Relaciones y operaciones transaccionales (Parte II)

Transacciones

Competencias

- Identificar qué son las transacciones y qué utilidad nos genera al momento de gestionar una base de datos.
- Integrar las transacciones en sentencias SQL para la realización por lote de consultas
- Cargar una base de datos utilizando dump.

Introducción

Hasta el momento hemos creado un par de tablas en nuestra base de datos, sus registros se han cargado de manera manual y a través del formato .csv, con los que realizamos distintos tipos de consultas para obtener información de una forma personalizada. Sin embargo, te has preguntado: ¿Qué pasa si al momento de realizar una consulta ya sea para recuperar información, crear nuevos registros, entre otros, surge algún inconveniente y obtenemos un error? La respuesta rápida y sencilla es que no se realiza el cambio o consulta que estamos procesando y en muchos casos eso puede ser un problema. Para darle una solución a este inconveniente surgen las transacciones en PostgreSQL, las cuales sirven como simuladores de consultas que nos permiten procesar sentencias SQL sin perjudicar de forma permanente la base de datos a menos que le indiquemos que así sea.

Las transacciones son usadas a menudo para hacer pruebas en una base de datos cuya información es de suma importancia y está enlazada a un producto que ya fue lanzado a producción, no obstante, tienen un uso aún más práctico relacionado a procedimientos que requieren de varias consultas y que se necesita que el 100% de estas sean realizadas con

éxito, de lo contrario no se realice nada para no perjudicar la consistencia de la base de datos.

La lógica de las transacciones así como lo indica su nombre son utilizadas en la programación de software en instituciones tan importantes como son los bancos, los cuales realizan sus transacciones bancarias basados en un todo o nada, pues no tendría sentido que se descuente el dinero de una cuenta A y no se sume a una cuenta B, por esto y mucho más, es importante que aprendas a usar las transacciones en SQL y es lo que verás en este capítulo.

Transacciones

Las transacciones son secuencias de instrucciones ordenadas, las cuáles pueden ser indicadas de forma manual o pueden ser aplicadas automáticamente. Estas realizan cambios en las bases de datos a la hora de aplicar comandos de manipulación de columnas y registros, su importancia recae en el control que nos ofrece sobre los cambios permanentes en la base de datos.

Estas transacciones tienen las siguientes propiedades:

- **Atomicidad:** Todas las operaciones realizadas en la transacción deben ser completadas. En el caso que ocurra un fallo, esta transacción es abortada y devuelve todo al estado previo a la transacción.

- **Consistencia:** La base de datos cambiará solamente cuando la transacción se haya realizado.

- **Aislamiento:** Las transacciones pueden ocurrir independientes una u otra.

- **Durabilidad:** El resultado de la transacción persiste a pesar de que el sistema falle.

Una transacción empaqueta varios pasos en una operación, de forma que se completen todos o ninguno, cuidando la integridad de la información, de la siguiente manera:

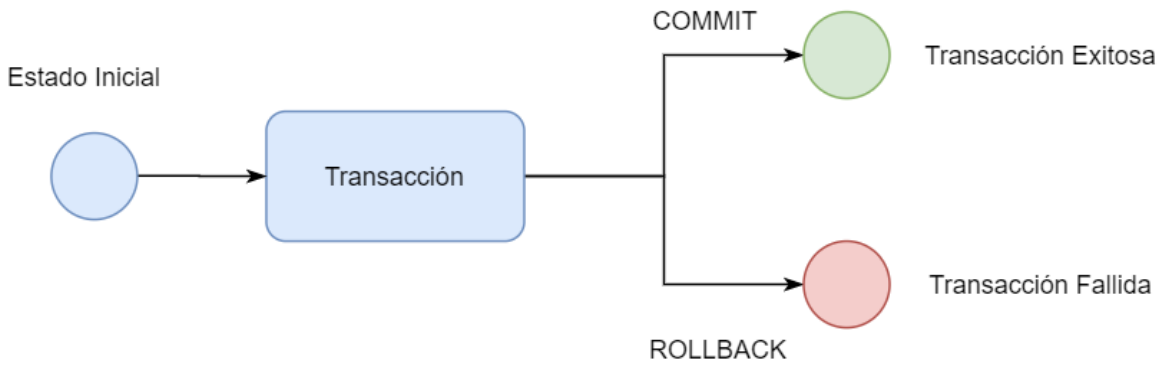

Imagen 1. Diagrama de flujo de las transacciones.

