Disparadores (TRIGGER)

1.- Definición: Un disparador (trigger en inglés) es un objeto de una base de datos que se ejecuta automáticamente cuando se ejecuta una sentencia **INSERT, UPDATE o DELETE** en una tabla en particular.

En otras palabras, un disparador no es otra cosa que una acción definida para una tabla de nuestra base de datos y que ejecuta automáticamente una función programada por el usuario.

Un disparador se puede definir de las siguientes maneras:

- Para que ocurra **ANTES** de cualquier **INSERT,UPDATE ó DELETE**.
- Para que ocurra **DESPUES** de cualquier **INSERT,UPDATE ó DELETE**.

2.-¿Cuándo se recomienda usar o construir un disparador?: Para comprender la importancia y forma de uso de un Tiggres partamos de un ejemplo: tenemos una tabla de existencias de productos de una organización que tiene una regla de negocio definida según la cual, cuando el stock mínimo de un producto sea menor o igual de cincuenta unidades, se debe hacer un pedido de cien unidades al proveedor.

Con los componentes vistos hasta ahora, se presentan dos posibles soluciones al problema, a saber:

• Añadir esta regla a todos los programas que actualizan las existencias de los productos.

• Hacer un programa adicional que haga un sondeo periódico de la tabla para comprobar la regla y realizar el pedido.

La primera solución tiene diversos inconvenientes: la regla de negocio está escondida, distribuida y reproducida dentro del código fuente de los diversos programas que conforman el sistema y, por lo tanto, es difícil de localizar y cambiar. Además, su eficacia o corrección depende del hecho de que cada programador inserte la regla en el lugar correcto.

La segunda solución, aunque la regla está concentrada en un solo sitio (el programa de sondeo), presenta los inconvenientes clásicos del sondeo: si se hace ocasionalmente, se puede perder el buen momento oportuno para su ejecución; y si en cambio se hace muy a menudo, se pierde eficiencia.

La solución correcta es: dotar al sistema de actividad o función; es decir, incorpora a la BD un nuevo componente, un disparador (triggers), que aplique la regla de negocio y se ejecute de forma automática cuando se cumpla la condición. Así, para el problema planteado (y sin entrar todavía en la sintaxis concreta), el disparador que se incorporaría a la BD es el siguiente: "Cuando se modifiquen las existencias de un producto y queden cincuenta unidades o menos, hay que pedir cien unidades nuevas". Como se puede ver, con esta solución se superan los inconvenientes de las dos soluciones vistas anteriormente: la regla de negocio está sólo en un sitio (en la BD), la eficacia no depende del programador y el disparador se ejecutará siempre que haga falta.

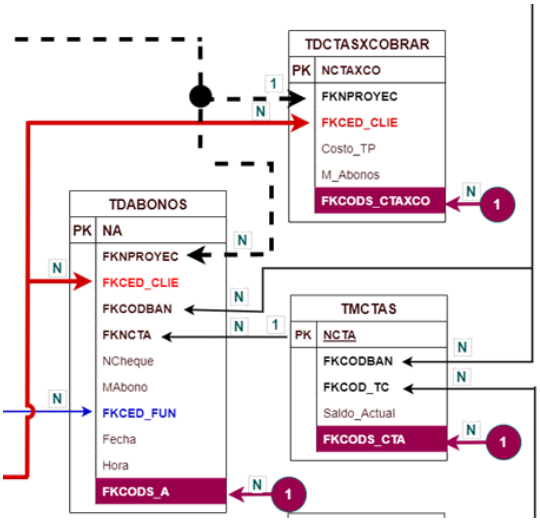
> Resumiendo, los disparadores son unos componentes que se ejecutan de una manera automática cuando se produce un evento (INSERT / UPDATE / DELETE) determinado. Se dice que un disparador es una regla ECA (**evento, condición, acción**): cuando se produce un evento determinado, si se cumple una condición, se ejecuta una acción.

