



Tecnicatura Universitaria en Desarrollo de Aplicaciones Informáticas (TUDAI)

Base de Datos

Tema 6: Restricciones de Integridad

2
0
2
5

Implementación Procedural en PostgreSQL

Alternativa para Especificar Restricciones

- ✓ imposibilidad de utilizar assertions en DBMS
- ✓ carencia de implementación de ciertas acciones referenciales
- ✓ no disponibilidad de acciones específicas diferentes al rechazo y la reparación estándar



necesidad de una herramienta útil para escribir aserciones, restricciones complejas, acciones específicas de reparación, etc.

→ Triggers (disparadores)



Trigger: Utilidad

Los *triggers* se pueden usar para:

- Mantener datos derivados - Generación automática de datos
- Forzado de reglas de integridad o del negocio complejas (*Ej. cuando no es posible incluirlas declarativamente*) o con acciones específicas de reparación (*diferentes al rechazo y la reparación estándar*)
- Propagación de actualizaciones
- Generación de logs para soporte de auditoría de las acciones de la base de datos y chequeos de seguridad
- Mantener vistas actualizadas (*cuando el DBMS no provee capacidades para hacerlo*)

Procedimientos Almacenados

STORED PROCEDURE es un programa (o procedimiento) almacenado físicamente en la BD. Su implementación varía de un DBMS a otro.

Procedimientos Almacenados

Desde los Stored Procedure se tiene acceso directo a los datos

Ventajas:

- Aíslar procesos comunes de las aplicaciones, delegándolas al DBMS.
- Eficiencia, porque sólo se envían los resultados al usuario (no hay una sobrecarga de grandes volúmenes viajando por la red.)

Desventajas:

- Cada proveedor de BD tiene su **propio** lenguaje procedural.
- El SQL3 incorporó estas características, pero poco de lo definido anteriormente se ajustaba a un estándar.

PL/pgSQL

Procedural Language/PostgreSQL Structured Query Language, es un lenguaje imperativo que permite ejecutar sentencias SQL a través de sentencias imperativas y funciones.

Brinda la posibilidad de realizar los controles que las sentencias declarativas de SQL no pueden otorgar.

PL/pgSQL

Como cualquier lenguaje de programación posee estructuras de control (repetitivas y condicionales)

Se pueden definir variables, tipos y estructuras de datos.

Se pueden crear funciones que sean invocadas en sentencias SQL normales o ejecutadas en eventos de tipo disparador (*trigger*).

PL/pgSQL

- Las funciones escritas en PL/pgSQL aceptan **argumentos** y pueden **devolver valores** de tipo básico o complejo (ejemplo, registros, vectores, conjuntos e incluso tablas)
- PL/pgSQL se puede utilizar para realizar funciones que se invoquen desde un trigger.

Disparadores (TRIGGERS)

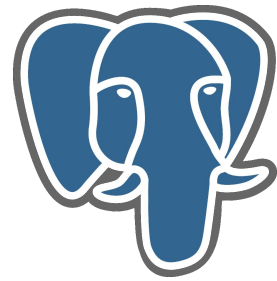


Pueden utilizarse para especificar acciones automáticas que debe realizar el DBMS cuando ocurran ciertos eventos y condiciones.

Funcionalidad de las Bases de Datos Activas, que siguen un modelo de **reglas ECA (Evento-Condición-Acción)**

- 1. Eventos** que activan la regla (operaciones de actualización de la BD)
- 2. Condición** que determina si la acción de la regla debe ejecutarse (Opcional). Si se especifica, entonces primero se evalúa, y si evalúa a VERDADERO se ejecuta la acción de la regla. Si evalúa FALSO o DESCONOCIDO no se ejecuta
- 3. Acción**, suele ser una secuencia de sentencias SQL, pero también podría ser una transacción de base de datos o un programa externo que se ejecutará automáticamente.

Triggers



Según PostgreSQL

CREATE TRIGGER *nombre*

{ BEFORE | AFTER | INSTEAD OF }

{ *evento* [OR ...] }

INSERT, UPDATE, DELETE, or TRUNCATE (para UPDATE OF columna1 [, columna2 ...]

ON *tabla*

[FOR [EACH] { ROW | STATEMENT }]

[WHEN (*condición*)]

EXECUTE PROCEDURE

función_específica ()

ACTIVACIÓN


Evento que lo dispara

GRANULARIDAD

Condición

Acción

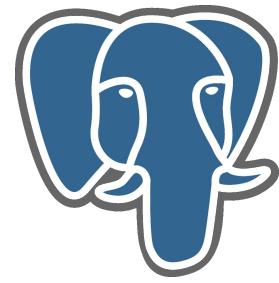
Stored Procedures y Triggers

Para  todo es una función!!!!