Automáticamente PostgreSQL está configurado para que se ejecuten estas transacciones sin manipular este flujo, sin embargo, es posible tener control de las transacciones. Para eso, existen los siguientes comandos:

Comando	Descripción
BEGIN ó START	El sistema permite que se ejecuten todas las sentencias SQL que necesitemos.
COMMIT	Guarda los cambios de la transacción
ROLLBACK	Retrocede los cambios realizados
SAVEPOINT	Guarda el punto de partida al cual volver a la hora de aplicar ROLLBACK
SET TRANSACTION	Le asigna nombre a la transacción

Tabla 1. Comandos usados en las transacciones

Estos comandos sólo pueden ser usados con las operaciones INSERT, UPDATE y DELETE, ya que son aquellos que manipulan toda la tabla y hacen este proceso automáticamente. La sintaxis de estos comandos es la siguiente:

- `COMMIT;`
- `SAVEPOINT nombre_savepoint;`
- `ROLLBACK [TO nombre_savepoint];`

Lo que está entre corchetes es de carácter opcional, por lo que podemos decirle con ROLLBACK a qué punto volver, este volverá al último punto guardado o por defecto al estado inicial de la transacción. En el siguiente código te muestro la sintaxis que puedes ocupar para iniciar una transacción.

```
SET TRANSACTION [READ ONLY|WRITE][NAME nombre_transaccion];
```

Como puedes notar, existen varias opciones, podemos usar READ ONLY para solamente leer la base de datos, READ WRITE para leer y escribir sobre ella, y poder nombrar la transacción con el comando NAME.

Un ejemplo muy sencillo de la utilidad de las transacciones, es pensar en el funcionamiento de las cuentas de los bancos. Al hacer una transferencia, por ejemplo, ¿Cómo nos aseguramos que el dinero que se resta de una cuenta, se sume en la siguiente? ¿Cómo controlamos que efectivamente se mantenga la integridad de los datos en caso de fallas?

Las transacciones nos permiten esto. Pensemos en la siguiente tabla:

```
CREATE TABLE cuentas (
numero_cuenta INT NOT NULL UNIQUE PRIMARY KEY,
balance DECIMAL CHECK (balance >= 0.00)
-- check valida la condición que el monto sea mayor a cero
);
```

Nota la palabra reservada “CHECK” escrita luego de definir el tipo de dato para la propiedad “balance”, esto setea una restricción a esa columna que devolverá un error cuando intente actualizar o ingresar un valor que no cumpla con la condición.

Ahora con el siguiente código procedemos con el registro de 2 cuentas con un saldo de $1000 cada una.

```
INSERT INTO cuentas (numero_cuenta, balance) VALUES (1,
1000);
INSERT INTO cuentas (numero_cuenta, balance) VALUES (2,
1000);
```

Obteniendo la siguiente tabla como resultado.

numero_cuenta	balance
1	1000
2	1000

Tabla 2. Modelo de la tabla ventas

Ahora si quisiéramos hacer una transferencia de $1000 desde nuestra cuenta 1 a la cuenta 2, una forma de asegurarnos que el monto de nuestro balance disminuya en $1000 y el de la segunda cuenta aumenta en la misma cifra, podría escribirse de la siguiente manera:

```
BEGIN TRANSACTION;
UPDATE cuentas SET balance = balance - 1000 WHERE numero_cuenta = 1;
UPDATE cuentas SET balance = balance + 1000 WHERE numero_cuenta = 2;
COMMIT;
```

Es importante que notes que se está iniciado por un "BEGIN TRANSACTION", esto permite definir el bloque de código que queremos controlar en la transacción y con el “COMMIT” damos por finalizada la transacción para proceder con los cambios permanentes en la base de datos.

