

EXERCISES QUE TRABAJAREMOS EN EL CUE:

- EXERCISE 1: BEGIN, COMMIT Y ROLLBACK.
- EXERCISE 2: TRANSACCIONES.
- EXERCISE 3: GESTIÓN DE TRANSACCIONES SQL.
- EXERCISE 4: CONSISTENCIA Y AISLAMIENTO.
- EXERCISE 5: PRIVILEGIOS Y USUARIOS.

EXERCISE 1: BEGIN, COMMIT Y ROLLBACK

Es fundamental conocer el concepto de transacción en los sistemas gestores de bases de datos, como PostgreSQL. Una transacción empaqueta varios pasos en una operación, de forma que se completen todos, o ninguno de ellos. Los estados intermedios entre los pasos no son visibles para otras transacciones ocurridas en el mismo momento.

En el caso de que ocurra algún fallo que impida que se complete la transacción, ninguno de los pasos se ejecuta, y éstos no afectan a los objetos de la base de datos.

Los pasos dentro de una transacción son varias sentencias SQL, las cuales deben completarse todas para que queden registradas. Para comenzar una, utilizamos el comando BEGIN. Luego, para indicar al sistema que han terminado correctamente todas las sentencias SQL, se emplea el comando COMMIT. En algunas ocasiones, tenemos que desechar uno de los pasos que se están realizando, entonces, para cancelar la transacción comenzada, se usa el comando ROLLBACK.

COMANDO BEGIN

Cuando lo utilizamos, **el sistema permite que se ejecuten todas las sentencias SQL que necesitamos**, y las registra en un fichero. A continuación, revisaremos un ejemplo donde se comienza una transacción, en la que deben completarse satisfactoriamente todas las sentencias.

```
1 BEGIN;
2 UPDATE cuentas SET balance = balance - 100.00 WHERE n_cuenta =
3 0127365;
4 UPDATE cuentas SET balance = balance + 100.00 WHERE n_cuenta =
5 0795417;
```

COMANDO COMMIT

Al ejecutarlo, **estamos confirmando que todas las sentencias son correctas**. Es decir que, mientras no se haya implementado, las sentencias no quedarán registradas. Por ejemplo: **si cerramos la conexión antes de ejecutar este comando, no se verá afectada ninguna de las relaciones de la base de datos**. A continuación, veremos un ejemplo en el cual se comienza una transacción, se ejecutan una serie de pasos, y confirmamos que todas las sentencias están correctas.

```
1 BEGIN;
2 INSERT INTO cuentas (n_cuenta, nombre, balance) VALUES (0679259,
3 'Pepe', 200);
4 UPDATE cuentas SET balance = balance - 137.00 WHERE nombre = 'Pepe';
5 UPDATE cuentas SET balance = balance + 137.00 WHERE nombre = 'Juan';
6 SELECT nombre, balance FROM cuentas WHERE nombre = 'Pepe' AND nombre =
7 'Juan';
8 COMMIT;
```

COMANDO ROLLBACK

Con éste **podemos desechar las transacciones que se hayan ejecutado**. Por lo tanto, después de haber realizado y confirmado una transacción, PostgreSQL nos permite anularla, de forma que no se modifiquen los datos de nuestra base de datos. A continuación, un ejemplo donde vamos a anular la transacción confirmada anteriormente.

```
1 BEGIN;
2 "SENTENCIAS SQL"
3 COMMIT;
4 ROLLBACK;
```

Para poder utilizar estos comandos mencionados: **BEGIN**, **COMMIT** y **ROLLBACK**, debemos **desactivar el AUTOCOMMIT desde psql**. Esta opción es a nivel de cliente, y por defecto, está activada. Es así como toda sentencia ejecutada queda confirmada, y también registrada en la base de datos.

```
SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Contraseña para usuario postgres:
psql (13.4)
ADVERTENCIA: El código de página de la consola (850) difiere del código
de página de Windows (1252).
Los caracteres de 8 bits pueden funcionar incorrectamente.
Vea la página de referencia de psql «Notes for Windows users»
para obtener más detalles.
Digite «help» para obtener ayuda.
postgres=# \set AUTOCOMMIT off
```

EXERCISE 2: TRANSACCIONES

Para comprender la propiedad de atomicidad, se presentan algunos ejemplos: el primero de éstos hace uso del comando COMMIT, el cual se utiliza para confirmar como permanentes, las modificaciones realizadas en una transacción:

Indica el inicio de una secuencia de comandos

Inserción de un registro en la tabla clientes

```
BEGIN;
INSERT INTO public."Clientes" VALUES
('CLI004', 'Nixon', 'El Cambio', '0981242156', 'INM003', 250);
COMMIT;
```

Confirma las modificaciones realizadas en una transacción

Para comprobar que los comandos anteriores se ejecutaron correctamente, y se almacenaron en la base de datos, se hace una consulta en la tabla clientes:

```
1 SELECT * FROM "Clientes";
```

Su resultado es:

	id_cli character varying	nombre_cli character varying	direccion_cli character varying	telefono_cli character varying	inmueble_preferido_cli character varying	importe_maximo_cli double precision
1	CLI001	Ufredo Romero	Santa Rosa	2890910	INM001	300
2	CLI002	Diego Romero	Santa Rosa	2890911	INM002	150
3	CLI003	Jazmin Quirola	Machala	2890912	INM002	190
4	CLI004	Nixon	El Cambio	0981242156	INM003	250

Con esto se puede evidenciar que el registro de la transacción se ha guardado en la base de datos inmobiliaria de forma correcta.

