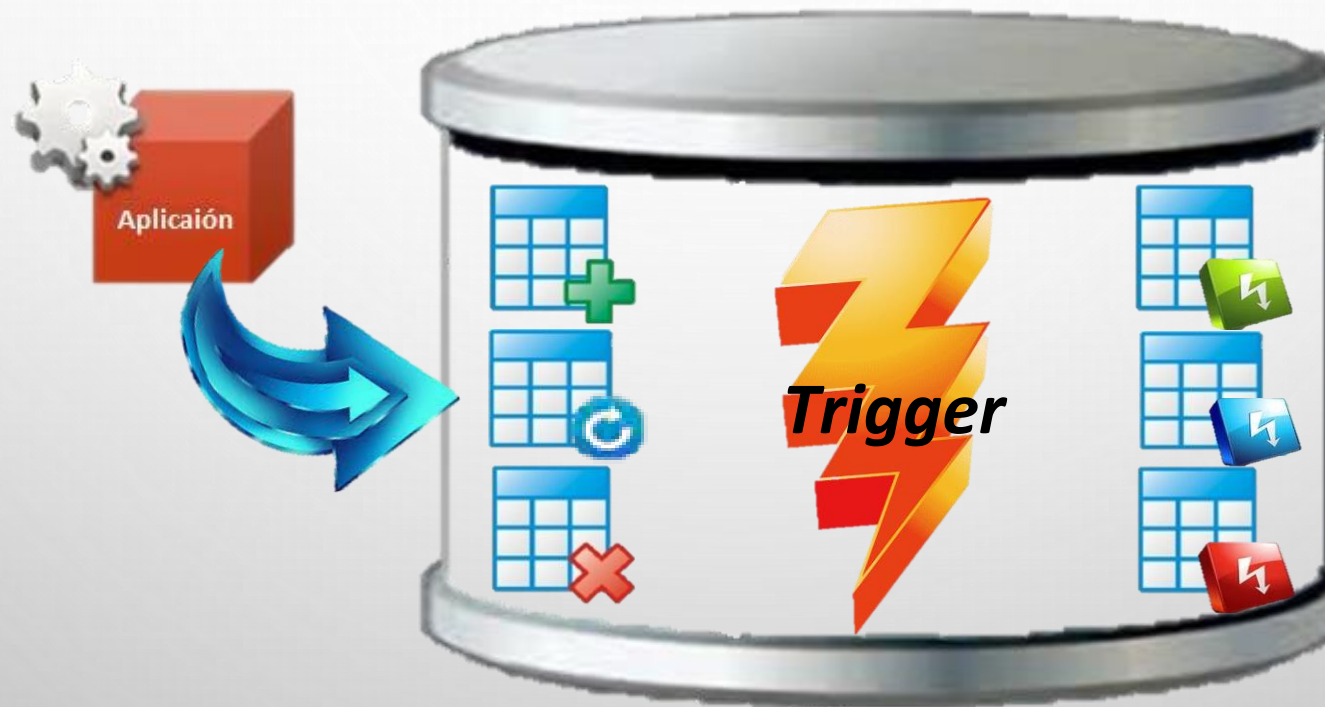


The background is a light gray gradient. It is decorated with several realistic water droplets of various sizes, some with highlights and shadows, scattered across the frame. In the upper center, there is a faint, circular logo or watermark that appears to be a university crest or seal.

# Conceptos de Bases de Datos II

*Triggers*

# Triggers



# Triggers

Durante la ejecución de una aplicación de base de datos, en ocasiones se requiere realizar una o más acciones de forma automática, si se produce un evento específico, es decir, que la primera acción provoca la ejecución de las siguientes acciones.

Los denominados Triggers o disparadores son el mecanismo de activación, que posee SQL para entregar esta capacidad.

Un **trigger** (o disparador) es una función asociada a una tabla que se ejecuta cuando se realiza una operación básica (**insert**, **update**, **delete** o **truncate**) sobre ésta.

# Triggers – Usos Comunes

Se usan para mejorar la administración de la Base de datos, sin la necesidad de que un usuario ejecute la sentencia de SQL.

Pueden generar valores de columnas, previene errores de datos, sincroniza tablas, modifica valores de una vista, etc., también se usa para auditorias entre otras cosas.

- Validación de datos
- Hacer modificaciones en cascada sobre tablas relacionadas
- Actualización de datos derivados
- Implementación de reglas de negocio complejas
- Mantenimiento de la integridad referencial
- Auditorias de cambios

# Triggers – Usos Comunes

**Validación de datos:** Los triggers se utilizan para validar los datos antes de que se inserten, actualicen o eliminen en una tabla. Pueden aplicar reglas de validación personalizadas, como verificar si se cumplen ciertas condiciones o asegurarse de que los datos cumplan ciertos criterios antes de permitir su modificación.

