Práctica 30

Disparadores

Objetivo

*En esta práctica se introduce el concepto de disparador (trigger) y se realizan diversos ejercicios para ver su aplicación y utilidad, en la consola de **psql**.*

Introducción

Un disparador es un objeto con nombre dentro de una base de datos el cual se asocia con una tabla y se activa cuando ocurre en ésta un evento en particular.

La sintaxis para crear disparadores es:

```
CREATE TRIGGER nombre momento evento ON tabla FOR EACH ROW
sentencia
```

El disparador queda asociado a la tabla *tabla*, la cual debe ser una tabla permanente. La sentencia CREATE TRIGGER necesita el privilegio SUPER

momento es el momento en que el disparador entra en acción y puede ser BEFORE (antes) o AFTER (después), para indicar que el disparador se ejecute antes o después del evento que lo activa.

evento indica la clase de sentencia que activa al disparador. Puede ser INSERT, UPDATE, o DELETE. Por ejemplo, un disparador BEFORE para sentencias INSERT podría utilizarse para validar los valores a insertar.

No puede haber dos disparadores en una misma tabla que correspondan al mismo momento y evento, por ejemplo, no se pueden tener dos disparadores BEFORE UPDATE. Pero sí es posible tener los disparadores BEFORE UPDATE y BEFORE INSERT o BEFORE UPDATE y AFTER UPDATE.

sentencia es la sentencia que se ejecuta cuando se activa el disparador. Si se desean ejecutar sentencias compuestas, deben colocarse entre BEGIN ... END, se pueden emplear los mismos tipos de sentencias permitidas en las rutinas almacenadas.

Las columnas de la tabla asociada con el disparador pueden referenciarse empleando los prefijos OLD y NEW.

- ***OLD.columna*** hace referencia al valor de una columna antes de ser actualizada o borrada.
- ***NEW.columna*** hace referencia al valor de una columna en una nueva fila a punto de ser insertada, o en una fila existente luego de ser actualizada.

Para eliminar un disparador, se emplea la sentencia ***DROP TRIGGER***.

Equipo necesario

Una computadora con sistema operativo ***Windows*** que cuente con el software ***psql***.

Metodología

1. Ejecute la herramienta ***SQL Shell (psql)***.
2. Deje el usuario ***postgres*** y proporcione la contraseña definida en la primera práctica.
3. Cambie a la base de datos ***pedidos***:

```
postgres=# \c pedidos
```

4. Escriba las siguientes sentencias para crear las tablas llamadas ***Cuenta*** y ***Suma***:

```
pedidos=# CREATE TABLE Cuenta (Folio INT PRIMARY KEY, Cantidad
DECIMAL(10,2), Fecha Date);
pedidos=# CREATE TABLE Suma (Suma NUMERIC);
```

5. Antes de definir un disparador, en ***postgreSQL***, es necesario definir la función que va a ejecutar el disparador cuando ocurra el evento previsto.
6. Entonces, escriba la siguiente función llamada ***Prueba***, que devuelve un tipo ***trigger***.

```
pedidos=# CREATE OR REPLACE FUNCTION Prueba() RETURNS TRIGGER
pedidos-# AS $$
pedidos$# BEGIN
pedidos$#   UPDATE Suma SET Suma = Suma + NEW.Cantidad;
pedidos$#   RETURN NEW;
pedidos$# END
pedidos$# $$
pedidos-# LANGUAGE plpgsql;
```

7. Esta función acumula en el atributo ***Suma*** de la tabla ***Suma*** el valor de la ***Cantidad*** indicado en el nuevo registro que se inserte en la tabla ***Cuenta***.
8. A continuación, va a crear un disparador que se active antes de hacer la inserción de un nuevo registro a la tabla ***Cuenta***, llamando la función ***Prueba*** definida previamente:

```
pedidos=# CREATE TRIGGER Sumatoria BEFORE INSERT ON Cuenta
pedidos-# FOR EACH ROW
pedidos-# EXECUTE PROCEDURE Prueba();
```

9. Obtenga una imagen completa de la pantalla, mostrando la creación de la función y el trigger, consérvela como evidencia.

