos para saber dónde actualizar y después realizar la escritura; pero sus planes se han visto

aplazados porque T1 simplemente no había acabado la transacción y por tanto liberado el Lock.

En definitiva, T2 se ha visto obligada a esperar porque T1 estaba haciendo uso del lock de la tabla para escritura y no lo había liberado al no haber hecho COMMIT.

**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?

Lo primero es comprometer la transacción T1:

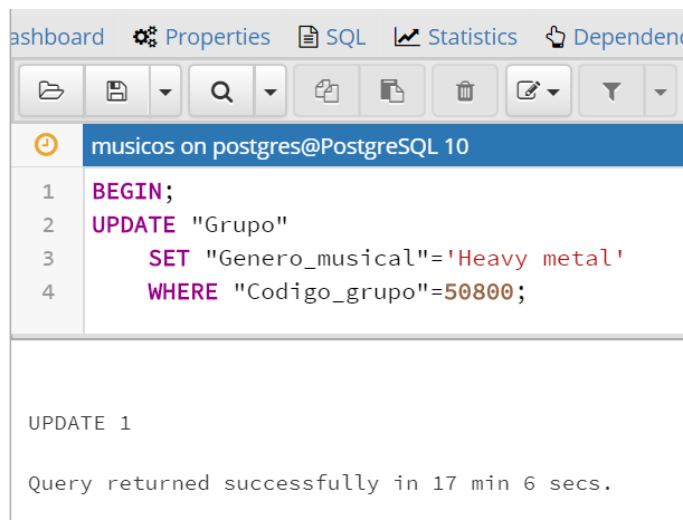


```
musicos on admin@PostgreSQL 10
1 COMMIT;

Data Output Explain Messages Query History
COMMIT

Query returned successfully in 63 msec.
```

Nos devuelve COMMIT, por lo que se ha ejecutado con éxito y los datos deben de haberse comprometido, dando fin a esta transacción. Esto implica que ahora T2 tiene vía libre para continuar con su ejecución. Para comprobar esto vamos a la otra sesión y vemos que se ha completado tras 17 minutos, todo el tiempo que ha estado esperando:



```
dashboard Properties SQL Statistics Dependence
musicos on postgres@PostgreSQL 10
1 BEGIN;
2 UPDATE "Grupo"
3     SET "Genero_musical"='Heavy metal'
4     WHERE "Codigo_grupo"=50800;


UPDATE 1

Query returned successfully in 17 min 6 secs.
```

Todo esto tiene sentido porque al hacer COMMIT se da por finalizada la transacción, liberando el lock que tenía antes. Este COMMIT pasará los datos de la memoria local a la base de datos. Al liberarse el lock, T2 podrá continuar con su ejecución volviendo a activar el lock para realizar su actualización.

En cuanto al estado final de la información, se han aplicado los cambios de T1, ya que se ha hecho COMMIT en ella. Así, el género musical del último grupo será Pop en la memoria

global. En la memoria local de U2 tendremos que ese género musical es Heavy metal, pero, como no se ha comprometido, sus cambios no se han guardado en memoria global aún. De este modo, si consultamos los datos de U2 tendremos Pop en el género musical, ya que es el dato que ha pasado a la memoria global como hemos indicado anteriormente.



musicos on postgres@PostgreSQL 10

1

SELECT \* FROM "Grupo"

Data Output

Explain

Messages

Notifications

Query History

	Codigo_grupo integer	Nombre text	Genero_musical text	Pais text	Sitio_web text
1	0	TryH0	Jazz	España	www.TryH0.com
2	1	SingY1	Pop	España	www.SingY1.com
3	20110	La guardia	Soul	España	www.NiceK5.com
4	50800	CoolR8	Pop	España	www.CoolR8.com

**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?

Repetimos los pasos de arriba realizando COMMIT en T2:

musicos on postgres@PostgreSQL 10	
1	<b>COMMIT;</b>
Data Output Explain Messages Notifications Query History	
COMMIT	
Query returned successfully in 47 msec.	

En este caso, al igual que antes, devuelve el mensaje de que se ha realizado con éxito por lo que se habrán actualizado los datos. Ahora no había ninguna operación que esperara a T2 por lo que el lock ya queda libre del todo para cualquier futura transacción. Todo esto ocurre porque T2 ha realizado COMMIT de los datos y, a diferencia de otros casos, al no haber abortado ni haber tenido errores, no ha sido necesario ningún ROLLBACK. Simplemente se ha seguido la ejecución de las transacciones en orden teniendo que esperar sin entrar en ningún conflicto de deadlock.