**Hacer modificaciones en cascada sobre tablas relacionadas:** Esto se logra invocando acciones adicionales en respuesta a eventos de inserción, actualización o eliminación en una tabla principal, lo que desencadena cambios en las tablas relacionadas. Ejemplo, cada cliente puede tener múltiples pedidos y queremos asegurarnos de que cuando se elimine un cliente, todos sus pedidos también se eliminen. Para lograr esto, podemos utilizar un trigger de eliminación en cascada.

**Actualización de datos derivados:** Los triggers se pueden utilizar para actualizar automáticamente datos derivados o calcular valores basados en los datos de una tabla. Por ejemplo, se puede crear un trigger que actualice un campo de totalización cada vez que se inserte, actualice o elimine un registro en una tabla.

# Triggers – Usos Comunes

**Implementación de reglas de negocio complejas:** Los triggers permiten implementar lógica de negocio personalizada en la base de datos. Pueden ejecutar una serie de acciones en respuesta a un evento, como enviar notificaciones, ejecutar cálculos complejos o realizar operaciones adicionales en otras tablas.

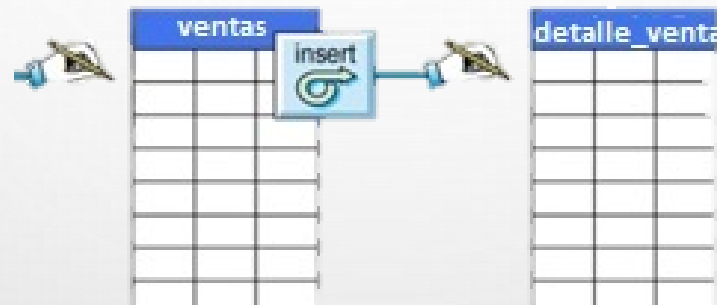
**Mantenimiento de la integridad referencial:** Los triggers se pueden utilizar para garantizar la integridad referencial entre tablas. Por ejemplo, se puede crear un trigger que elimine registros en una tabla si hay registros relacionados en otra tabla.

**Auditoría de cambios:** Los triggers se pueden utilizar para llevar un registro de los cambios realizados en una tabla. Por ejemplo, se pueden crear triggers que registren quién realizó una modificación, cuándo se realizó y qué datos se modificaron. Esto es útil para el seguimiento de cambios y para fines de auditoría.

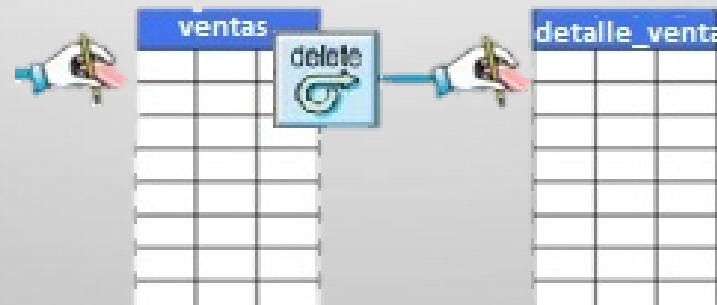
# Triggers – Usos Comunes

Ejemplo:

cuando se agrega un registro a la tabla ventas puede activar una sentencia que agregue registros a la tabla detalles\_ventas, por ejemplo, agregar los productos que intervienen en la venta.



Del mismo modo, al eliminar una fila en la tabla de ventas puede activar la acción para eliminar el detalle de dicha venta.





# Triggers

La operación que activa al trigger es conocida como sentencia desencadenante o disparadora. La ejecución del trigger puede ocurrir antes o después de llevada a cabo la sentencia disparadora.

Un Trigger es una orden que el sistema ejecuta de manera automática como efecto de alguna modificación en la base de datos. Se activan mediante los comandos **insert**, **delete**, **update** o **truncate** y siguen en orden la secuencia:

Evento → Condición → Acción.

Para diseñar un Trigger hay que tener en cuenta:

**Especificar condiciones para ejecutar el Trigger:** Se debe definir un evento (**insert**, **delete**, **update** o **truncate** ) que causa la comprobación del Trigger y una condición (puede aplicarse a una o mas columnas) que se debe cumplir para ejecutarlo.

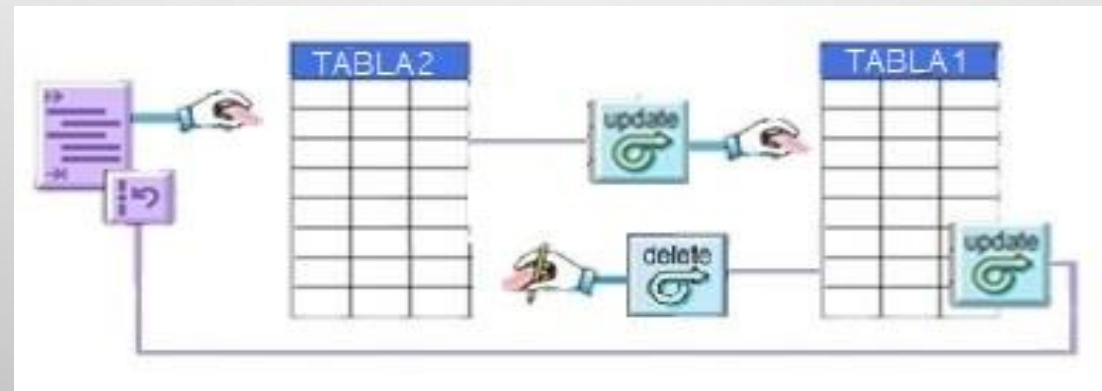
**Especificar las acciones:** Se debe precisar qué acciones se van a realizar cuando se ejecute el Trigger.