PostgreSQL

...pero hay funciones que devuelven **void** entonces podemos tener los **procedimientos**. La sintaxis de las funciones es:

```
create [or replace] function nombre_funcion (lista_parametros)
    returns tipo_retorno as
$$
declare
-- declaracion de variables
begin
-- logica
end;
$$
language plpgsql
```

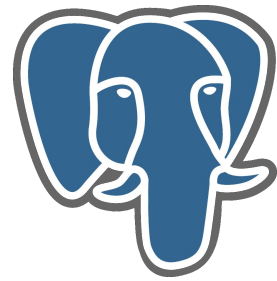


Funciones de tipo Trigger

En PostgreSQL

```
CREATE FUNCTION nombre ( )  
RETURNS trigger AS $$  
  
[ DECLARE ]  
[ declaraciones de variables ]  
  
BEGIN  
  Código PlpgSQL  
END;  
$$ LANGUAGE 'plpgsql';
```

Funciones de tipo Trigger

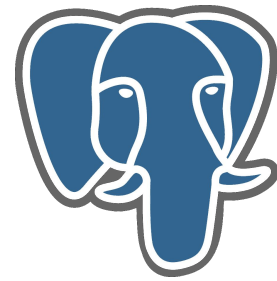


En PostgreSQL

Dentro del cuerpo de la función tengo a disposición diferentes variables:

NEW Tipo de dato RECORD; variable que almacena la nueva fila para las operaciones INSERT/UPDATE en Triggers a nivel ROW, *en los Triggers a nivel STATEMENT es NULL*

OLD Tipo de datos RECORD; variable que almacena la antigua fila para operaciones UPDATE/DELETE en Triggers de nivel ROW, *en Triggers de nivel STATEMENT es NULL*



Funciones

En PostgreSQL

Algunas otras (hay muchas más) variables especiales son:

TG_NAME Tipo de dato text; variable que contiene el nombre del trigger actualmente disparado.

TG_WHEN Tipo de dato text; una cadena conteniendo el string BEFORE o AFTER dependiendo de la definición del trigger.

TG_LEVEL Tipo de dato text; una cadena conteniendo el string ROW o STATEMENT dependiendo de la definición del trigger.

TG_OP Tipo de dato text; una cadena conteniendo el string INSERT, UPDATE o DELETE indicando por cuál operación se disparó el trigger.

TG_TABLE_NAME Tipo de dato text; variable que contiene el nombre de la tabla que disparó el trigger

Trigger: Eventos y Activación

EVENTO: puede ser una operación de actualización sobre la tabla o vista a la que está asociado el trigger:

- Inserción (INSERT)
- Actualización (UPDATE): se puede especificar columna/s
- Eliminación (DELETE)

TIEMPO DE ACTIVACIÓN: en relación a la sentencia que lo activa, el trigger puede ejecutarse:

- antes de la sentencia disparadora (BEFORE)
- después de la sentencia disparadora (AFTER)
- en lugar de la sentencia disparadora (INSTEAD OF)

Trigger: Granularidad

Los triggers pueden ser:

- **FOR EACH ROW:** se ejecuta una vez por cada fila afectada
- **FOR EACH STATEMENT:** se ejecuta una vez para la sentencia SQL disparadora, independientemente de la cantidad de filas que afecte

(Por defecto → FOR EACH STATEMENT)

Trigger: Referencias

INSERT manipula una *nueva* fila (si el trigger es FOR EACH ROW) o un *nuevo conjunto de filas* (si es FOR EACH STATEMENT)

DELETE manipula una fila *vieja* (para triggers a nivel fila) o un conjunto de filas o tabla *vieja* (para triggers de sentencia)

UPDATE manipula estados *viejos* y *nuevos*, tanto de filas como de conjuntos de filas, según corresponda

Corresponde referirse a estos elementos como NEW. y OLD. dentro del cuerpo de la función triggers