En el ejemplo anterior, **el evento que activa el disparador** es la modificación de las existencias de algún producto**; la condición**, que las existencias quede o sema menor o igual a cincuenta unidades, y **la acción que se ejecuta**, ejecutar un proceso para pedir cien unidades nuevas

Se pude afirmar que se recomienda usar disparadores en los siguientes escenarios:

- Implementación de una regla de negocio. Como por ejemplo en el caso de las existencias mencionado anteriormente.

- Mantenimiento automático de una tabla de auditoría que lleve el registro de las actividades que se ejecutan en la BD. Se trata de registrar de una manera automática los cambios que se hacen en una determinada tabla.
- Mantenimiento automático de columnas derivadas. El valor de una columna derivada se calcula a partir del valor de otras columnas; posiblemente, de otras tablas. Cuando se modifican las columnas base, hay que recalcular de una manera automática el valor de la columna derivada. Por ejemplo: en un sistema donde mensualmente se registran los abonos de los clientes a un crédito (Ver TDAbonos - MAbono), y además en una tabla de cuentas por cobra (Ver TDCtasXCobrar – M_Abonos) se van acumulando los montos de los abonos del cliente



- Comprobación de restricciones de integridad no expresables en el sistema mediante CHECK o por medio de restricciones de integridad referencial. Como ejemplo: no se puede modificar un sueldo con un monto o valor menor al sueldo que actualmente está registrado.

3.-¿Cuándo NO se recomienda usar o construir un disparador?:

Si bien los disparadores, como hemos visto, pueden ser útiles a la hora de implementar determinadas situaciones del mundo real, no hay que abusar de ellos: no se deberían utilizar disparadores en las situaciones en las que el sistema se pudiera resolver con sus propios mecanismos. Así, por ejemplo, **no se tendrían que usar para implementar las restricciones de clave primaria**, **ni las restricciones de integridad referencial**, **ni todas las expresables como CHECK**, etc.

4.- Pasos para elaborar y ejecutar un trigger

4.1.- Crear la Función TRIGGER

```
CREATE OR REPLACE FUNCTION Function_Trigger(P1 Tipo_dato, ...., Pn Tipo_dato)
                                                    RETURNS TRIGGER AS $$
BEGIN
        Instrucción1 ;
        .
        InstrucciónN ;
        RETURN NEW ;
END ;
$$ LANGUAGE plpgsql ;
```

4.2.- Crear el Trigger: // BEFORE (Andes de) / AFTER (Después de)

```
CREATE OR REPLACE TRIGGER
        Tigger_Nombre   { BEFORE | AFTER }   { INSERT | UPDATE | DELETE }
                ON tabla FOR EACH { ROW | STATAMENT } -- Por cada línea / Por cada Sentencia
                        EXECUTE PROCEDURE Funcion_Trigger (P1, ...., Pn ) ;
```

Nota:

Los disparadores pueden ser **FOR EACH ROW** o **FOR EACH STATEMENT:**

- Si el disparador es **FOR EACH ROW**, el procedimiento asociado se ejecuta una vez por cada fila afectada por la sentencia que activa el disparador.
- Si el disparador es **FOR EACH STATEMENT**, el procedimiento se ejecuta una sola vez, independientemente del número de filas afectadas por la sentencia que activa el disparador.

Así, por ejemplo, en una operación de **DELETE** que afecte a diez filas de la tabla Tx; si el disparador es de tipo **FOR EACH ROW,** el disparador se ejecuta diez veces, una vez por cada fila borrada. Sin embargo, si el disparador es de tipo **FOR EACH STATEMENT**, el disparador en cuestión se ejecutará una sola vez (sin importar la cantidad de filas borradas); es decir ejecuta en lote.