10. Para utilizar el disparador, se debe agregar un registro con valor cero a la tabla Suma.

```
pedidos=# INSERT INTO Suma VALUES (0);
```

11. Ahora deberá ejecutar varias inserciones y al final consulte el valor que contiene la tabla ***Suma***:

```
pedidos=# INSERT INTO cuenta VALUES (137,14.98, current_date);
pedidos=# INSERT INTO cuenta VALUES (141,1937.50,'2023-11-12');
pedidos=# INSERT INTO cuenta VALUES (97,-100,NULL);
pedidos=# SELECT * FROM cuenta;
pedidos=# SELECT * FROM Suma;
```

12. Obtenga una imagen completa de la pantalla, mostrando la ejecución de estas sentencias, consérvela como evidencia.

13. Si intenta ingresar un registro con una llave duplicada, generará un error y como el registro no se insertó, el trigger no se ejecuta, quedando el mismo valor anterior, repita la última inserción:

```
pedidos=# INSERT INTO cuenta VALUES (97,-100,NULL);
pedidos=# SELECT * FROM Suma;
```

14. En el siguiente ejemplo se crea un disparador para verificar que los valores de las cantidades actualizadas en la tabla ***Cuenta*** se encuentren dentro de un rango de 0 a 500.

15. En caso de que no se cumpla, cambiará la cantidad para que quede dentro de este rango.

16. Como primer paso, escriba la siguiente función llamada ***Cambia***, que devuelve un tipo ***trigger***, el prefijo NEW se refiere al valor que se pretende actualizar y que puede ser cambiado por el disparador mediante la función, antes de ser modificado el registro.

```
pedidos=# CREATE OR REPLACE FUNCTION Cambia() RETURNS TRIGGER
pedidos-# AS $$
pedidos$# BEGIN
pedidos$#   IF NEW.Cantidad < 0 THEN
pedidos$#      NEW.Cantidad = 0;
pedidos$#   ELSEIF NEW.Cantidad >500 THEN
pedidos$#      NEW.Cantidad = 500;
pedidos$#   END IF;
pedidos$#   RETURN NEW;
pedidos$# END
```

```
pedidos$# $$
pedidos-# LANGUAGE plpgsql;
```

17. El disparador se debe definir BEFORE, porque los valores deben verificarse y cambiarse antes de actualizar el registro:

```
pedidos=# CREATE TRIGGER Checa BEFORE UPDATE ON Cuenta
pedidos-# FOR EACH ROW
pedidos-# EXECUTE PROCEDURE Cambia();
```

18. Obtenga una imagen completa de la pantalla, mostrando la creación de la función y el trigger, consérvela como evidencia.

19. Revise el contenido de la tabla:

```
pedidos=# SELECT * FROM Cuenta;
```

20. Ahora proceda a realizar varias actualizaciones:

```
pedidos=# UPDATE Cuenta SET Cantidad=55 WHERE Folio=137;
pedidos=# UPDATE Cuenta SET Cantidad=-55 WHERE Folio=97;
pedidos=# UPDATE Cuenta SET Cantidad=555 WHERE Folio=141;
```

21. Todas las actualizaciones procedieron, ahora revise nuevamente el contenido de la tabla:

```
pedidos=# SELECT * FROM Cuenta;
```

22. Obtenga una imagen completa de la pantalla, mostrando la ejecución de estas sentencias, consérvela como evidencia.

23. Para ser congruentes, deberá modificar al disparador ***Checa*** para incluir al evento ***INSERT*** además del evento ***UPDATE***, con el fin de evitar que se inserten registros que no cumplan con valores dentro del rango establecido.

24. Primeramente, deberá eliminar el disparador Checa anterior.

```
pedidos=# DROP TRIGGER Checa ON CUENTA;
```

25. El disparador se redefine para incluir ambos eventos:

```
pedidos=# CREATE TRIGGER Checa BEFORE UPDATE OR INSERT ON
Cuenta
pedidos-# FOR EACH ROW
pedidos-# EXECUTE PROCEDURE Cambia();
```

26. Obtenga una imagen completa de la pantalla, mostrando la creación del nuevo trigger, consérvela como evidencia.

27. Para probar el nuevo disparador inserte un nuevo registro:

```
pedidos=# INSERT INTO Cuenta VALUES (39,982,current_date);
```

28. Verifique que el valor insertado es 500, mismo que se acumula en la tabla ***Suma***, (considere que el acumulado previo no se modificó con los cambios hechos a los registros de cuenta).

```
pedidos=# SELECT * FROM Cuenta;
pedidos=# SELECT * FROM Suma;
```