Entendiendo lo anterior, la actualización en ambas cuentas sólo se realizará si las dos transacciones son correctas, y como claramente así sería, obtendrías la siguiente tabla.

numero_cuenta	balance
1	0
2	2000

Tabla 3. Modelo de la tabla ventas

No obstante, si queremos volver a ejecutar la misma transacción recibimos un error como este:

```
ERROR:  el nuevo registro para la relación «cuentas» viola la
restricción «check» «cuentas_balance_check»
```

Puesto que la cuenta 1 quedaría con balance negativo y violaría la restricción declarada en la creación de la tabla y como resultado no se realiza la transacción.

Ejercicio propuesto (1)

La empresa Mawashi Cars Spa ha detectado un problema con el sistema que permite registrar ventas de los autos que no tienen stock y ha provocado incomodidades con los clientes, por lo que necesita urgentemente que se arregle este error y evitar futuras ventas en donde no se tenga disponibilidad del auto que se quiera vender. En el apoyo lectura de esta sesión encontrarás 2 archivos llamados autos.csv y ventas.csv para el desarrollo de este ejercicio propuesto.

Para empezar con las soluciones debes crear las tablas correspondientes a los archivos .csv deduciendo el tipo de dato de cada columna, definiendo la llave primaria y foránea correspondiente en las tablas, además de agregarle el comando CHECK a la columna stock en la tabla autos.

Realizar una transacción que incluya la inserción de un registro en la tabla “ventas” del auto con id 5 y una actualización en la tabla “autos” que reste 1 al stock de dicho auto.

ROLLBACK

Con este comando podemos deshacer las transacciones que se hayan ejecutado, revirtiendo los cambios realizados por una transacción hasta el último COMMIT o ROLLBACK ejecutado. Esto permite controlar los flujos de ejecución en nuestras transacciones, de manera que volvamos a un estado anterior, sin alterar los datos almacenados.

Continuando con el ejemplo del banco, en el punto anterior realizamos una transacción en donde transferimos $1000 de la cuenta 1 a la cuenta 2, como verás no obtuvimos ningún error, no obstante podemos verificar que en las transacciones que terminan en error no se altera el estado de nuestros datos. ¿Y cómo lo hacemos? Ejecuta el siguiente código en tu terminal.

```
BEGIN TRANSACTION;

UPDATE cuentas SET balance = balance + 1000 WHERE numero_cuenta = 2;
UPDATE cuentas SET balance = balance - 1000 WHERE numero_cuenta = 1;

ROLLBACK;
```

Como notarás, recibiste un error con el siguiente detalle por la terminal.

```
DETALLE:  La fila que falla contiene (1, -1000.00).
```

Esto debido a la restricción del atributo balance. El ROLLBACK ha cancelado todos los cambios involucrados en la transacción por lo que si consultas la tabla obtendrás lo siguiente.

numero_cuenta	balance
1	0
2	2000

Tabla 4. Modelo de la tabla ventas

Como verás no hubo ningún cambio en los datos, a pesar de haber cargado $1000 a la cuenta 2 primero, esta no se realiza puesto que la resta al balance de la cuenta 1 devolvió un error.

Si por alguna razón, en medio de una transacción se decide que ya no se quiere registrar los cambios (tal vez nos dimos cuenta que estamos actualizando todos los registros de nuestra base y no es lo que buscábamos), se puede recurrir a la orden ROLLBACK en lugar de COMMIT y todas las actualizaciones hasta ese punto quedarán canceladas.