4.3.- Ejecutar el comando INSERT | UPDATE | DELETE

```
=# INSERT / UPDATE / DELETE ….
```

5.- Ejemplos

5.1.- Ejemplo BEFORE (Antes) del UPDATE (Actualización)

```
CREATE TABLE  tmempleados_update(
        nr serial not null  ,
        ced varchar(12) not null  ,
        nom_viejo varchar(40) not null  ,
        fecha_update date not null  ,
        hora_update time not null  )  ;

CREATE OR REPLACE FUNCTION  ft_tmempleados_update()  RETURNS  TRIGGER  AS  $$
BEGIN
        INSERT INTO  tmempleados_update(ced, nom_viejo, fecha_update, hora_update)
                VALUES  (old.pkcedemple, old.nomemple, old.fecha, current_time )  ;

        -- SELECT current_time; ---> retorna la hora actual del servidor
        INSERT INTO tmempleados_update(ced, nom_viejo, fecha_update, hora_update)
                VALUES (new.pkcedemple,  new.nomemple, current_date, current_time )  ;

        RETURN new  ;
END  ;
$$ LANGUAGE plpgsql  ;
.
//BEFORE (Antes) del  UPDATE (Actualización)
CREATE  OR REPLACE  TRIGGER  trigger_update_tmempledos  BEFORE  UPDATE
            ON  tmempleados  FOR  EACH  ROW
                    EXECUTE PROCEDURE  ft_tmempleados_update()  ;

=# SELECT  *  FROM  tmempleados  ;
=# SELECT  *  FROM  tmempleados_update  ;
=# UPDATE  tmempleados  SET  nomemple ='GREISY'  WHERE  pkcedemple ='9000'  ;

=# SELECT  *  FROM  tmempleados  ;
=# SELECT  *  FROM  tmempleados_update  ;
```

5.2.- Ejemplo AFTER (Después) del INSERT (Insertar)

```
CREATE TABLE tmcargos_insert(
        nx serial not null,  codcar integer not null,  dcargo varchar(50) not null ,
        sueldo numeric(12,2) not null,  usuario varchar(50) not null,  fecha_insert date not null,
        hora_insert time not null ) ;

CREATE OR REPLACE FUNCTION ft_tmcargos_insert() RETURNS TRIGGER AS $$
DECLARE
        usuario varchar(50) := user ;
        fecha date := current_date ;
        tiempo time := current_time ;
BEGIN
        INSERT INTO tmcargos_insert(codcar, dcargo, sueldo, usuario, fecha_insert, hora_insert)
                VALUES (NEW.pkcodcar, NEW.dcargo, NEW.sueldo, usuario, fecha, tiempo) ;

        RETURN new ;
END ;
$$ LANGUAGE plpgsql ;
.
//AFTER (Después) del INSERT (Insertar)
CREATE OR REPLACE TRIGGER trigger_insert_tmcargos AFTER INSERT
            ON tmcargos FOR EACH ROW
                    EXECUTE PROCEDURE ft_tmcargos_insert() ;

=# SELECT * FROM tmcargos ;
=# SELECT * FROM tmcargos_insert ;
=# INSERT INTO tmcargos(dcargo, sueldo, fkcods) VALUES
                    ('Sub-Vendedor', 800000.05, 1) , ('Obrero 1', 750000.05, 1),
                    ('Obrero 2', 730000.05, 1) , ('Obrero 3', 720000.05, 0) ;

=# SELECT * FROM tmcargos ;
=# SELECT * FROM tmcargos_insert ;
```