Otra forma de comprobar la propiedad de atomicidad es la que se indica a continuación. En ésta se hace uso del comando ROLLBACK, el cual permite deshacer todas las modificaciones que se han realizado a la base de datos, pero que no han sido escritas en el disco duro por la sentencia COMMIT.

PROPIEDAD DE ATOMICIDAD.

Indica el inicio de una secuencia de comandos

Insertión de un registro en la tabla clientes

```
BEGIN;
INSERT INTO public."Clientes" VALUES
('CLI005', 'Ezequiel', 'Atahualpa', '0981233158', 'INM004', 200);
ROLLBACK;
```

Deshace las modificaciones que no han sido escritas en el disco duro

Para comprobar que los comandos anteriores se ejecutaron, y que no se almacenaron en la base de datos, se hace una consulta en la tabla clientes:

```
1 SELECT * FROM "Clientes";
```

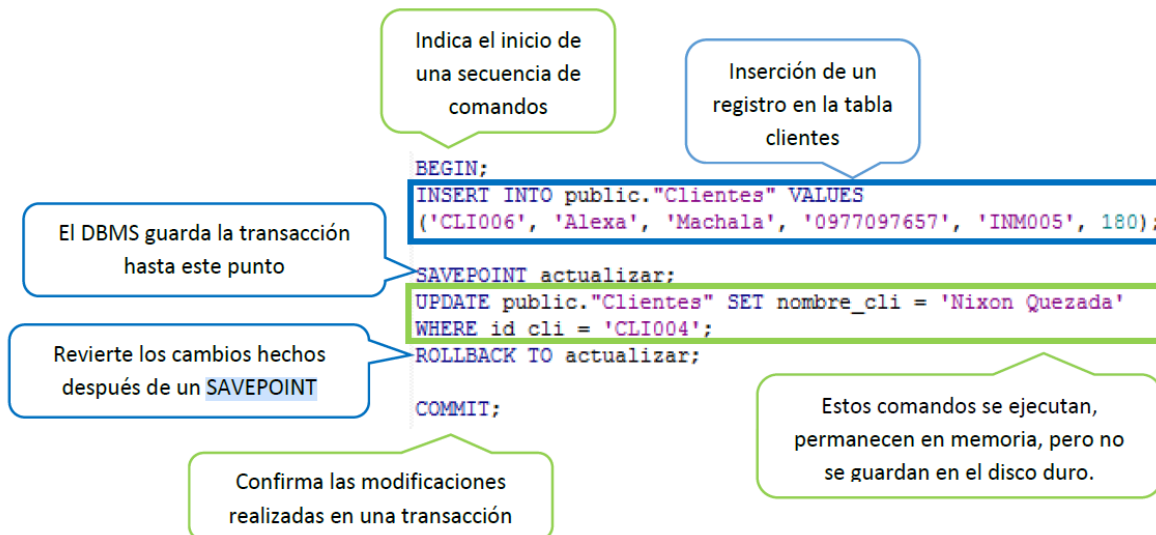
Su resultado es:

	id_cli character varying	nombre_cli character varying	direccion_cli character varying	telefono_cli character varying	inmueble_preferido_cli character varying	importe_maximo_cli double precision
1	CLI001	Ufredo Romero	Santa Rosa	2890910	INM001	300
2	CLI002	Diego Romero	Santa Rosa	2890911	INM002	150
3	CLI003	Jazmin Quirola	Machala	2890912	INM002	190
4	CLI004	Nixon	El Cambio	0981242156	INM003	250

Con ello, se puede evidenciar que el registro de la transacción no se ha guardado en la base de datos inmobiliaria.

Otros de los comandos que se pueden usar para la comprobación de la propiedad de atomicidad, son: SAVEPOINT y ROLLBACK TO. EL primero le indica al DBMS la ubicación de un punto de retorno en una transacción, en caso de que ésta sea cancelada. Mientras que el segundo, revierte los cambios hechos después de un SAVEPOINT.

COMPROBACIÓN DE ATOMICIDAD.



Para comprobar que los comandos anteriores se ejecutaron, y que éstos cumplen con las definiciones, se hace una consulta en la tabla clientes de la base de datos inmobiliaria:

```
1 SELECT * FROM "Clientes";
```



Su resultado es:

	id_cli character varying	nombre_cli character varying	direccion_cli character varying	telefono_cli character varying	inmueble_preferido_cli character varying	importe_maximo_cli double precision
1	CLI001	Ufredo Romero	Santa Rosa	2890910	INM001	300
2	CLI002	Diego Romero	Santa Rosa	2890911	INM002	150
3	CLI003	Jazmin Quirola	Machala	2890912	INM002	190
4	CLI004	Nixon	El Cambio	0981242156	INM003	250
5	CLI006	Alexa	Machala	0977097657	INM005	180

Con esto se evidencia que las instrucciones que se encuentran dentro del bloque SAVEPOINT <actualizar>, y ROLLBACK TO <actualizar>, no han efectuado ningún cambio, pues no han sido escritas en el disco duro; y, en cambio, la información que se ubica fuera del bloque SAVEPOINT <actualizar> y ROLLBACK TO <actualizar>, si ha sido almacenada en la base de datos.

Reglas ACID: comprobar la propiedad de consistencia, utilizando la tabla "Contrato".

EXERCISE 3: GESTIÓN DE TRANSACCIONES SQL.