# Triggers

Los Triggers se almacena en la base de datos, por lo que son accesibles y persistentes para todas las operaciones de la misma. El SGBD asume la responsabilidad de ejecutarlo cada vez que ocurra el evento especificado y se satisfaga la condición.

Hay que evitar crear disparadores recursivos. Por ejemplo, el crear un trigger que se active después de borra un registro en la tabla2 que modifica la tabla1, que a su vez realiza un borrado en la tabla2, provoca una ejecución recursiva del disparador que agota la memoria.



# Triggers

```
create trigger trigger_name { before | after }  
  { event [ or ... ] } on table_name  
  [ for [ each ] { row | statement } ]  
  [ when ( condition ) ]  
  execute procedure function_name (arguments)
```

donde el evento puede ser un de estos:

**insert**

**update** [ **of** column\_name [, ... ] ]

**delete**

**truncate**

# Triggers

**trigger\_name:** El nombre del trigger debe ser distinto al de cualquier otro trigger de la misma tabla (puede haber dos trigger con el mismo nombre en tablas distintas. Sin embargo, se recomienda usar nombres distintos para evitar confusiones.

**before | after:** Determina si la función será invocada **antes** o **después** del evento. Si se elige before, se ejecuta la función llamada por el trigger y luego se realiza el evento (insert, update o delete), por lo contrario, si se elige after, primero se realiza el evento y luego se llama a la función. No puede haber dos triggers en una misma tabla que se disparen al mismo momento y evento.

**table\_name:** nombre de la tabla a la cuál se asocia el trigger

**when:** es opcional, incluye una condicion en la definición de un trigger. Se evalúa la expresión, si es **true**, se ejecuta el cuerpo del trigger. Si es **false** o **not true** (desconocido, como con los valores nulos), el cuerpo del trigger no se ejecute.

# Triggers

**for each row:** El trigger se ejecuta una vez por cada fila afectada por la operación **insert**, **update** o **delete**. Se puede acceder a las variables especiales **new** (nueva fila) para insert o update y/o **old** (fila anterior) para delete o update. El cuerpo del trigger se ejecuta una vez por cada fila de la tabla que haya sido afectada por la sentencia activadora. Por ejemplo, un delete afecta a 10 filas provocará que el trigger **on delete** de la tabla, se invoque 10 veces, una vez por cada fila eliminada.

**For each statement:** El trigger se ejecuta una sola vez por sentencia, sin importar cuántas filas sean afectadas. No puedes usar **old** ni **new** porque no se refiere a una fila específica. Es útil cuando quieres registrar que una operación ocurrió, sin importar en cuántas filas (en particular, una operación que modifica cero filas aún resultará en la ejecución de cualquier **for each statement**).

Escenario	FOR EACH ROW	FOR EACH STATEMENT
Auditar cambio específico	✓ Ideal	✗ No sirve
Registrar una operación en la tabla	✗ Costoso innecesariamente	✓ Perfecto
Variables <b>old</b> - <b>new</b>	✓ Disponibles	✗ No disponibles
Eficiencia con muchos cambios	✗ Puede ser lento	✓ Muy eficiente

# Triggers

**function\_name**: Una función definida por el usuario, que se declara sin argumentos y retorna un tipo trigger. Se ejecuta cuando se dispara el trigger.

**Evento**: **insert**, **update**, **delete** o **truncate** indica el evento que va a activar el trigger. Se pueden indicar varios mediante **or**. Si es **update** se puede incluir una lista de columnas con **of**, el trigger se activará sólo si se modifica alguna de las columnas especificadas. De lo contrario, el trigger se activa con cualquier modificación en la tabla.

**arguments**: Una lista opcional de argumentos separados por comas que se proporcionará a la función cuando se ejecute el trigger. Los argumentos son constantes de cadena o numéricas, pero todos se convertirán en cadenas.



# Triggers - Variables Especiales y tg

Cuando una función PL/pgSQL es llamada por un trigger, se crean automáticamente varias variables especiales en el bloque de nivel superior.

**new**: Variable de tipo registro que contiene la fila “nueva” de la tabla que activo el trigger con las operaciones **insert/update**. Esta variable no está asignada para **delete**.