5.3.- Ejemplo Validación. BEFORE (Antes) del INSERT (Insertar): Validar que al insertar un registro en la tmcargos: el nombre del cargo no se repita y sueldo sea < 1'000'000

```
CREATE OR REPLACE FUNCTION ft_insert_validar() RETURNS TRIGGER AS $$
DECLARE
        xdcargo   varchar(30) ;
        sw integer := 1   ;
BEGIN
        IF (TG_OP = 'INSERT')   THEN
                SELECT dcargo INTO xdcargo FROM tmcargos  WHERE dcargo = NEW.dcargo   ;

                -- SELECT INTO Crea una nueva tabla a partir del resultado de una query.
                --Típicamente esta query recupera los datos de una tabla existente,

                -- UPPER() ------> Transforma el texto a mayúsculas LOWER() --→ Minúsculas
                -- INITCAP() ----> Transforma la primera letra del texto a Mayúscula

                New.dcargo = UPPER(New.dcargo)  ;
                IF (UPPER(xdcargo) = NEW.dcargo   OR   NEW.sueldo >= 1000000) THEN
                        sw := 0   ;
                        RAISE EXCEPTION
                        'El cargo: %  ya existe o   El sueldo: % es muy grande ....Insercion Imposible..',
                        UPPER(xdcargo), to_char( NEW.sueldo, '999G999G999D99' )  ;
                END IF   ;
        END IF   ;
        RETURN NEW   ;
END  ;
$$ LANGUAGE plpgsql   ;
```

```
//BEFORE (Antes )  del  INSERT (Insertar)
CREATE OR  REPLACE TRIGGER  Trigger_validar   BEFORE   INSERT
        ON   tmcargos    FOR EACH ROW
                EXECUTE PROCEDURE  ft_insert_validar ()  ;

=# SELECT  *  FROM    tmcargos  ;
=# INSERT  INTO    tmcargos(dcargo, sueldo, fkcods)  values  ('COORDINADOR', 2000000, 0)  ;
=# SELECT  *  FROM    tmcargos  ;
```

VARIABLES ESPECIALES

A.- NEW (*): Variable de Tipo RECORD para las operaciones INSERT/UPDATE,la cual contiene los datos de la tupla antes de la ejecución de la operación. No se recomienda usarlo con los disparadores del tipo FOR EACH STATEMENT, pues retornar NULL

B.- OLD (*): Variable de Tipo RECORD para las operaciones UPDATE/DELETE, la cual contiene los datos de la tupla después de la ejecución de la operación. No se recomienda usarlo con los disparadores del tipo FOR EACH STATEMENT, pues retornar NULL

C.- TG_OP (*): Retorna una cadena de texto con el valor INSERT, UPDATE o DELETE; es decir, retorna la operación que activo el disparador.

D.- TG_NARGS (*): Retorna el número de parámetros dados al procedimiento en la sentencia CREATE TRIGGER. (Valor de tipo integer)

E.- TG_ARGV[] (*): Es una variable de Tipo text array, la cual almacena los parámetros de la sentencia CREATE TRIGGER(). Así, el índice empieza en 0. Los Índices inválidos son: menores a 0 ó mayores/iguales que tg_nargs -→ Implican o resultan en valores nulos.

F.- current_user (*): Almacena el nombre del usuario que está conectado actualmente a la base de datos y que está ejecutando las sentencias.

G.- current_date (*): Almacena la fecha actual del servidor (no la del cliente)

H.- current_time (*): Almacena la hora actual del servidor.