Veremos un ejemplo donde aplicaremos transacciones utilizando sentencias SQL. Comenzaremos con la palabra reservada "BEGIN" y, a continuación, "COMMIT". Estas nos permiten confirmar que las sentencias escritas entre ellas son correctas y quedan registradas en un fichero.

Query	Query History
1	BEGIN;
2	
3	
4	COMMIT;
5	

Crearemos un ejemplo utilizando estas palabras reservadas. Para ello, crearemos dos usuarios con contraseñas. El primero será "Juan" con la contraseña "pass1".

Query	Query History
1	BEGIN;
2	CREATE USER Juan WITH PASSWORD 'pass1';
3	
4	COMMIT;

Al ejecutar, nuestra query queda registrada en el fichero gracias a la palabra reservada "COMMIT".

The screenshot shows the pgAdmin 4 query editor with a query window. The query text is:

```
1 BEGIN;
2 CREATE USER Juan WITH PASSWORD 'pass1';
3
4 COMMIT;
```

The "Messages" tab is selected, showing the output:

```
COMMIT
Query returned successfully in 81 msec.
```

Desde pgAdmin 4, podemos decidir si queremos desactivar el "Auto Commit". En esta oportunidad, lo desactivaremos para llevar a cabo los ejemplos que realizaremos a continuación.



Ahora, borraremos el usuario que acabamos de crear. Para eso, utilizaremos "DROP" y presionaremos "Execute".

The screenshot shows the pgAdmin 4 query editor with a query window. The query text is:

```
6 DROP USER Juan;
7
8
9
```

The "Messages" tab is selected, showing the output:

```
DROP ROLE
Query returned successfully in 96 msec.
```

Si volvemos a ejecutar la query de creación de usuario, nos aparecerá una advertencia indicando que ya hay una transacción en curso.

```
Query  Query History
1  BEGIN;
2  CREATE USER Juan WITH PASSWORD 'pass1';
3
4  COMMIT;
5
6  DROP USER Juan;
7
8
```

Data Output Messages Notifications

WARNING: ya hay una transacción en curso
COMMIT

Query returned successfully in 47 msec.

Si ejecutamos solo el "COMMIT", veremos un mensaje de advertencia distinto, indicándonos que ya no hay una transacción en curso.

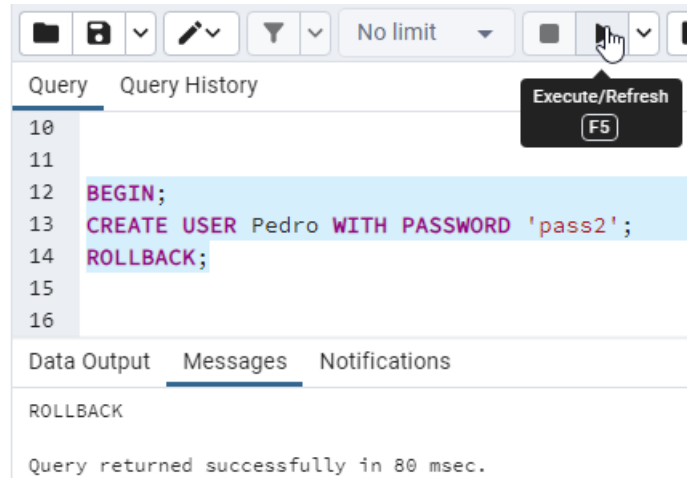
```
3
4  COMMIT;
5
6  DROP USER Juan;
7
8
9
10
```

Data Output Messages Notifications

WARNING: no hay una transacción en curso
COMMIT

Query returned successfully in 76 msec.

Ahora crearemos un nuevo usuario llamado "Pedro" utilizando "BEGIN" y "ROLLBACK". Este último comando nos servirá para deshacer las transacciones que se hayan ejecutado. Hacemos clic en "Execute" y veremos que todo se realizó correctamente.



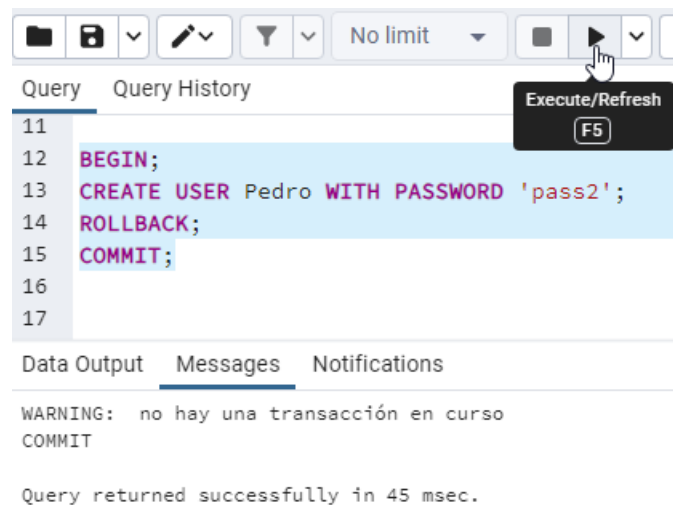
The screenshot shows a SQL IDE interface. The top toolbar includes icons for file operations, a dropdown menu, a filter icon, and a "No limit" dropdown. A tooltip for the "Execute/Refresh" button (F5) is visible. The query editor contains the following SQL code:

```
10  
11  
12 BEGIN;  
13 CREATE USER Pedro WITH PASSWORD 'pass2';  
14 ROLLBACK;  
15  
16
```

The "Messages" tab is selected, displaying the output:

```
ROLLBACK  
  
Query returned successfully in 80 msec.
```

Si agregamos un "COMMIT" al final de la query anterior, veremos el mensaje de que no hay una transacción en curso.