**old**: Variable de tipo registro, que contiene la fila “antigua” de la tabla que activo el trigger con las operaciones **update/delete**. Esta variable no está asignada para **insert**.

**tg\_name**: Variable que contiene el nombre del trigger que se disparó.

**tg\_op**: Indica que operación activó el trigger, es una cadena de caracteres que contiene **'insert'**, **'update'** o **'delete'**, según sea el caso.

**tg\_when**: indica cuando se ejecutó el trigger, es una cadena de caracteres, contiene **'before'** si el se ejecutó antes de la opreacion o **'after'**, si se disparó despues.

# Triggers - Variables Especiales y tg

**tg\_level**: Cadena de caracteres, contiene 'row' o 'statement', según la definición del trigger.

**tg\_rename**: Contiene el nombre de la tabla que provoca la invocación del trigger. Se puede reemplazar por tg\_table\_name

**tg\_argv[]**: Tipo de dato text array, los argumentos de la sentencia **create trigger**. El índice desde 0. Índices inválidos (menores que 0 ó mayores/iguales que tg\_nargs) resultan en valores nulos.

**tg\_table\_name** : Contiene el nombre de la tabla que provoca la invocación del trigger.

**tg\_table\_schema**: Tipo de dato name, el nombre de la schema de la tabla que ha activado el disparador que está usando la función actualmente.

**tg\_nargs**: Tipo de dato integer, el número de argumentos dados al procedimiento en la sentencia create trigger.

**tg\_reild**: Tipo de dato oid; el ID de la tabla que provoca la invocación del trigger.



# Alter Triggers

**alter trigger:** cambia la definición de un trigger

**alter trigger** name **on** table\_name **rename to** new\_name

Debe ser propietario de la tabla en la que actúa el trigger para poder cambiar sus propiedades.  
Parámetros

**name:** El nombre de un disparador existente que se va a modificar.

**table\_name:** El nombre de la tabla sobre la que actúa este trigger.

**new\_name:** El nuevo nombre del trigger.

También puede habilitar o deshabilitar un trigger

**alter table** name\_table **disable/enable trigger** name\_triggre

**alter table** medicamento **disable trigger** modi\_stock;

# drop Triggers

**drop trigger** : elimina un trigger existente. Para esto, debe ser propietario de la tabla para la que se define el trigger.