Ejercicio propuesto (2)

Continuando con el ejercicio propuesto de Mawashi Cars, realiza otra transacción en donde insertes registros en la tabla ventas pero ahora con los autos de id 2, 3 y 5. Considera que alguno de estos autos podría no tener stock por lo que deberás realizar un rollback al recibir el error por consola y confirmar que los cambios no fueron realizados consultando ambas tablas.

SAVEPOINT

Es posible tener un mayor control de las transacciones por medio de puntos de recuperación (SAVEPOINTS). Éstos permiten seleccionar qué partes de la transacción serán descartadas bajo ciertas condiciones, mientras el resto de las operaciones sí se ejecutan.

Después de definir un punto de recuperación seguido de su nombre representativo, se puede volver a él por medio de ROLLBACK TO. Todos los cambios realizados por la transacción, entre el punto de recuperación y el rollback se descartan.

Probemos esto, con el siguiente código Intentemos registrar una nueva cuenta de número 3 en nuestra tabla “cuentas” con un saldo de $5000 y justo luego guardemos ese punto de la transacción con un SAVEPOINT de nombre “nueva_cuenta”.

```
BEGIN TRANSACTION;
INSERT INTO cuentas(numero_cuenta, balance) VALUES (3,
5000);
SAVEPOINT nueva_cuenta;
```

Hasta este punto tenemos la transacción en curso y hemos fijado que podríamos volver a este estado en cualquier circunstancia. Ahora, intentemos transferir a esta nueva cuenta $3000 desde la cuenta 2. Para esto continua la transacción de la siguiente manera.

```
UPDATE cuentas SET balance = balance + 3000 WHERE numero_cuenta = 3;
UPDATE cuentas SET balance = balance - 3000 WHERE numero_cuenta = 2;
-- Justo acá deberás recibir un error
ROLLBACK TO nueva_cuenta;
COMMIT;
```

Esto provocará que PostgreSQL te devuelva un error por la terminal puesto que la cuenta 2 no dispone de $3000. ¿Entonces qué pasó? Si consultas la tabla “cuentas” obtendrás lo siguiente.

numero_cuenta	balance
1	0
2	2000
3	5000

Tabla 5. Tabla cuentas luego de un ROLLBACK TO a un punto guardado en una transacción

Como puedes notar se registró con éxito la cuenta 3, no obstante su balance sigue siendo $5000 y el balance de la cuenta 2 no se redujo. Esto es porque volvimos al punto guardado, luego de haber hecho la inserción de nuestra nueva cuenta a pesar de no haberse procesado lo que le procedía a esa instrucción.

Ejercicio propuesto (3)

Realizar una nueva transacción intentando volver a registrar las ventas de los autos de id 2, 3 y 5, pero para evitar que no se realice ningún cambio crea un punto de guardado por cada auto que no arroje un error en su actualización de stock y si recibes algún error realiza un ROLLBACK a último SAVEPOINT. Posteriormente revisa ambas tablas para verificar que se realizaron los cambios correspondientes.

AUTOCOMMIT

En PostgreSQL, por defecto viene configurado el modo AUTOCOMMIT, es decir, que implícitamente una vez que hemos realizado una acción sobre la base de datos, ésta realiza un COMMIT.

Comprobémoslo con la siguiente instrucción:

```
\echo :AUTOCOMMIT
```

La respuesta a esto por parte del motor es "ON", es decir "activo". En caso de recibir como respuesta “:autocommit” es porque lo estás escribiendo en minúsculas y no se reconoce, debes tipear en mayúscula.

También podemos comprobarlo al insertar cualquier registro en nuestra base, por ejemplo en la tabla de cuentas:

```
INSERT INTRO cuentas values(4,1000);
```

Recibiremos la respuesta “INSERT 0 1” puesto que ingresamos una nueva cuenta sin problemas, y si intentas ejecutar un COMMIT PostgreSQL reacciona con la siguiente alerta

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

Esto es porque por defecto PostgreSQL realiza un COMMIT cada vez que insertamos, actualizamos o eliminamos un registro.