The screenshot shows the same SQL IDE interface. The query editor now includes a "COMMIT" statement at the end:

```
11  
12 BEGIN;  
13 CREATE USER Pedro WITH PASSWORD 'pass2';  
14 ROLLBACK;  
15 COMMIT;  
16  
17
```

The "Messages" tab displays the following output:

```
WARNING: no hay una transacción en curso  
COMMIT  
  
Query returned successfully in 45 msec.
```

Intentaremos ahora borrar al usuario “Pedro”, pero nos daremos cuenta de que este rol no existe debido al comando “ROLLBACK”.

```
11
12 BEGIN;
13 CREATE USER Pedro WITH PASSWORD 'pass2';
14 ROLLBACK;
15 COMMIT;
16
17 DROP USER Pedro;
18
19
```

Data Output Messages Notifications

ERROR: no existe el rol «pedro»

SQL state: 42704

Comentamos el comando “Rollback”, creamos el usuario “Pedro” con “CREATE”. A continuación, listaremos los usuarios creados y veremos finalmente todos los que existen en el sistema.

```
11
12 BEGIN;
13 CREATE USER Pedro WITH PASSWORD 'pass2';
14 --ROLLBACK;
15 COMMIT;
16
17 DROP USER Pedro;
18
19
```


Data Output Messages Notifications

COMMIT

Query returned successfully in 70 msec.

```
18
19 SELECT username FROM pg_user;
20
```

Data Output Messages Notifications

	username 
1	postgres
2	juan
3	pedro

Para continuar con el ejemplo, crearemos algunas tablas. La primera será “producto” con los atributos “id”, “nombre” y “precio”.

```
CREATE TABLE producto (  
    id SERIAL PRIMARY KEY,  
    nombre VARCHAR(100) NOT NULL,  
    precio SMALLINT  
);
```

Luego, crearemos la tabla “cuentas” con los atributos con los atributos “n_cuenta”, “nombre” y “balance”.

```
CREATE TABLE cuentas (  
    n_cuenta NUMERIC PRIMARY KEY,  
    nombre VARCHAR(100) NOT NULL,  
    balance REAL  
);
```

Para gestionar las transacciones, agregaremos “BEGIN” y “COMMIT” al principio y al final de la consulta, respectivamente. Además, entre la creación de tablas, incluiremos un “ROLLBACK” y un “COMMIT”

```
BEGIN;  
CREATE TABLE producto (  
    id SERIAL PRIMARY KEY,  
    nombre VARCHAR(100) NOT NULL,  
    precio SMALLINT  
);  
  
ROLLBACK;  
COMMIT;  
CREATE TABLE cuentas (  
    n_cuenta NUMERIC PRIMARY KEY,  
    nombre VARCHAR(100) NOT NULL,  
    balance REAL  
);  
COMMIT;
```

Seleccionaremos nuestra query y la ejecutaremos. Al tener un “ROLLBACK”, notaremos que solo se creará una tabla.

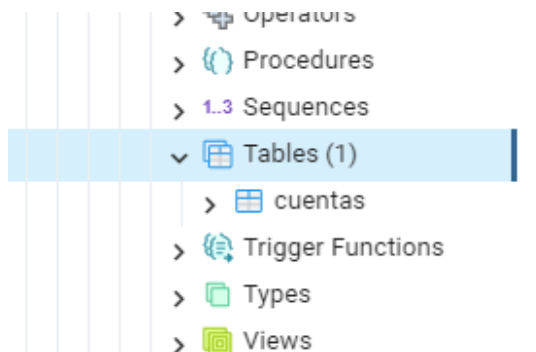
```
18 BEGIN;
19 CREATE TABLE producto (
20     id SERIAL PRIMARY KEY,
21     nombre VARCHAR(100) NOT NULL,
22     precio SMALLINT
23 );
24
25 ROLLBACK;
26 COMMIT;
27 CREATE TABLE cuentas (
28     n_cuenta NUMERIC PRIMARY KEY,
29     nombre VARCHAR(100) NOT NULL,
30     balance REAL
31 );
32 COMMIT;
```

Data Output **Messages** Notifications

WARNING: ya hay una transacción en curso
WARNING: no hay una transacción en curso
WARNING: no hay una transacción en curso
COMMIT

Query returned successfully in 99 msec.

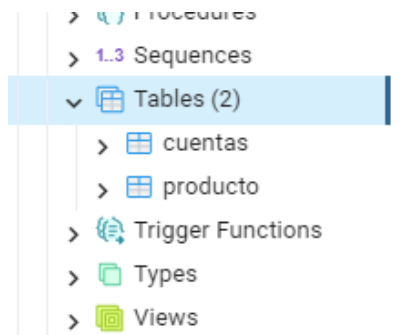
Si vamos y revisamos las tablas podremos confirmar que sólo se creó la tabla “cuentas” ya que después de la creación de la tabla “producto” tenemos un “rollback”



Para crear la tabla "producto", comentaremos el "ROLLBACK" y ejecutaremos la consulta correspondiente.

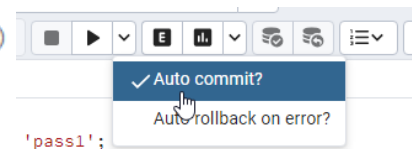
```
BEGIN;  
CREATE TABLE producto (  
    id SERIAL PRIMARY KEY,  
    nombre VARCHAR(100) NOT NULL,  
    precio SMALLINT  
);  
--ROLLBACK;  
COMMIT;
```

Al volver a revisar las tablas, veremos ambas creadas.