**drop trigger** [ **if exists** ] name **on** table\_name [ **cascade** | **restrict** ]

**if exists** No arroje un error si el trigger no existe.

**name**: El nombre del trigger que se eliminará.

**table\_name**: nombre de la tabla para la que se define el trigger

**cascade**: borra todos los objetos que dependen del trigger

**restrict** : No borra el trigger si algún objeto depende de él. Este es el predeterminado.

# Triggers - Ejemplo

**create or replace function**

**nombre\_funcion( )**

Nunca se definen parámetros

El retorno **siempre** debe ser del tipo trigger

**returns**

**trigger**

**as**

**\$nombre\_trigger\$**

Generalmente lleva el nombre del trigger que la invoca

**declare**

**begin**

--cuerpo de la función

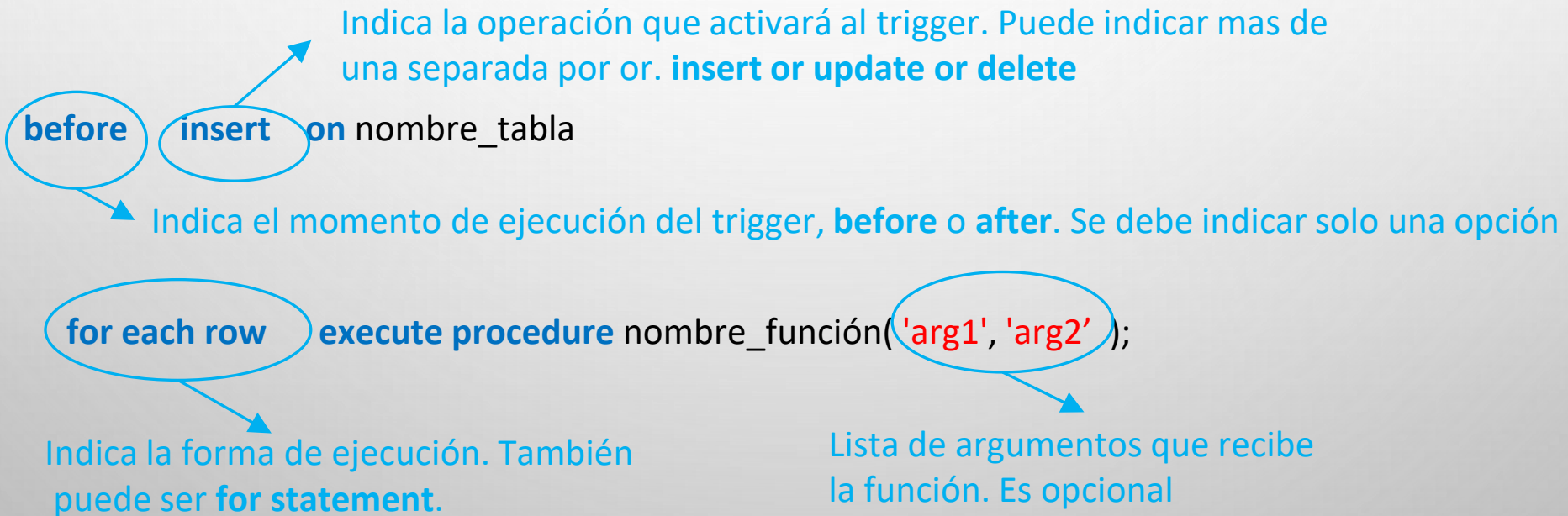
**end;**

**\$nombre\_trigger\$ language plpgsql;**

# Triggers - Ejemplo

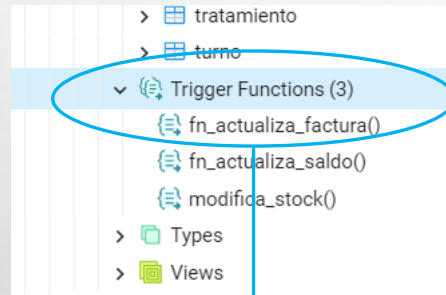
Una vez creada la función, se crea el trigger

**create trigger** nombre\_trigger

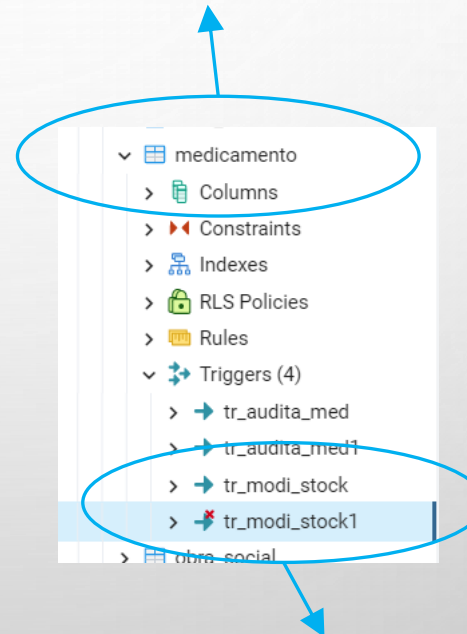


# Triggers - Ejemplo

Los triggers se pueden ver en la tabla asociada a cada uno de ellos, es decir, en cada tabla se puede ver que trigger está listo para su ejecución



Muestra todas las funciones del tipo Trigger, es decir, las funciones que Retornan trigger



Se puede desactivar o activar según la necesidad

# Triggers - Ejemplo

```
create table med_faltante ( id integer,  
                           nombre text,  
                           fecha date,  
                           cantidad integer);
```


```
create or replace function modifica_stock( ) returns trigger as $modi_stock$  
begin  
  if new.stock < 5 then  
    insert into med_faltante values (new.id_medicamento, new.nombre, now(), new.stock);  
  end if;  
  return new;  
end;  
$modi_stock$ language plpgsql ;
```



# Triggers - Ejemplo

```
create trigger modi_stock before update on medicamento  
for each row execute procedure modifica_stock();
```

Se activa solo si se modifica el  
campo stock de la tabla  
medicamento



```
create trigger modi_stock before update of stock on medicamento  
for each row execute procedure modifica_stock();
```

```
create trigger modi_stock after update on medicamento for each row when  
(old.stock is distinct from new.stock) execute procedure modifica_stock();
```

Se activa solo si el valor del campo stock “viejo” es distinto al “nuevo”