5.4.- Mantenimiento Automático

El objetivo del disparador es mantener de una manera automática una tabla llamada auditoría que contiene un registro de todas las modificaciones a la columna cant que hacen los usuarios en la tabla ítems

```
CREATE TABLE Auditoria(
        item integer,      username char(8),
        hora_modif timestamp,    cant_vieja integer,
        cant_nueva integer)  ;
```

```
CREATE TABLE items(
        item integer primary key,
        nom char(25),
        cant integer,
        precio_total decimal(9,2))  ;


CREATE FUNCTION insertar_auditoria() RETURNS trigger AS $$
BEGIN
        INSERT INTO auditoria VALUES
                (OLD.item,current_user,current_date,OLD.cant,NEW.cant)  ;
        RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER     items_auditoria    AFTER UPDATE
                OF   cant   ON   items
                        FOR   EACH   ROW   EXECUTE   PROCEDURE   insertar_auditoria();


INSERT INTO items(item, nom, cant, precio_total) VALUES
                (1,  'sacapunta', 100, 0.30),
                (2,  'boligrafo', 5000, 0.50),
                (3,  'borrador', 500, 0.60);
```

```
bdretardados=# select * from items;
 item |            nom            | cant | precio_total
------+---------------------------+------+--------------
    1 | sacapunta                 |  100 |         0.30
    2 | boligrafo                 | 5000 |         0.50
    3 | borrador                  |  500 |         0.60
(3 filas)
```

```
=# UPDATE items SET cant=cant+10 WHERE item<>3;

=# SELECT  *  FROM  auditoria ;

=# SELECT  *  FROM   items ;
```

```
bdretardados=# UPDATE items SET cant=cant+10 WHERE
bdretardados-# item<>3;
UPDATE 2
bdretardados=# select * from auditoria;
 item | username |     hora_modif      | cant_vieja | cant_nueva
------+----------+---------------------+------------+------------
    1 | postgres | 2025-10-15 00:00:00 |        100 |        110
    2 | postgres | 2025-10-15 00:00:00 |       5000 |       5010
(2 filas)


bdretardados=# select * from items;
 item |            nom            | cant | precio_total
------+---------------------------+------+--------------
    3 | borrador                  |  500 |         0.60
    1 | sacapunta                 |  110 |         0.30
    2 | boligrafo                 | 5010 |         0.50
(3 filas)
```

5.5.- Mantenimiento automático de un atributo derivado

En este ejemplo se muestra cómo se utilizan los disparadores para mantener calculado de una manera automática el atributo derivado **precio_total** de la tabla **ítems** cuando se producen modificaciones de la cantidad de existencias de algún ítem.

El valor del atributo derivado se calcula a partir del valor de otras columnas de la misma tabla o bien de otras tablas. Por lo tanto, cuando se modifican las columnas que se utilizan para calcular el atributo derivado, hay que recalcular de forma automática el valor del atributo derivado

Partamos nuevamente de la siguiente tabla items:

```
CREATE TABLE items(
        item integer primary key,
        nom char(25),
        cant integer,
        precio_total decimal(9,2))  ;

CREATE FUNCTION calcula_nuevo_precio_total()   RETURNS trigger AS $$
BEGIN
        IF (old.cant<>0) then
                NEW.precio_total=((OLD.precio_total/OLD.cant)*NEW.cant);
        END IF;
        RETURN NEW;
END
$$ LANGUAGE plpgsql;
```

```
bdretardados=# select * from items;
 item |           nom            | cant | precio_total
------+---------------------------+------+--------------
    1 | sacapunta                 |  110 |         0.30
    2 | boligrafo                 | 5010 |         0.50
    3 | borrador                  |  500 |        20.00
(3 filas)


bdretardados=# UPDATE items SET cant = 20 WHERE item=1;
UPDATE 1
bdretardados=# select * from items;
 item |           nom            | cant | precio_total
------+---------------------------+------+--------------
    2 | boligrafo                 | 5010 |         0.50
    3 | borrador                  |  500 |        20.00
    1 | sacapunta                 |   20 |         0.05
(3 filas)
```

```
CREATE TRIGGER atributo_derivado  BEFORE UPDATE
OF  cant  ON  items  FOR EACH ROW
EXECUTE PROCEDURE calcula_nuevo_precio_total()  ;

=# SELECT  *  FROM  items  ;

=# UPDATE  items  SET  cant  =  20  WHERE  item  =  1  ;

=# SELECT  *  FROM  items ;
```

6.- Listar los trigger de una Tabla

=# \d <nombre_tabla> -→ Muestra la estructura de la tabla + los tiggers asociados
=# \df ----> muestra el listado de las funciones