Para modificar esto, podemos ejecutar el siguiente comando:

```
\set AUTOCOMMIT off
```

Aunque no es lo más recomendable mantener el AUTOCOMMIT en off, con esto, tendremos control de todas las transacciones realizadas y nos puede servir como espacio para realizar todas las pruebas que queramos en nuestra base de datos.

Cargar una base de datos utilizando dump

Dump es una herramienta que nos permite de manera simple generar copias de nuestra base de datos en archivos de extensión .sql, con la secuencia de instrucciones que representen la composición de las tablas y su registros, para así respaldarla o bien cargar datos de este respaldo en un momento determinado.

Dump para exportar una base de datos

Para crear un archivo a partir de una base de datos, deberemos ejecutar el siguiente comando desde la terminal.

```
pg_dump -U nombre_usuario nombre_db > db.sql
```

Donde db.sql será el archivo que se creará a partir de la base de datos solicitada, y quedará guardado en la ubicación donde nos encontramos. Si tienes Linux y recibes el mensaje “permiso denegado” por la terminal, te recomiendo que realices esto en la carpeta /tmp de tu distribución.

En el caso de que utilices Windows, primero debes posicionarte en el siguiente directorio:

```
cd /Program Files\PostgreSQL\10\bin ejecutar pg_dump.exe
```

y luego ejecutar el siguiente comando, de esta forma:

```
pg_dump.exe -U nombre_usuario -d nombre_bd -f nombre_ruta
```

En el caso de que desconozcas la ruta aquí tenemos un ejemplo de nombre_ruta por ejemplo:

```
pg_dump.exe -U katy -d banco -f C:\Users\Catalina\Desktop\banco.sql
```

Dump para exportar todas las bases de datos

Es posible que necesitemos exportar no solo una base de datos sino todas las que tengamos en el sistema, por si queremos obtener o resguardar el respaldo de todas las bases de datos. Para lograr esto debemos ejecutar el siguiente comando.

```
$ sudo su - postgres
$ pg_dumpall > /directorio/dumpall.sql
```

Nota que debes primero autenticarte con tu usuario en la terminal para luego hacer el dumpall con todas las bases de datos.

Restaurar una base de datos

Supongamos que estamos cambiando el servidor en donde tendremos alojada nuestra base de datos y hemos realizado una exportación para hacer la migración, para importar este backup en el nuevo servidor debemos utilizar el siguiente comando:

```
$ sudo su - postgres
$ psql -U postgres nombredb < archivo_restauracion.sql
```

Es importante que sepas que para importar una base de datos en un archivo .sql ya debes tener una creada con el mismo nombre.

Para el caso de windows, al crear una base de datos “nombre_bd” usando shell SQL, se debe de posicionar en el siguiente directorio:

```
cd /Program Files\PostgreSQL\10\bin
```

Luego, se procede a ejecutar PSQL con el usuario correspondiente, como se muestra a continuación:

```
psql -U nombre_usuario nombre_bd < nombre_ruta
```

Ejemplo:

```
psql -U katy banco < C:\Users\Catalina\Desktop\banco.sql
```

Restaurar todas las bases de datos

Supongamos que necesitas hacer una migración total de todas las bases de datos, no solo una y tienes el fichero .sql, luego de la exportación que ya realizaste y solo necesitas importar en el nuevo servidor. Para eso deberás utilizar el siguiente comando:

```
$ sudo su - postgres
$ psql -f /var/lib/pgsql/backups/dumpall.sql mydb
```

Ejercicio guiado: Pongamos a prueba los conocimientos

La pizzeria nacional Hot Cheese, ofrece en su sitio web para el servicio de pizzas a domicilio y apesar de estar funcionando bien los primeros meses ha bajado su clientela por incomodidades en sus usuarios, que realizan pagos electrónicos en la aplicación y posteriormente reciben un correo de disculpas por la empresa diciendo que la pizza que compró ya no está disponible. El dueño de la pizzería ha tomado cartas sobre el asunto y ha solicitado contratar a un programador de bases de datos, para que cree una nueva base de datos aplicando las restricciones para evitar que siga sucediendo esta situación.