29. Obtenga una imagen completa de la pantalla, mostrando la ejecución de estas sentencias, consérvela como evidencia.
30. Definiendo por separado a la función invocada por el *trigger*, permite crear disparadores tan complejos como se requiera.
31. En el siguiente ejercicio, se va a definir un disparador que modifique varias tablas simultáneamente.
32. Cree primero dos tablas sencillas:

```
pedidos=# CREATE TABLE Tallas (Talla INT PRIMARY KEY, Total INT
DEFAULT 0);
pedidos=# CREATE TABLE Colores (Color VARCHAR(10) PRIMARY KEY,
Total INT DEFAULT 0);
```

33. A continuación, ingrese ocho registros a cada una de las tablas creadas:

```
pedidos=# INSERT INTO Tallas (Talla) VALUES (28), (30), (32),
(34), (36), (38), (40), (42);
pedidos=# INSERT INTO Colores (Color) VALUES ('Rojo'),
('Verde'), ('Azul'), ('Negro'), ('Amarillo'), ('Rosa'),
('Blanco'), ('Beige');
```

34. Consulte los datos ingresados a esas tablas:

```
pedidos=# SELECT * FROM Tallas;
pedidos=# SELECT * FROM Colores;
```

35. Obtenga una imagen completa de la pantalla, mostrando la ejecución de estas sentencias, consérvela como evidencia.
36. Ahora cree otra tabla en las que se van a registrar los errores en productos inexistentes:

```
pedidos=# CREATE TABLE Errores (Error SERIAL PRIMARY KEY,
Motivo CHAR(20), Valor CHAR(10));
```

37. Como en los casos anteriores, se debe definir primero la función del disparador:

```
pedidos=# CREATE OR REPLACE FUNCTION Incrementa() RETURNS
TRIGGER
pedidos-# AS $$
pedidos$# BEGIN
pedidos$#   UPDATE Tallas SET Total=Total+NEW.Cantidad WHERE
Talla=NEW.Talla;
pedidos$#   UPDATE Colores SET Total=Total+NEW.Cantidad WHERE
Color= NEW.Color;
pedidos$#   UPDATE Producto SET Existencia=Existencia+
NEW.Cantidad WHERE
```

```
pedidos$#     Código=NEW.Producto AND Talla = NEW.Talla AND
Color = NEW.Color;
pedidos$#   RETURN NEW;
pedidos$# END
pedidos$# $$
pedidos-# LANGUAGE plpgsql;
```

38. Ahora se define el disparador ***Registro***, para el evento INSERT en el momento AFTER, de tal forma que se actualicen los totales de talla y color, una vez que el registro haya sido insertado en *Pedidos*:

```
pedidos=# CREATE TRIGGER Registro AFTER INSERT ON Pedidos
pedidos-# FOR EACH ROW
pedidos-# EXECUTE PROCEDURE Incrementa();
```

39. Obtenga una imagen completa de la pantalla, mostrando la creación de la función y del nuevo trigger, consérvela como evidencia.

40. En la tabla ***Pedidos***, inserte los siguientes seis registros:

```
pedidos=# INSERT INTO Pedidos VALUES (7,'2024-11-16',22,'MO-
21',34,'Negro',10), (8,'2024-11-17',12,'MO-21',30,'Verde',3),
(9,'2024-11-17',23,'MO-21',28,'Azul',8), (10,'2024-11-
18',20,'ME-14',34,'Verde',7), (11,'2024-11-18',20,'ME-14',40,
'Blanco',10), (12,'2024-11-18',20,'ME-14',36,'Rojo',11);
```

41. Verifique que se insertaron los registros correctamente en la tabla Pedidos:

```
pedidos=# SELECT * FROM Pedidos;
```

42. Además, consulte las tablas de tallas y colores:

```
pedidos=# SELECT * FROM Tallas;
pedidos=# SELECT * FROM Colores;
```