Finalmente comprobamos los datos para ver si son correctos:

musicos on admin@PostgreSQL 10

1

2

SELECT "Codigo\_grupo", "Nombre", "Genero\_musical", "Pais", "Sitio\_web"

FROM public."Grupo";

Data Output

Explain

Messages

Query History

	Codigo_grupo integer	Nombre text	Genero_musical text	Pais text	Sitio_web text
1	0	TryH0	Jazz	España	www.TryH0.com
2	1	SingY1	Pop	España	www.SingY1.com
3	20110	La guardia	Soul	España	www.NiceK5.com
4	50800	CoolR8	Heavy metal	España	www.CoolR8.com

Efectivamente, los datos que se muestran son correctos, siendo el género musical Heavy metal. La razón de esto es que los datos de la última transacción han sido comprometidos con éxito.

**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.

Una vez cerradas todas las sesiones, volvemos a abrir la sesión con un usuario de músicos para ejecutar la siguiente transacción:

musicos on admin@PostgreSQL 10	
1	<code>BEGIN;</code>
2	<code>UPDATE public."Discos"</code>
3	<code>SET "Codigo_disco"=23560</code>
4	<code>WHERE "Codigo_disco"=13560;</code>
Data Output Explain Messages Query History	
UPDATE 1	
Query returned successfully in 101 msec.	

Abrimos con otro usuario otra transacción realizando el cambio que se nos pide sin cerrarla:

musicos on postgres@PostgreSQL 10	
1	<code>BEGIN;</code>
2	<code>UPDATE public."Discos"</code>
3	<code>SET "Formato"='MP3'</code>
4	<code>WHERE "Codigo_disco"=13560;</code>
Data Output Explain Messages Notifications Query History	
Waiting for the query execution to complete	



**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?

Comprometemos la transacción T1:

```

musicos on admin@PostgreSQL 10
1 COMMIT;

Data Output Explain Messages Query History
COMMIT

Query returned successfully in 62 msec.

```

Lo que ocurre es lo mismo que en el caso anterior. Se hace COMMIT correctamente y los datos se modifican en la base de datos. Además, en T2 se realiza el UPDATE, pero el mensaje que muestra es que se han actualizado 0 registros. Esto se debe a que al hacer COMMIT los datos que estaban en memoria local pasan a la base de datos y es ahí donde T2, al hacer la transacción, intenta buscar el número de antes para realizar la actualización de los datos, pero no lo encuentra, por eso actualiza cero:

```

musicos on postgres@PostgreSQL 10
1 BEGIN;
2 UPDATE public."Discos"
3     SET "Formato"='MP3'
4     WHERE "Codigo_disco"=13560;

Data Output Explain Messages Notifications Query History
UPDATE 0

Query returned successfully in 1 min 20 secs.

```

La información que hay en la tabla de la base de datos es la siguiente:

musicos on admin@PostgreSQL 10

1  
2

```
SELECT "Codigo_disco", "Titulo", "Fecha_edicion", "Genero", "Formato", "Codigo_grupo_Grupo"
FROM public."Discos";
```

Data Output

Explain

Messages

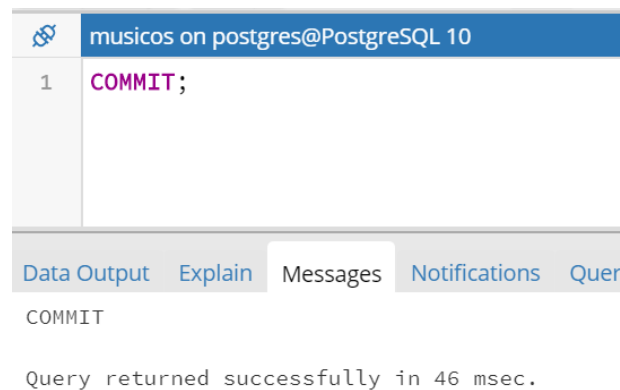
Query History

	Codigo_disco integer	Titulo text	Fecha_edicion date	Genero text	Formato text	Codigo_grupo_Grupo integer	
1	1	VillancicosJ0	1966-03-20	jazz	Vinilo	1	
2	23560	CantantesM5	2019-01-11	rock&roll	Vinilo	20110	

Esto indica que sí se ha cambiado correctamente el número de disco, pero que no se ha actualizado el formato a MP3. Esto se debe a lo que hemos explicado en el párrafo anterior: al cambiar los datos con el COMMIT, cuando T2 quiere leer los datos para actualizar, se encuentra que no hay ningún código 13560, sino que está el actualizado y por ende no actualiza ningún registro.