Para la solución de este ejercicio deberás realizar lo siguiente:

1. Crear una base de datos llamada “pizzeria”.
2. Conectarse a la base de datos pizzeria.
3. Crear 2 tablas llamadas “ventas” y “pizzas” con la siguiente estructura.

cliente	fecha	monto	pizza(FK)

Tabla 6. Modelo de la tabla ventas.

id(PK)	stock	costo	nombre

Tabla 7. Modelo de la tabla pizzas.

.

La columna stock debe tener una restricción que diga que su valor debe ser mayor o igual a 0.

4. Agregar 1 registro a la tabla “pizzas” seteando como stock inicial 0.
5. Realizar una transacción que registre una nueva pizza con un stock positivo mayor a 1.
6. Realizar una transacción que registre una venta con la pizza con stock 0 e intentar actualizar su stock restándole 1.
7. Realizar una transacción que intente registrar 1 venta por cada pizza, guardando un SAVEPOINT luego de la primera venta y volviendo a este punto si surge un error.
8. Exportar la base de datos “pizzeria” como un archivo pizzeria.sql.
9. Eliminar la base de datos “pizzeria”.
10. Importar el archivo pizzeria.sql.

Pasos a seguir

- ***Paso 1:*** Partimos este ejercicio con la creación de nuestra base de datos, cuyo nombre será “pizzeria”

```
CREATE DATABASE pizzeria;
```

- ***Paso 2:*** Procedemos con la conexión a la base de datos “pizzeria” previamente creada

```
\c pizzeria ;
```

- ***Paso 3:*** Según el planteamiento del problema, necesitamos crear 2 tablas llamadas “ventas” y “pizzas” en donde “pizzas” debe tener una restricción en su atributo “stock” y como clave primaria el id, y para el caso de “venta” una clave foránea en el atributo id también, entendiendo que esta depende de la tabla “pizzas”. Para esto ejecutamos las siguientes instrucciones.

```
CREATE TABLE pizzas(
id INT,
stock INT CHECK (stock >= 0.00),
costo DECIMAL,
nombre VARCHAR(25),
PRIMARY KEY(id)
);
```

```
CREATE TABLE ventas(
cliente VARCHAR(20),
fecha DATE,
monto DECIMAL,
```

```
pizza INT,
FOREIGN KEY (pizza) REFERENCES pizzas(id)
);
```

- ***Paso 4:*** Para proceder con nuestras pruebas necesitamos agregar 1 registro a la tabla “pizzas”, sabiendo que probaremos las transacciones en este ejercicio, en los próximos pasos seteamos como stock inicial 0 recordando que este es el atributo con la restricción.

```
INSERT INTO pizzas (id, stock, costo, nombre) VALUES (1, 0, 12000,
'Uhlalá');
```

Hasta este punto nuestra tabla “pizzas” va quedando de la siguiente manera.

id	stock	costo	nombre
1	0	12000	Uhlalá

Tabla 8. Modelo de la tabla ventas.

- ***Paso 5:*** Probemos una primera transacción ingresando una nueva pizza a la tabla “pizzas” y ésta tendrá un stock mayor a 1, por lo que usaremos el siguiente código.

```
BEGIN;
INSERT INTO pizzas (id, stock, costo, nombre) VALUES (2, 2, 15000,
'Jamón a todo dar');
COMMIT;
```

Quedando ahora nuestra tabla de la siguiente manera.

id	stock	costo	nombre
1	0	12000	Uhlalá
2	2	15000	Jamón a todo dar

Tabla 9. Tabla pizzas.