Continuaremos insertando datos en la tabla "cuentas" y activaremos "Auto commit".

```
INSERT INTO cuentas (n_cuenta, nombre, balance)  
VALUES(148, 'Pepe', 2000);
```



También utilizaremos "UPDATE" para modificar el atributo "balance" restando 137. Al ejecutar la consulta, observaremos que la operación de "UPDATE" se realizó correctamente.

```
INSERT INTO cuentas (n_cuenta, nombre, balance)
VALUES(148, 'Pepe', 2000);
UPDATE cuentas SET balance = balance - 137.0 WHERE nombre = 'Pepe'
```


```
38 INSERT INTO cuentas (n_cuenta, nombre, balance)
39 VALUES(148, 'Pepe', 2000);
40 UPDATE cuentas SET balance = balance - 137.0 WHERE nombre = 'Pepe';
41
42
```

Data Output **Messages** Notifications

UPDATE 1

Query returned successfully in 72 msec.

A continuación, procederemos a insertar datos en la tabla "producto". Ejecutaremos esta operación utilizando "BEGIN" al inicio de nuestra consulta y "COMMIT" al final, desactivando el "Auto Commit".



```
42 BEGIN;
43 INSERT INTO producto (id, nombre, precio)
44 VALUES(1, 'Zapato', 1000);
45
46 INSERT INTO producto (id, nombre, precio)
47 VALUES(2, 'Polera', 500);
48
49 INSERT INTO producto (id, nombre, precio)
50 VALUES(3, 'Pantalón', 700);
51 COMMIT;
```

Data Output **Messages** Notifications

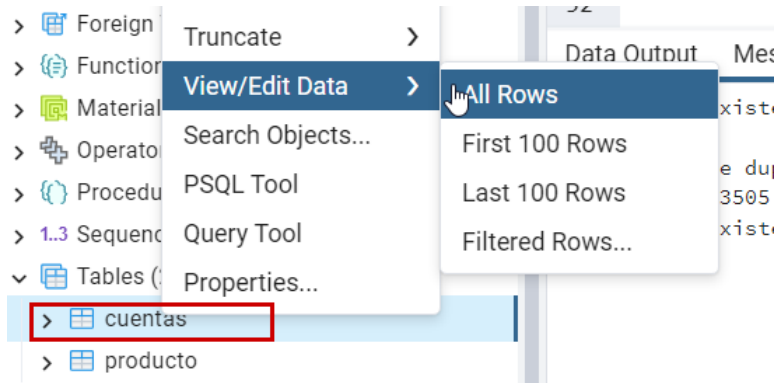
COMMIT

Query returned successfully in 58 msec.

Auto commit?

Auto rollback on error?

Posteriormente, revisaremos los datos ingresados en nuestras tablas. Para ello, haremos clic derecho sobre la tabla que deseamos consultar, seleccionaremos "View/Edit Data" y luego "All Rows".



De esta manera, podremos visualizar los datos de la tabla "cuentas". Repetiremos el mismo proceso para la tabla "productos".

Data Output			
Messages			
Notifications			
	n_cuenta [PK] numeric	nombre character varying (100)	balance real
1	148	Pepe	1863

Data Output			
Messages			
Notifications			
	id [PK] integer	nombre character varying (100)	precio smallint
1	1	Zapato	1000
2	2	Polera	500
3	3	Pantalón	700

Ahora procederemos a actualizar el precio del producto "Zapato" haciendo uso de "SAVEPOINT", lo cual nos permitirá conservar el estado de la transacción hasta este punto. Es importante destacar que el "SAVEPOINT" debe recibir un nombre, en este caso, lo llamaremos "sp".

```
BEGIN;  
UPDATE producto SET precio = precio + 100.25 WHERE nombre = 'Zapato';  
SAVEPOINT sp;
```

Realizaremos también la actualización del precio de la polera mediante la sentencia "UPDATE".










```
UPDATE producto SET precio = precio - 180.50  
WHERE nombre = 'Polera';
```

Para concluir, emplearemos la instrucción "ROLLBACK TO SAVEPOINT 'sp'" y cerraremos la transacción con "COMMIT". La estructura final de nuestra consulta se presenta a continuación:

```
BEGIN;  
  
UPDATE producto SET precio = precio + 100.25  
WHERE nombre = 'Zapato';  
SAVEPOINT sp;  
  
UPDATE producto SET precio = precio - 180.50  
WHERE nombre = 'Polera';  
ROLLBACK TO SAVEPOINT sp;  
COMMIT;
```

Después de ejecutar la consulta, examinaremos la tabla "producto". Observaremos que el precio de "Zapato" ha sido alterado, sumándole únicamente "100" a su valor original. Este comportamiento se debe a la "truncación" del número, ya que el tipo de dato de "precio" es "smallint", mientras que el valor añadido es de tipo "Real".

En cuanto a "Polera", notaremos que su "precio" permanece sin cambios. Esto se debe a la sentencia "ROLLBACK TO SAVEPOINT", que intenta revertir la transacción hasta el punto guardado "sp". Esta acción deshacerá únicamente las actualizaciones realizadas después de la creación del punto de guardado, en este caso, la segunda actualización.