43. Obtenga una imagen completa de la pantalla, mostrando la ejecución de estas consultas, consérvela como evidencia.

44. Para evitar que se registre un pedido de una talla o color que no existan, es necesario escribir un procedimiento, que valide la información, y que, en caso dado, genere un registro en la tabla de errores.

```
pedidos=# CREATE OR REPLACE PROCEDURE Valida(Num INT, Fe DATE,
pedidos-#    So INT, Pr VARCHAR(6), Ta INT, Co VARCHAR(10),
Cant INT)
pedidos-# LANGUAGE plpgsql
pedidos-# AS $$
pedidos$# DECLARE
pedidos$#   K Boolean;
pedidos$# BEGIN
pedidos$#   K := (SELECT Count(*) FROM Tallas WHERE Talla=Ta);
pedidos$#   IF NOT K THEN
pedidos$#     INSERT INTO Errores (Motivo, Valor) VALUES
```

```
('Talla inexistente',
pedidos$#       Ta);
pedidos$#     RAISE NOTICE 'Talla inexistente';
pedidos$#   ELSE
pedidos$#     K := (SELECT Count(*) FROM Colores WHERE
Color=Co);
pedidos$#     IF NOT K THEN
pedidos$#       INSERT INTO Errores (Motivo, Valor) VALUES
pedidos$#        ('Color inválido', Co);
pedidos$#       RAISE NOTICE 'Color inexistente';
pedidos$#     ELSE
pedidos$#       INSERT INTO Pedidos VALUES (Num, Fe, So, Pr,
Ta, Co, Cant);
pedidos$#     END IF;
pedidos$#   END IF;
pedidos$# END;
pedidos$# $$;
```

45. Obtenga una imagen completa de la pantalla, mostrando la creación del procedimiento, consérvela como evidencia.

46. Ahora, ejecute una inserción en la tabla ***pedido*** mediante este procedimiento:

```
pedidos=# CALL Valida (13, current_date, 15,'ME-14', 30,
'Verde',15);
```

47. Puede ver que el pedido y las cantidades se registran normalmente en las tres tablas:

```
pedidos=# SELECT * FROM Pedidos;
pedidos=# SELECT * FROM Tallas;
pedidos=# SELECT * FROM Colores;
```

48. Ahora intente registrar una talla incorrecta y obtendrá un mensaje de advertencia.

```
pedidos=# CALL Valida (14, Current_date, 10, 'ME-14', 29,
'Verde', 10);
```

49. Y si intenta ingresar un color incorrecto, obtendrá otro mensaje de advertencia:

```
pedidos=# CALL Valida (14, Current_date, 10, 'ME-14', 30,
'Green', 10);
```

50. Finalmente, liste el contenido de la tabla errores:

```
pedidos=# SELECT * FROM Errores;
```

51. Obtenga una imagen completa de la pantalla, mostrando las llamadas al procedimiento y la tabla de errores, consérvela como evidencia.

52. Ahora, consulte la tabla ***Producto***, y podrá verificar que varios de los pedidos registrados (del 7 al 11) NO corresponden a productos registrados en esta

tabla.

```
pedidos=# SELECT * FROM Producto;
```

53. Modifique el procedimiento ***Valida***, agregando una condición ***IF*** inmediatamente antes del ***INSERT*** sobre la tabla ***Pedidos***, que inserte un nuevo registro en la tabla ***Producto*** con existencia ***CERO***, para cuando se haga un pedido de un producto que no esté registrado previamente.
54. Además, agregue una excepción para manejar cualquier error que pudiera ocurrir.
55. Obtenga una imagen completa de la pantalla, mostrando la modificación del procedimiento, consérvela como evidencia.
56. Hecho esto, ejecute la siguiente sentencia para registrar el pedido ***14*** mediante el procedimiento ***Valida*** y verificar que funciona correctamente.

```
pedidos=# CALL Valida (14, Current_date, 16,'MO-21', 30,
'Verde',25);
pedidos=# SELECT * FROM Producto;
```

57. Obtenga una imagen completa de la pantalla, mostrando la ejecución del procedimiento y la consulta de la tabla, consérvela como evidencia.
58. Cierre la consola de psql.
59. Fin de la Práctica

Evidencias

El alumno deberá enviar al instructor las evidencias requeridas durante la realización de la práctica.

Sugerencias didácticas

El instructor deberá atender a los alumnos que tengan dificultades en la interpretación y la realización de las instrucciones de la práctica.

Resultados

Se aprendió a crear disparadores con ***psql***, y analizar su comportamiento.

Bibliografía

- http://es.tldp.org/Postgresql-es/web/navegable/user/sql-createtrigger.html