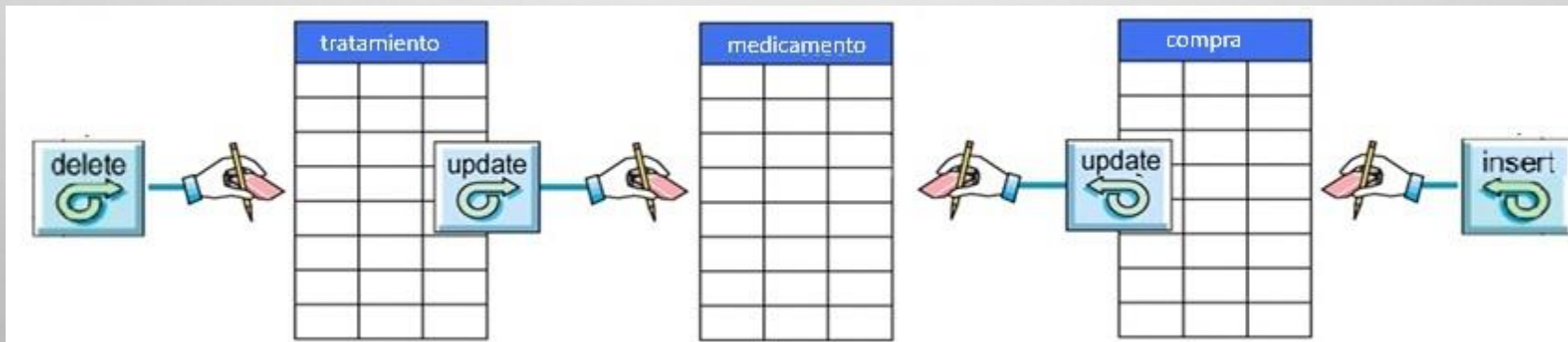


Se crea tres veces un trigger con el mismo nombre en la misma tabla, esto da error, para probar los ejemplos se debe borrar el trigger creado para crear otro con el mismo nombre. Debe tener en cuenta que si cambia el nombre de los trigger se ejecutarán los 3 si se cumple la condición, para no cometer errores debería deshabilitar el trigger creado

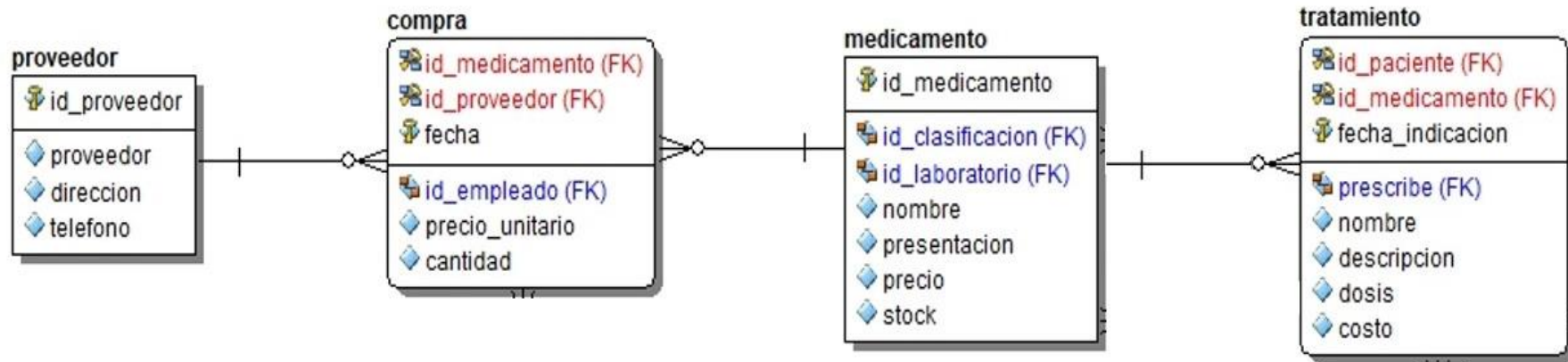


# Triggers - Ejemplo

```
create or replace function repone_stock( ) returns trigger as $calcula_stock$
begin
  if (tg_relname = 'tratamiento' ) then
    update medicamento set stock = stock + old.dosis where id_medicamento = old.id_medicamento;
    return old;
  else
    update medicamento set stock = stock+new.cantidad where id_medicamento = new.id_medicamento;
    return new;
  end if;
end;
$calcula_stock$ language plpgsql;
```



# Triggers - Ejemplo



**create trigger** calcula\_stock  
**after delete on** tratamiento **for each row**  
**execute procedure** repone\_stock();

**create trigger** calcula\_stock  
**after insert on** compra **for each row**  
**execute procedure** repone\_stock();

# Triggers - Ejemplo

```
create or replace function borra_tratamiento()  
returns trigger as $borra_tratamiento$  
begin  
    delete from tratamiento where id_medicamento = old.id_medicamento;  
    delete from compra where id_medicamento = old.id_medicamento;  
    return old;  
end;  
$borra_tratamiento$ language plpgsql;
```

```
create trigger borra_tratamiento after delete on medicamento  
for each row execute procedure borra_tratamiento();
```

```
delete from medicamento where id_medicamento =4
```

No siempre se puede disparar un trigger después (after) debido a que puede generar errores.

# Triggers - Ejemplo

Data Output Messages Notifications

ERROR: update o delete en «medicamento» viola la llave foránea «Refmedicamento97» en la tabla «tratamiento»  
La llave (id\_medicamento)=(4) todavía es referida desde la tabla «tratamiento».

SQL state: 23503

Detail: La llave (id\_medicamento)=(4) todavía es referida desde la tabla «tratamiento».