- ***Paso 6:*** Es hora de probar nuestra base de datos, por lo que realizaremos una transacción que registre una venta con la pizza con stock 0 e intente actualizar su stock restándole 1.

```
BEGIN;
INSERT INTO ventas (cliente, fecha, monto, pizza) VALUES ('Dexter
Gonzalez', '2020-02-02', 12000, 1);
UPDATE pizzas SET stock = stock - 1 WHERE id = 1;
COMMIT;
```

Esto nos devolverá el siguiente error.

```
ERROR:  el nuevo registro para la relación «pizzas» viola la restricción
«check» «stock»
```

Y es una buena señal pues no tendría sentido vender una pizza que no se tiene. Si consultamos la tabla obtendremos lo siguiente.

id	stock	costo	nombre
1	0	12000	Uhlalá
2	2	15000	Jamón a todo dar

Tabla 10. Tabla pizzas luego de una transacción fallida.

Como puedes notar correctamente no han cambiado los datos.

- ***Paso 7:*** Ahora supongamos que se quiere realizar una venta de las 2 pizzas registradas pero solo 1 de ellas tiene stock, por lo que marcaremos un punto de guardado justo después de actualizar el stock de la pizza disponible y volviendo a este punto en el momento que recibimos un error.

```
BEGIN;
INSERT INTO ventas (cliente, fecha, monto, pizza) VALUES ('Juan Bravo',
'2020-02-02', 15000, 2);
UPDATE pizzas SET stock = stock - 1 WHERE id = 2;
SAVEPOINT checkpoint;
INSERT INTO ventas (cliente, fecha, monto, pizza) VALUES ('Utonio
Ramirez', '2020-02-02', 12000, 1);
UPDATE pizzas SET stock = stock - 1 WHERE id = 1;
-- Acá recibirás un error por intentar rebajar el stock de una pizza
```

```
cuyo stock es 0
ROLLBACK TO checkpoint;
```

Si revisamos la tabla de ventas veremos lo siguiente.

cliente	fecha	monto	pizza
Juan Bravo	2020-02-02	15000	2

Tabla 11. Tabla ventas luego de una transacción con SAVEPOINT.

Comprobamos entonces que se registró esa venta sin problemas y si revisamos la tabla pizzas obtendremos lo siguiente.

id	stock	costo	nombre
1	0	12000	Uhlalá
2	1	15000	Jamón a todo dar

Tabla 12. Tabla pizzas luego de una transacción con SAVEPOINT.

En donde confirmamos que se rebajó la pizza de id 2 pero por supuesto no cambió la pizza de id 1 pues esta fue la que nos dió problemas en la transacción.

- ***Paso 8:*** Supongamos que la pizzería Hot Cheese no le va muy bien y decide guardar un respaldo de sus datos y solicita que se exporte la base de datos como un archivo llamado pizzeria.sql. Para esto realizamos la siguiente instrucción.

```
pg_dump -U postgres pizzeria > pizzeria.sql
```

- ***Paso 9:*** En el caso que la pizzería cierre sus puertas y solicite eliminar la base de datos “pizzeria” usamos la siguiente sentencia.

```
DROP DATABASE pizzeria;
```

- ***Paso 10:*** Y si quisiera aperturar el negocio y recuperar los datos de su último respaldo, deberíamos crear de nuevo la base de datos y posteriormente importar

nuestro archivo pizzeria.sql, exportado en el paso 8 tal y como te muestro a continuación.

```
CREATE DATABASE pizzeria ;
CREATE DATABASE pizzeria ;
```

```
$ sudo su - postgres
$ psql -U postgres pizzeria < pizzeria.sql
```

Solución de los ejercicios propuestos

1. La empresa Mawashi Cars Spa ha detectado un problema con el sistema que permite registrar ventas de los autos que no tienen stock y ha provocado incomodidades con los clientes, por lo que necesita urgentemente que se arregle este error y evitar futuras ventas en donde no se tenga disponibilidad del auto que se quiera vender. En el apoyo lectura de esta sesión encontraras 2 archivos llamados autos.csv y ventas.csv para el desarrollo de este ejercicio propuesto.