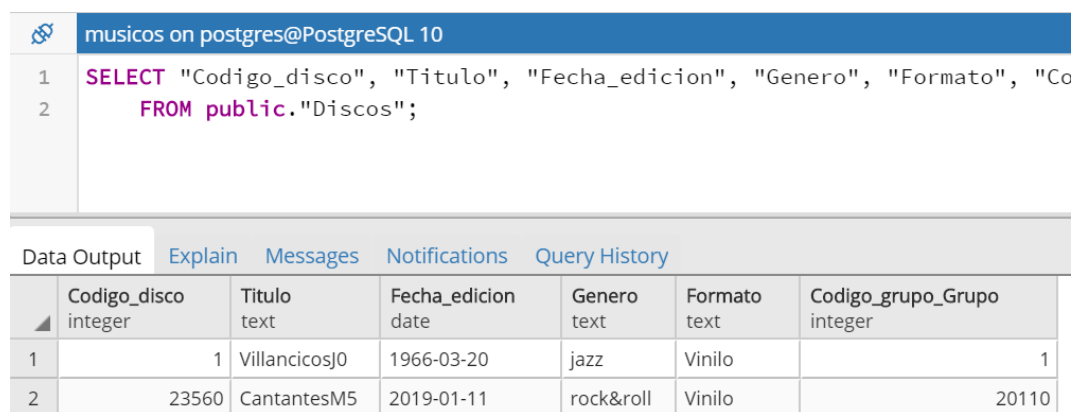
**Cuestión 2.28:** Comprometa la transacción T2, ¿Qué es lo que ocurre? ¿Por qué? ¿Cuál es el estado final de la información del disco con código 13560 para ambos usuarios? ¿Por qué?

Realizamos el COMMIT y vemos que se realiza correctamente. Se ha realizado correctamente porque no ha habido ningún error en la consulta ni tampoco se ha producido un conflicto, como un deadlock, con otra transacción:



The screenshot shows a PostgreSQL query window titled "musicos on postgres@PostgreSQL 10". The query entered is "COMMIT;". The output shows "COMMIT" and "Query returned successfully in 46 msec.".

El estado final de la información para el disco es el siguiente:



The screenshot shows a PostgreSQL query window titled "musicos on postgres@PostgreSQL 10". The query entered is "SELECT 'Codigo\_disco', 'Titulo', 'Fecha\_edicion', 'Genero', 'Formato', 'Codigo\_grupo\_grupo' FROM public.\"Discos\";". The output is a table with 7 columns: Codigo\_disco (integer), Titulo (text), Fecha\_edicion (date), Genero (text), Formato (text), and Codigo\_grupo\_grupo (integer). The table contains two rows of data.

	Codigo_disco integer	Titulo text	Fecha_edicion date	Genero text	Formato text	Codigo_grupo_grupo integer
1	1	VillancicosJ0	1966-03-20	jazz	Vinilo	1
2	23560	CantantesM5	2019-01-11	rock&roll	Vinilo	20110

Los datos permanecen igual que en el resultado del COMMIT de T1 porque, como hemos dicho, el UPDATE no actualiza ningún valor de la base de datos, ya que simplemente no lo encuentra al tener un número de disco distinto al de la condición.

**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?

Lo que ocurre es que PostgreSQL muestra un mensaje de Warning indicando que ya hay una transacción en curso, al ser un aviso y no un error la transacción puede seguir siendo ejecutada, aunque como un solo bloque.

```
musicos on admin@PostgreSQL 10
1 BEGIN;
2 UPDATE public."Grupo"
3   SET "Nombre"='Grupo'
4   WHERE "Codigo_grupo"=50800;
5
```

Data Output Explain Messages Query History

UPDATE 1

Query returned successfully in 57 msec.

```
musicos on admin@PostgreSQL 10
1 BEGIN;
2 UPDATE public."Grupo"
3   SET "País"='China'
4   WHERE "Codigo_grupo"=50800;
5
6
```

Data Output Explain Messages Query History

WARNING: ya hay una transacción en curso  
UPDATE 1

Query returned successfully in 55 msec.

```
musicos on admin@PostgreSQL 10
1 COMMIT;
2
```

Data Output Explain Messages Query History

COMMIT

Query returned successfully in 56 msec.

```
musicos on admin@PostgreSQL 10
1 COMMIT;
2
```

Data Output Explain Messages Query History

WARNING: no hay una transacción en curso  
COMMIT

Query returned successfully in 57 msec.

Esto se debe a que PostgreSQL no soporta las transacciones autónomas o nested BEGIN, actuando únicamente como bloques de transacciones. Esto puede solucionarse empleando SAVEPOINT, que sirven para guardar estados de la transacción dentro de una misma, permitiendo conseguir lo que se proponía.