Se intentará borrar un registro de la tabla medicamento, donde el id\_medicamento es referenciado por registros de las tablas compra y tratamiento, es decir, compra.id\_medicamento “apunta” a un medicamento.id\_medicamento no existente, lo mismo pasa con tratamiento. Primero se debe borrar los registros de las tablas tratamiento y compras, luego poder borrar el registro de medicamento. La forma correcta es hacer que el trigger se dispare antes de borrar el registro en la tabla ventas

Se cambia el  
after por  
**before**

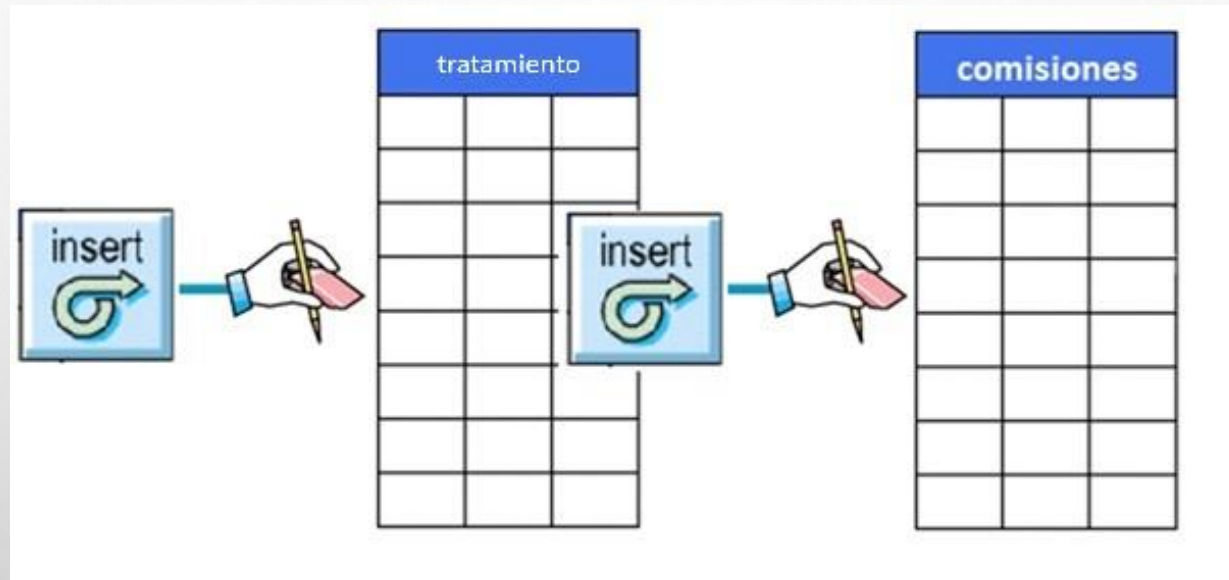
**create trigger** borra\_tratamiento  
**before delete on** medicamento  
**for each row execute procedure** borra\_tratamiento();

# Triggers - Ejemplo

```
create or replace function borra_persona()  
returns trigger as $borra_persona$  
begin  
    delete from consulta where id_paciente = old.id_persona;  
    delete from consulta where id_empleado = old.id_persona;  
    ...  
    delete from empleado where id_empleado = old.id_persona;  
    delete from paciente where id_paciente = old.id_persona;  
    return old;  
end;  
$borra_persona$ language plpgsql;  
  
create trigger borra_persona  
before delete on persona for each row  
execute procedure borra_persona();
```



# Triggers - Ejemplo



# Triggers - Ejemplo

```
create or replace function comision_tratamiento() returns trigger as $comision$
begin
if (new.prescribe in (select id_empleado from comisiones)) then
    update comisiones set fecha=new.fecha_indicacion, monto = monto + new.costo,
    comision = (monto + new.costo)* tg_argv[0]::float --(parámetro)
    where id_empleado=new.prescribe;
    return new;
else
    insert into comisiones values(new.fecha_indicacion, new.prescribe,
    (select nombre from empleado e inner join persona p on p.id_persona = e.id_empleado where id_persona=new.prescribe),
    (select apellido from empleado e inner join persona p on p.id_persona = e.id_empleado where id_persona=new.prescribe),
    new.costo, new.costo * tg_argv[0]::float);
    return new;
end if;
end;

$comision$ language plpgsql;
```



# Triggers - Ejemplo

```
create table comisiones (  
    fecha date,  
    id_empleado integer,  
    nombre varchar(100),  
    apellido varchar(100),  
    monto numeric(8,2),  
    comision numeric(8,2));
```

```
create trigger comision  
after insert on tratamiento for each row  
execute procedure comision_tratamiento(0.1);
```

```
insert into tratamiento values(1000,10,'2025-06-02', 398, 'AERO OM GOTAS', 'FRASCO X 60 CC',4, 5500 )
```

# Triggers - Ejemplo

```
create or replace function cambios_tablas() returns trigger as $cambios_tablas$
Begin
  create table if not exists modifica(operacion varchar(10),
                                     username varchar(20), tabla varchar(50),
                                     triggers varchar(30), cuando varchar(10));

  if (tg_op = 'delete' ) then
    insert into modifica values('D', user, tg_rename, tg_name, tg_when);
    return old;
  elsif (tg_op = 'update') then
    insert into modifica values('U', user, tg_rename, tg_name, tg_when);
    return new;
  elsif (tg_op = 'insert' ) then
    insert into modifica values('I', user, tg_rename, tg_name, tg_when);
    return new;
  end if;
end;
$cambios_tablas$ language plpgsql;
```