=# **\d tmcargos**
=# **\df**

Listar triggers usando consultas SQL

También se puede obtener la información de las tablas del catálogo de la vista **information_schema.triggers**:.

```
SELECT      trigger_name,
            event_manipulation,
            action_timing,
            event_object_table
FROM    information_schema.triggers
WHERE   event_object_table  =  'tmcargos'  ;
```

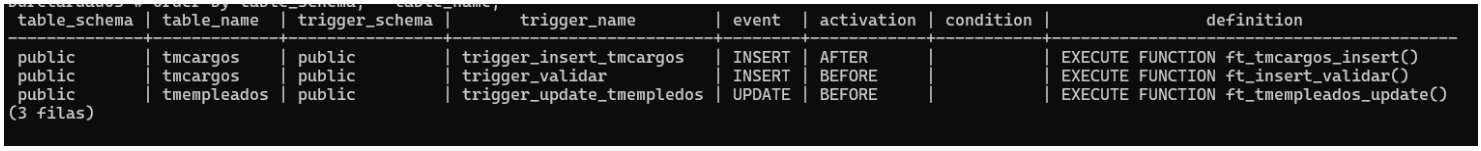
Esta consulta te proporcionará el **nombre del trigger**, el **tipo de evento que lo dispara** (INSERT, UPDATE, DELETE), **el momento en que se ejecuta** (BEFORE o AFTER) y **el nombre de la tabla a la que está asociado**.

```
bdretardados=#
bdretardados=# SELECT   trigger_name,
bdretardados-# event_manipulation,
bdretardados-# action_timing,
bdretardados-# event_object_table
bdretardados-# FROM   information_schema.triggers
bdretardados-# WHERE event_object_table = 'tmcargos'  ;
      trigger_name       | event_manipulation | action_timing | event_object_table
-------------------------+--------------------+---------------+--------------------
 trigger_insert_tmcargos | INSERT             | AFTER         | tmcargos
 trigger_validar         | INSERT             | BEFORE        | tmcargos
(2 filas)
```

Ahora bien, esta vista **information_schema.triggers** es más flexible, en caso que se necesite filtrar por otras condiciones o si quieres ver **triggers** de múltiples tablas en una sola consulta.
Sintaxis general de la vista del catálogo:

```
select event_object_schema as table_schema,
        event_object_table as table_name,
        trigger_schema,
        trigger_name,
        string_agg(event_manipulation, ',') as event,
        action_timing as activation,
        action_condition as condition,
        action_statement as definition
from information_schema.triggers
group by 1,2,3,4,6,7,8
order by table_schema,   table_name;
```

Aquí, se observan todos los tiggres de la base de datos ordenados por nombre de esquema, nombre de tabla (Ordered by schema name, table name)



table_schema	table_name	trigger_schema	trigger_name	event	activation	condition	definition
public	tmcargos	public	trigger_insert_tmcargos	INSERT	AFTER		EXECUTE FUNCTION ft_tmcargos_insert()
public	tmcargos	public	trigger_validar	INSERT	BEFORE		EXECUTE FUNCTION ft_insert_validar()
public	tmempleados	public	trigger_update_tmempledos	UPDATE	BEFORE		EXECUTE FUNCTION ft_tmempleados_update()

(3 filas)

Descripción de las columnas
table_schema - nombre del esquema de la tabla (name of the table schema)
table_name - nombre de la tabla de activación (name of the trigger table)
trigger_schema - nombre del esquema de activación (name of the trigger schema)
trigger_name - nombre del activador (name of the trigger)
event - operación SQL específica (specific SQL operation): Insert, Update or Delete
activation - activación del gatillo (trigger activation) time: After(Después de), Instead of or BEFORE (Antes de) condition - desencadenar la condición de activación (trigger activation condition)
definition - definition of trigger - in postgreSQL it is always EXECUTE PROCEDURE function_name()

Además, puede ver el código fuente de la función del **trigger** usando:

=# \sf nombre_de_la_funcion_trigger

=# **\sf ft_tmcargos_insert()**

```
bdretardados=# \sf ft_tmcargos_insert()
CREATE OR REPLACE FUNCTION public.ft_tmcargos_insert()
 RETURNS trigger
 LANGUAGE plpgsql
AS $function$
DECLARE
usuario varchar(50) := user   ;
fecha date := current_date   ;
tiempo time := current_time   ;
BEGIN
INSERT INTO tmcargos_insert(codcar, dcargo, sueldo, usuario, fecha_insert, hora_insert)
VALUES (NEW.pkcodcar,   NEW.dcargo,  NEW.sueldo,   usuario,   fecha,   tiempo)   ;

RETURN   new   ;
END   ;
$function$
```

7.- Borrar un trigger

=# **DROP TRIGGER** [IF EXISTS] <Nombre_Trigger> **ON** <Nombre_table> [**CASCADE | RESTRICT**] **;**

CASCADE ---> Eliminará todos los objetos que dependen del disparador.

RESTRICT --> No se eliminarán los objetos que dependen del disparador.

Ejemplo:

=# **DROP TRIGGER IF EXISTS** trigger_insert_tmcargos **ON** tmcargos CASCADE **;**

=# **DROP TRIGGER IF EXISTS** trigger_update_tmempledos **ON** tmempleados CASCADE **;**

8.- Caracterísitsticas o reglas a tener presente

8.1. La función Tigger que va a utilizar el disparador se debe de crear antes de construir el propio disparador.
8.2. La función Tigger debe retornar un valor de tipo "trigger"
8.3. La función Tigger puede ser utiliza por múltiples disparadores en diferentes tablas.
8.4. Si una tabla tiene más de un disparador definido para un mismo evento (INSERT,UPDATE,DELETE), estos se ejecutarán en orden alfabético por el nombre del disparador.
8.5. Las Funciones Tigger almacenados utilizados por disparadores pueden ejecutar sentencias SQL que a su vez pueden activar otros disparadores. Esto se conoce como disparadores en cascada. No existe límite para el número de disparadores que se pueden llamar pero es responsabilidad del programador el evitar una recursión infinita de llamadas en la que un disparador se llame asi mismo de manera recursiva.
8.6. El uso de disparadores de manera incorrecta ó inefectiva puede afectar significativamente al rendimiento de nuestra base de datos.

Fuente

1.- https://e-mc2.net/blog/disparadores-triggers-en-postgresql/

2.- https://platzi.com/tutoriales/1480-postgresql/6820-creacion-de-triggers-postegresql/

3.- Canal ----> TecnoBinaria / @Tecnobinaria
https://www.youtube.com/channel/UCJeVpLbYfAHivHHrwG2TqFw
3.1.- Curso PostegreSQL
https://www.youtube.com/watch?v=jxIEDKzGrOs&list=PL8gxzfBmzgezMknCY9-mnOXYoVuV2YYXO
3.1.A.- Como hacer un "disparador" y su función - TRIGGER | PostgreSQL #34
https://www.youtube.com/watch?v=KT6TJ7NChcg
3.1.B.- Como usar la clausula After para el Insert - TRIGGER | PostgreSQL #35
https://www.youtube.com/watch?v=bxZM2fqo0wk

4.- Canal ---> Tecno Computo e Informática.
4.1.- Curso PostegreSQL https://www.youtube.com/watch?v=5vtKKcWdHDM&list=PLg-j_rhz94Y_40Z-N5QxvE_AXDeGAwu6Y
4.1.A.- Como Hacer un Disparador y su Función TRIGGER PostgreSQL #33
https://www.youtube.com/watch?v=bDfjvvlaL4w
4.1.B.- Como usar la clausula After para el Insert TRIGGER PostgreSQL #34
https://www.youtube.com/watch?v=taarBAksHUQ