   Para empezar con las soluciones debes crear las tablas correspondientes a los archivos .csv deduciendo el tipo de dato de cada columna, definiendo la llave primaria y foránea correspondiente en las tablas, además de agregarle el comando CHECK a la columna stock en la tabla autos.

   Realizar una transacción que incluya la inserción de un registro en la tabla “ventas” del auto con id 5 y una actualización en la tabla “autos” que reste 1 al stock de dicho auto.

```
BEGIN TRANSACTION;
INSERT INTO ventas (fecha, id_auto, cliente, referencia, cantidad,
metodo_pago) values ('2020-02-02', 5 , 'Diana Palma', 40302, 2000000,
'credito');
UPDATE autos SET stock = stock - 1 WHERE id = 5;
COMMIT;
```

2. Continuando con el ejercicio propuesto de Mawashi Cars, realiza otra transacción en donde insertes registros en la tabla ventas pero ahora con los autos de id 2, 3 y 5. Considera que alguno de estos autos podría no tener stock por lo que deberás realizar un rollback al recibir el error por consola y confirmar que los cambios no fueron realizados consultando ambas tablas.

```
BEGIN TRANSACTION;
INSERT INTO ventas (fecha, id_auto, cliente, referencia, cantidad,
metodo_pago) values ('2020-02-03', 2 , 'Maria Urbaez', 23302, 2000000,
'debito');
UPDATE autos SET stock = stock - 1 WHERE id = 2;
INSERT INTO ventas (fecha, id_auto, cliente, referencia, cantidad,
metodo_pago) values ('2020-02-04', 3 , 'Chary Malave', 43302, 1200000,
'credito');
UPDATE autos SET stock = stock - 1 WHERE id = 3;
INSERT INTO ventas (fecha, id_auto, cliente, referencia, cantidad,
metodo_pago) values ('2020-02-05', 5 , 'Diana Palma', 12302, 50000000,
'credito');
UPDATE autos SET stock = stock - 1 WHERE id = 5;
ROLLBACK;
SELECT * FROM ventas;
SELECT * FROM autos;
```

3. Realizar una nueva transacción intentando volver a registrar las ventas de los autos de id 2, 3 y 5, pero para evitar que no se realice ningún cambio crea un punto de guardado por cada auto que no arroje un error en su actualización de stock y si recibes algún error realiza un ROLLBACK a último SAVEPOINT. Posteriormente revisa ambas tablas para verificar que se realizaron los cambios correspondientes.

```
BEGIN TRANSACTION;
INSERT INTO ventas (fecha, id_auto, cliente, referencia, cantidad,
metodo_pago) values ('2020-02-03', 2 , 'Maria Urbaez', 23302, 2000000,
'debito');
UPDATE autos SET stock = stock - 1 WHERE id = 2;
SAVEPOINT checkpoint1;
INSERT INTO ventas (fecha, id_auto, cliente, referencia, cantidad,
metodo_pago) values ('2020-02-04', 3 , 'Chary Malave', 43302, 1200000,
'credito');
UPDATE autos SET stock = stock - 1 WHERE id = 3;
SAVEPOINT checkpoint2;
INSERT INTO ventas (fecha, id_auto, cliente, referencia, cantidad,
metodo_pago) values ('2020-02-05', 5 , 'Diana Palma', 12302, 50000000,
'credito');
UPDATE autos SET stock = stock - 1 WHERE id = 5;
ROLLBACK TO checkpoint2;
SELECT * FROM ventas;
SELECT * FROM autos;
```