Data Output				Messages	Notifications
        					
	id [PK] integer	nombre character varying (100)	precio smallint		
1	1	Zapato	1100		
2	2	Polera	500		
3	3	Pantalón	700		

EXERCISE 4: CONSISTENCIA Y AISLAMIENTO

Para comprender la propiedad de consistencia, a continuación, se presentan algunos ejemplos, los cuales enfatizan sobre la integridad de los datos que lleva a cabo PostgreSQL.

```
1 BEGIN;  
2 INSERT INTO public. "Clientes" VALUES  
3 ('CLI004', 'Nixon', 'El Cambio', '0981242156', 'INM003', 250);  
4 COMMIT;
```

PROPIEDAD DE CONSISTENCIA

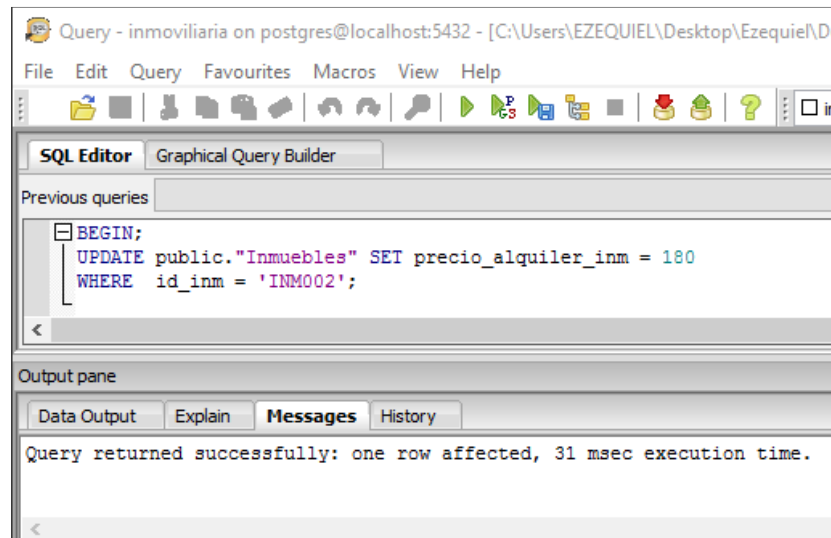
Al ejecutar la transacción anterior, se produce el siguiente resultado:

```
ERROR: llave duplicada viola restricción de unicidad «Clientes_pkey»  
DETAIL: Ya existe la llave (id_cli)=(CLI004).  
***** Error *****
```

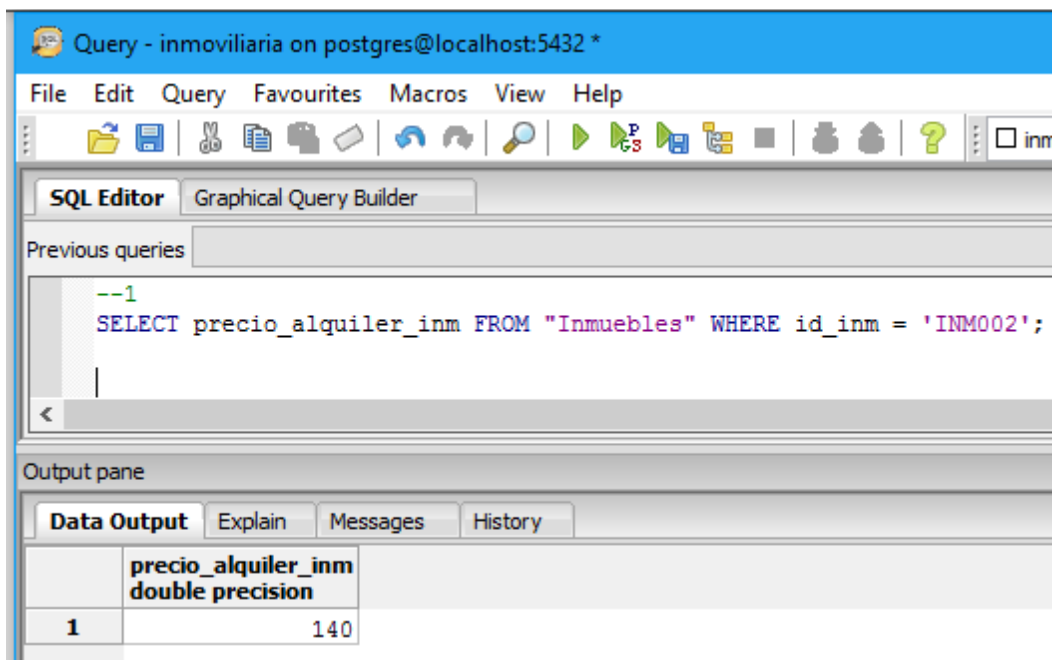
Esto ocurre debido a que ya existe un registro que contiene el valor de CLI004 como clave primaria. Con ello, se evidencia que PostgreSQL es un DBMS que controla la integridad de los datos, y por lo tanto, cumple con la propiedad de consistencia.

Reglas ACID: verificaremos la propiedad de aislamiento (los cambios en una transacción no terminada no se ven en otra sesión), utilizando la tabla "Contrato".