Los datos que cambian en la base de datos son los siguientes:

PostgreSQL 10 - musicos - public.Grupo					
1 SELECT * FROM public."Grupo"					
2 ORDER BY "Codigo_grupo" ASC					
Data Output Explain Messages Query History					
	Codigo_grupo [PK] integer	Nombre text	Genero_musical text	País text	Sitio_web text
1	0	TryH0	Jazz	España	www.TryH0.com
2	1	SingY1	Pop	España	www.SingY1.com
3	20110	La guardia	Soul	España	www.NiceK5.com
4	50800	Grupo	Heavy metal	China	www.CoolR8.com

En nuestro caso, como hemos hecho COMMIT para verificar si había varias transacciones pero solo había una al final, los datos se han comprometido por lo que van a estar todos los cambios que hemos hecho en la base de datos.

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

Lo primero de todo es detener el servidor con el siguiente comando `pg_ctl -D "C:\Program Files\PostgreSQL\10\data" stop` que ofrece PostgreSQL para detenerlo:

```
Microsoft Windows [Versión 10.0.17134.523]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\WINDOWS\system32>pg_ctl -D "C:\Program Files\PostgreSQL\10\data" stop
esperando que el servidor se detenga.... listo
servidor detenido

C:\WINDOWS\system32>
```

Después, si tenemos suficiente memoria, hacemos una copia de todo el cluster de datos para tener un respaldo. Una vez tenemos la copia borramos todo el cluster de data.

Pasamos a restaurar el backup utilizando el siguiente comando

```
pg_restore -h localhost -p 5432 -U postgres -d musicos -v  
"C:\Backup\000000010000000100000041.00000028.backup"
```

Selecciónar Administrador: Símbolo del sistema - pg\_restore -h localhost -p 5432 -U postgres -d musicos -v

```
Microsoft Windows [Versión 10.0.17134.523]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\WINDOWS\system32>pg_restore -h localhost -p 5432 -U postgres -d musicos -v
"C:\Backup\000000010000000100000041.00000028.backup"
```

Como solo se quiere la instancia del punto 2.6 con restaurar el backup valdría, además habría que borrar los archivos de `pg_wal`. Si se quisiera una recuperación hasta los datos actuales podríamos hacerlo con los archivos WAL que haya del cluster data que hemos guardado. Una vez hecho eso sería configurar el `recovery.conf` para indicar como queremos la recuperación y el servidor se encargaría de hacerlo.

Por tanto, para recuperar la instancia del 2.6 tenemos que aplicar lo que hay aquí arriba.

**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.

El tipo de recuperación que tiene PostgreSQL es de tipo REDO, es decir, es de modificación diferida. Este método lo implementa a través del WAL que registra todas las transacciones que han sido completadas con COMMIT de tal forma que si hay una caída este podrá recuperar los datos empleando el WAL y una copia de seguridad para ir rehaciendo todas las operaciones hasta la última que se hizo COMMIT.

El protocolo de gestión de la concurrencia que tiene es del tipo de gestión de bloqueos, dentro de los cuales se identifica con el protocolo de bloqueo normal de niveles, de tal forma que solo hay un lock por transacción y no se elevan ni se bajan, sino que son adquiridos según el tipo de operación que se realice, estando ya todo predefinido. Para los conflictos de bloqueos se podría identificar con el método de Herir-Esperar, ya que provoca un error en la transacción que finaliza en ROLLBACK porque así PostgreSQL lo define, y como se puede comprobar cuando se hace COMMIT en la pregunta en la que surge deadlock. Todo esto lo hace con los diferentes mecanismos de los que dispone, que son MVCC, los Locks y los modos de aislamiento de transacciones.

PostgreSQL tiene un sistema de aislamiento de transacciones, que permite elegir el nivel que se quiera teniendo como máxima secuenciabilidad el nivel de Serializable Isolation que garantiza una secuenciabilidad de las transacciones, una detrás de otra como se indica en la documentación que hay. Además, hay más niveles que permitirán una mayor o menor secuenciabilidad y con ello la aparición de ciertos fenómenos como dirty read..

No hay ROLLBACK en cascada porque los datos solo se hacen comunes cuando hay COMMIT, en caso de que se produzca un fallo es el propio PostgreSQL quien inicia un ROLLBACK en la transacción, pero no se producirá en cascada porque los datos no se modifican hasta que haya un COMMIT y por tanto no afectará al resto de las transacciones. Esto se ha visto en los casos de transacciones que hemos hecho en la práctica.

Como no hay ROLLBACK en cascada toda planificación será recuperable como se ha visto en teoría

## **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.