# Triggers - Ejemplo

```
create trigger cambios_laboratorio  
before insert or update or delete on laboratorios  
for each row execute procedure cambios_tablas();
```

```
create trigger cambios_proveedores  
after insert or update or delete on proveedores  
for each row execute procedure cambios_tablas();
```

```
create trigger cambios_productos  
after insert or update or delete on productos  
for each row execute procedure cambios_tablas();
```

# Triggers – Ejemplo

```
create function fn_actualiza_saldo() returns trigger as $$  
begin  
    update factura set saldo=saldo - new.monto where id_factura = new.id_factura;  
    if (select saldo from factura where id_factura = new.id_factura) = 0 then  
        update factura set pagada = 'S' where id_factura = new.id_factura;  
    end if;  
    return new;  
end;  
$$ language plpgsql;
```

```
create trigger tr_actualizar_saldo  
after insert or update on pago for each row  
execute procedure fn_actualiza_saldo();
```

```
update pago set monto = 4560 where id_factura = 2  
insert into pago values(2, '2019-08-19', 30000)
```

# Triggers – Ejemplo

```
create function fn_actualiza_factura() returns trigger as $$
declare _monto numeric(11,2); _fecha date; reg record;
begin
if (tg_relname = 'estudio_realizado' ) then
    _monto = new.precio; _fecha = new.fecha;
elseif (tg_relname = 'internacion') then
    _monto = new.costo; _fecha = new.fecha_alta;
else
    _monto = new.costo; _fecha = new.fecha_indicacion;
end if;
select * into reg from factura where id_paciente = new.id_paciente and extract(month from fecha) = extract(month from _fecha)
and extract(year from fecha) = extract(year from _fecha);

if not found then
insert into factura values((select max(id_factura) from factura)+1, new.id_paciente, _fecha, current_time, _monto, 'N', _monto);
else
update factura set monto= monto + _monto, saldo=saldo + _monto where id_paciente = new.id_paciente
and extract(month from fecha) = extract(month from _fecha) and extract(year from fecha) = extract(year from _fecha);
end if;
return new;
end;
$$ language plpgsql;
```

# Triggers - Ejemplo

```
create trigger tr_actualizar_estudiorealizados  
after insert on estudio_realizado for each row  
execute procedure fn_actualiza_factura();
```

```
create trigger tr_actualizar_internacion  
after update on internacion for each row  
execute procedure fn_actualiza_factura();
```

```
create trigger tr_actualizar_tratamiento  
after insert on tratamiento for each row  
execute procedure fn_actualiza_factura();
```



# Triggers - Ejemplo

Crear triggers que se llamen mutuamente (o que entren en un bucle infinito) es muy peligroso, porque puede llevar a:

- ✓ Ciclos infinitos.
- ✓ Saturación del sistema.
- ✓ Fallos por stack overflow o máximo número de llamadas recursivas.

```
create table tabla_a (id serial primary key,  
                      valor integer);
```

```
create table tabla_b (id serial primary key,  
                      valor integer);
```



# Triggers - Ejemplo

-- Trigger en tabla A: al insertar, inserta en B

```
create or replace function trigger_a() returns trigger as $$  
begin  
    insert into tabla_b (valor) values (new.valor);  
    return new;  
end;  
$$ language plpgsql;
```

-- Trigger en tabla B: al insertar, inserta en A

```
create or replace function trigger_b() returns trigger as $$  
begin  
    insert into tabla_a (valor) values (new.valor);  
    return new;  
end;  
$$ language plpgsql;
```

# Triggers - Ejemplo

```
create trigger tr_a          -- Trigger que llama a la función trigger_a
after insert on tabla_a
for each row execute function trigger_a();
```

```
create trigger tr_b          -- Trigger que llama a la función trigger_b
after insert on tabla_b
for each row execute function trigger_b();
```

**insert into** tabla\_a (valor) **values** (1); Se ejecuta trigger\_a, que inserta en tabla\_b y se dispara trigger\_b, que inserta en tabla\_a y vuelve a disparar trigger\_a y así sucesivamente resultando en un loop infinito de inserciones entre tabla\_a y tabla\_b.

Data Output Messages Notifications

ERROR: límite de profundidad de stack alcanzado

HINT: Incremente el parámetro de configuración «max\_stack\_depth» (actualmente 2048kB), después de asegurarse que el límite de profundidad de stack de la plataforma es adecuado.

CONTEXT: sentencia SQL: «insert into tabla\_a (valor) values (new.valor)»

función PL/pgSQL trigger\_b() en la línea 3 en sentencia SQL

sentencia SQL: «insert into tabla\_b (valor) values (new.valor)»



# Gracias