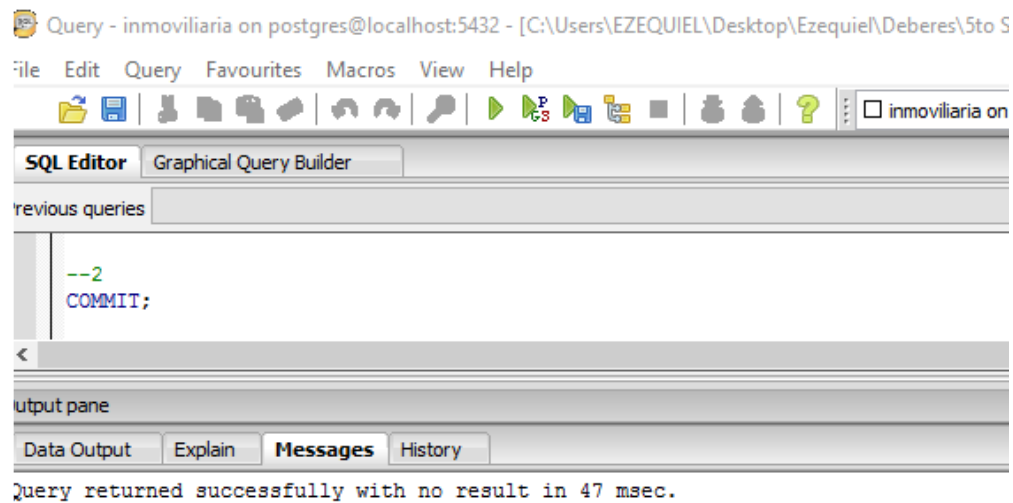
Para comprobar que PostgreSQL cumple con la propiedad de aislamiento, se abren dos ventanas para ejecutar consultas SQL. En la primera se llevarán a cabo los siguientes comandos, con lo que se indica que la consulta fue exitosa:



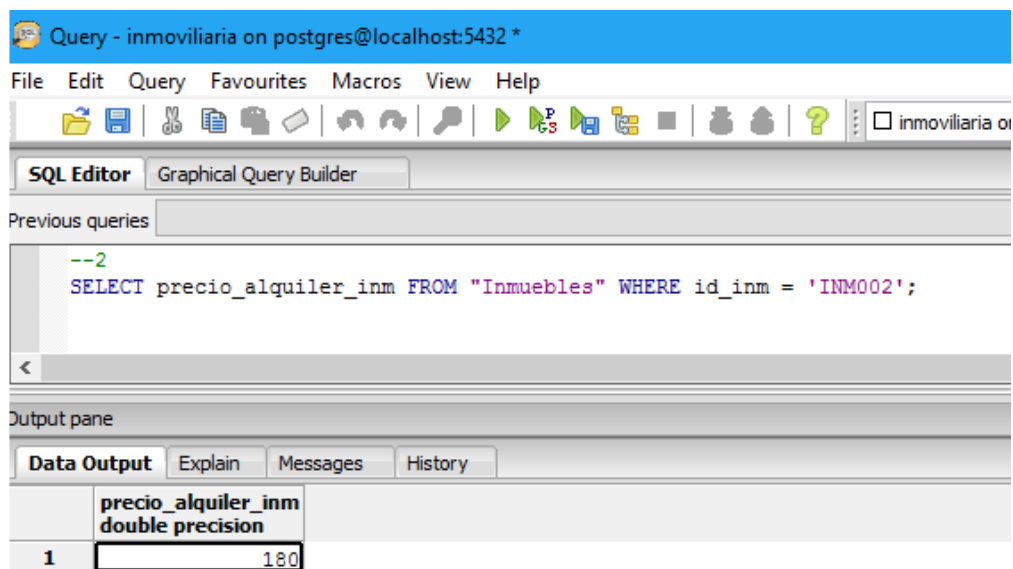
Si otro usuario deseara visualizar cierta información de la base de datos inmobiliaria, específicamente el precio de alquiler del inmueble cuyo código es INM002, se muestra el valor que estaba registrado en la base, y no el que se modificó en la ventana 1.



Al escribir el comando COMMIT en la ventana 1, y ejecutarlo, recién se estará guardando el registro en el disco duro.



Ahora, si se vuelve a generar la misma consulta en la ventana 2, se podrá visualizar que los cambios han sido almacenados en el disco duro.



EXERCISE 5: PRIVILEGIOS Y USUARIOS

GRUPOS DE USUARIOS

Esta propiedad nos permite agrupar a varios usuarios, con el objetivo de asignar privilegios de manera general para optimizar tiempo. Luego, podemos crear usuarios de manera independiente, y enlazarlos con algún grupo.

Sintaxis del grupo:

```
1 CREATE GROUP [nombregrupo]
```

Sintaxis del usuario:

```
1 CREATE USER [nombreusuario] WITH PASSWORD 'password' IN GROUP
2 [nombregrupo]
```

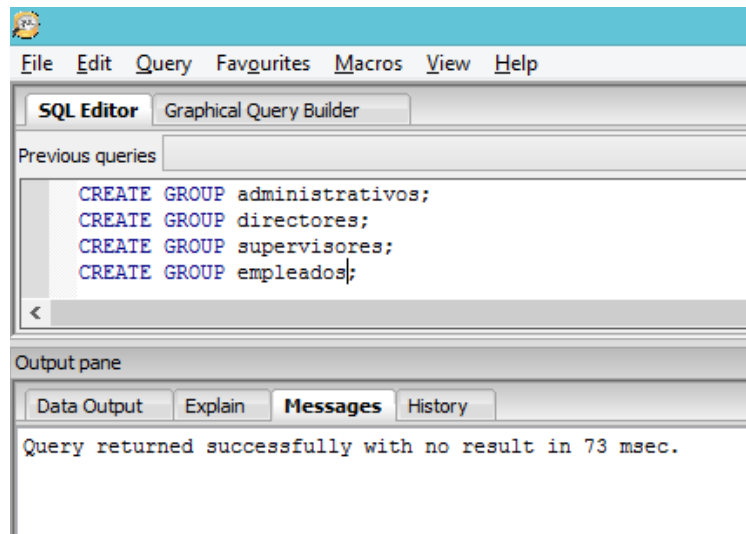
PRIVILEGIOS

Con la asignación de privilegios al usuario, se da la autorización a que éste, o un grupo, realice cualquier acción sobre una tabla específica. Dichas acciones pueden ser otorgadas con el comando "GRANT", o a su vez, eliminadas con el comando "REVOKE".