5.-Canal ---> Denisse VM (Curso con 29 Videos)
https://www.youtube.com/watch?v=U5XEcwaGVP8&list=PLU-N6yYDZ0LISy093Gd-DJYlGsmBFI73g
5.1.- TRIGGER (Disparadores)
5.1.A.- TBD. Ejercicio - Triggers (disparadores), procedimientos almacenados en PostgreSQL
https://www.youtube.com/watch?v=t_PylR9UI3I

6.- Canal ---> Bases de Datos (Ver Tema 3) ******
6.1.- 01 PLPGSQL | Procedimientos Almacenados | PostgreSQL
https://www.youtube.com/watch?v=It3tHF9EZfk&list=PLhnOQ_ypZS5ViZsK28otsfImA56Gm0lG_
6.1.- 02 PLPGSQL | Funciones que devuelven tablas | PostgreSQL
https://www.youtube.com/watch?v=WOsjB0zjryk&list=PLhnOQ_ypZS5ViZsK28otsfImA56Gm0lG_&index=2
6.1.- 03 PLPGSQL | Disparadores | PostgreSQL
https://www.youtube.com/watch?v=gesNyPypgek&list=PLhnOQ_ypZS5ViZsK28otsfImA56Gm0lG_&index=3
6.2.- 01 Lenguaje de Consulta Estructurado (SQL) | Conceptos básicos (Curso 16 videos)
https://www.youtube.com/watch?v=HlRhFDY1Bmg&list=PLhnOQ_ypZS5W8r7a2C_Rnl2fEcpSmEVvg

7.- Canal ---> Jesús Domínguez Gutú (Base de datos para aplicacines)
https://www.youtube.com/watch?v=OJPQzJCzUCk&list=PLC30JtWgCnDhTytR1rdg4W_99tig0mUrl

8.- Canal ---> TecnoBinaria (68 Videos) / SQL Server 2008, 2012, 2014 y 2016 | Windows 7, 8, 8.1 y 10
https://www.youtube.com/watch?v=Y-s4OFFwxcs&list=PL8gxzfBmzgey8iTCCgnViozTP8NlXKJpu

https://www.youtube.com/watch?v=XxZpIp7TCSM

9.- Canal ----> pildorasinformaticas (25 Videos)
9.1.- Curso SQL https://www.youtube.com/watch?v=iOiyJgnN71c&list=PLU8oAlHdN5Bmx-LChV4K3MbHrpZKefNwn
9.1.A.- Curso SQL. Triggers I (Disparadores). Vídeo 20
https://www.youtube.com/watch?v=kDu_5F159QA
9.1.B.- Curso SQL. Triggers II Disparadores. Vídeo 21
https://www.youtube.com/watch?v=bGQbgejFyBo&list=PLU8oAlHdN5Bmx-LChV4K3MbHrpZKefNwn&index=21
9.1.C.- Curso SQL. Triggers III Disparadores. Vídeo 22
https://www.youtube.com/watch?v=5sLU-drd8P4&list=PLU8oAlHdN5Bmx-LChV4K3MbHrpZKefNwn&index=22
10.- Web
10.1.- https://www.postgresqltutorial.com/
10.2.- https://e-mc2.net/blog/disparadores-triggers-en-postgresql/

DOCUMENTOS
https://sqltemuco.wordpress.com/2017/05/17/triggers-con-postgresql/
https://es.slideshare.net/AndresOrtega17/postgres-trigger
https://www.postgresqltutorial.com/postgresql-triggers/creating-first-trigger-postgresql/
https://www.dongee.com/tutoriales/como-funciona-un-trigger-en-sql-un-tutorial-paso-a-paso/
https://www.w3resource.com/PostgreSQL/postgresql-triggers.php
GUIA FUNCIONES / DISPARADORES --> https://uneweb.edu.ve/tuto-docs/guia-postgres-II.pdf
03 PLPGSQL | Disparadores | PostgreSQL
https://www.youtube.com/watch?v=gesNyPypgek&list=PLhnOQ_ypZS5ViZsK28otsfImA56Gm0lG_&index=3