Su sintaxis es:

```
1 GRANT [SELECT, INSERT UPDATE, DELETE, ALL] ON [nombretabla] TO
2 [nombreusuario o nombreGrupo]
```

Bajo estos conceptos, aplicaremos lo mismo a la base de datos llamada inmobiliaria. Primero creamos los 4 grupos principales: Administrativo, Director, Supervisor, Empleado.



Luego, nos guiaremos en la siguiente matriz de trazabilidad, donde observamos qué acciones se han asignado a los respectivos grupos, en las diferentes tablas de nuestra base de datos.

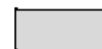
Tabla Matriz de Trazabilidad de los Usuarios.

	Administrativo	Director	Supervisor	Empleado
Personas				
Directores				
Empleados				
Pariente				
Oficinas				
Visitas				
Clientes				
Inspección				
Inmuebles				
Inmuebles - Factura				
Factura				
Periódico				
Publicidad				
Propietario				
Contrato				
Pago				

SELECT



ALL



Por lo tanto, quedaría así:

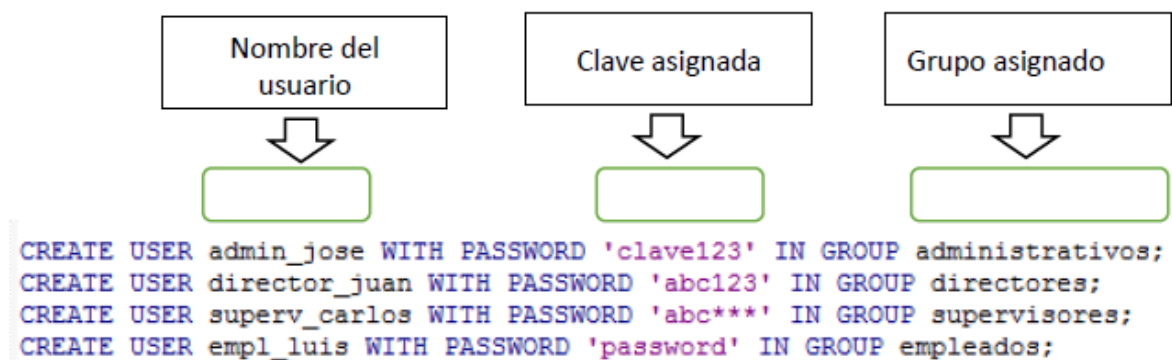
```
GRANT ALL ON "Personas" TO GROUP administrativos;
GRANT ALL ON "Directores" TO GROUP administrativos;
GRANT ALL ON "Empleados" TO GROUP administrativos;
GRANT ALL ON "Pariente" TO GROUP administrativos;
GRANT ALL ON "Oficinas" TO GROUP administrativos;
GRANT ALL ON "Visitas" TO GROUP administrativos;
GRANT ALL ON "Clientes" TO GROUP administrativos;
GRANT ALL ON "Inspeccion" TO GROUP administrativos;
GRANT ALL ON "Inmuebles" TO GROUP administrativos;
GRANT ALL ON "Inmueble_Factura" TO GROUP administrativos;
GRANT ALL ON "Facturas" TO GROUP administrativos;
GRANT ALL ON "Periodico" TO GROUP administrativos;
GRANT ALL ON "Publicidad" TO GROUP administrativos;
GRANT ALL ON "Propietario" TO GROUP administrativos;
GRANT ALL ON "Contrato" TO GROUP administrativos;
GRANT ALL ON "Pago" TO GROUP administrativos;
```

Output pane

Data Output Explain **Messages** History

Query returned successfully with no result in 23 msec.

Una vez generados estos grupos de usuarios, procedemos a crear a los respectivos usuarios, basándonos en la sintaxis presentada anteriormente. Se creará uno por grupo, y se podrán añadir más dependiendo las necesidades de la empresa.



Si ejecutamos la sentencia: `SELECT * FROM pg_shadow`, podremos ver como los usuarios han sido creados.



```
SELECT * from pg_shadow;
```

	username name	usesysid oid	usecreatedb boolean	usesuper boolean	userepl boolean	usebypassrls boolean	passwd text	valuntil abstime	useconfig text[]
7	admin_jose	25163	f	f	f	f	md5f88b89f6053da685eb10dc9987f93a0f		
8	director_juan	25164	f	f	f	f	md574a9f7b3165b009439ae89794a363bf5		
9	superv_carlos	25165	f	f	f	f	md53d90de7a08b80e7d61388bcd5c4a3424		
10	empl_luis	25166	f	f	f	f	md510bee7d2eddfce65b4c517f4c149570f		

Y si queremos modificar, por ejemplo, la contraseña, solo es necesario ejecutar la sentencia ALTER USER:

```
1 ALTER USER admin_jose WITH PASSWORD 'nueva_password';
```