Trigger: Acción

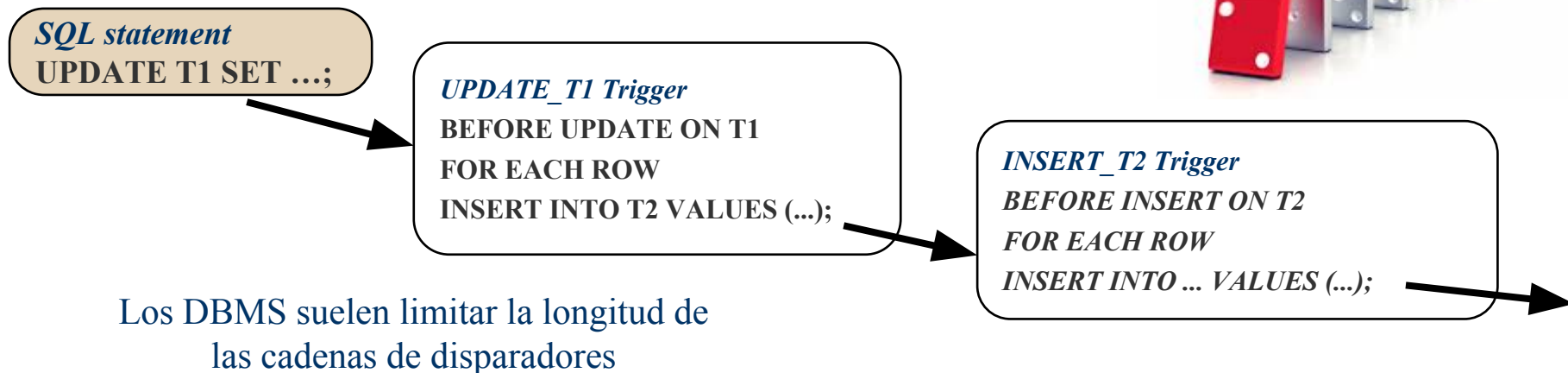
- La acción consiste en una **sentencia SQL aislada** o un **conjunto de sentencias**, delimitadas en un bloque BEGIN . . . END
- Puede referirse a valores anteriores y nuevos que se modifican, nuevos que se insertan, o anteriores que se eliminaron, según el evento que desencadenó la acción
- Pueden incluir sentencias de control (IF ... ELSE, FOR, WHILE, ...)
- No pueden incluir sentencias DDL (*CREATE*, *ALTER*, *DROP*)

Trigger: Acción

- Un trigger BEFORE no debería contener sentencias SQL que alteren datos (*INSERT*, *UPDATE*, *DELETE*): esto puede disparar otros triggers BEFORE (sus acciones van quedando pendientes)
- La acción del trigger es un **procedimiento atómico** → Si cualquier sentencia del cuerpo del trigger falla, la acción completa del trigger se deshace, incluyendo las correspondientes a la sentencia que lo disparó

Trigger: Comportamiento

- La acción del trigger es un procedimiento atómico
- Ante un cierto evento sobre una tabla → pueden activarse varios triggers!
- Se puede producir una activación de triggers en cascada → Si la activación de un trigger *T1* dispara otro trigger *T2*: se suspende la ejecución de *T1*, se ejecuta el trigger anidado *T2* y luego se retoma la ejecución de *T1*
→ esto podría dar lugar a una cadena “infinita” de activaciones!



Triggers: Utilidad

Forzar las reglas de integridad que no pudieron ser especificadas de forma declarativa dentro del DBMS

Cómo comienzo a transformar las RI declarativas en procedurales?

Paso 1 - Determinando los eventos críticos, usar la matriz de ayuda

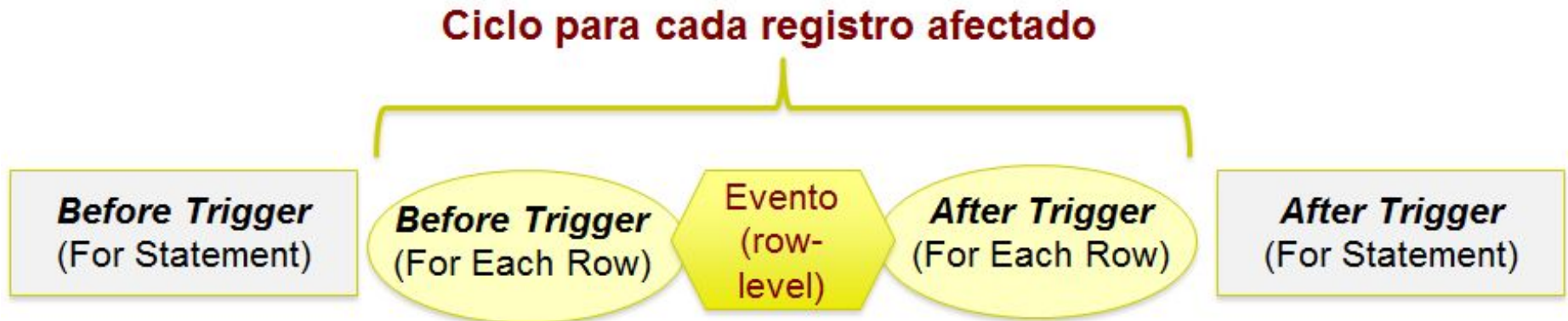
Tabla / Evento	INSERT	UPDATE	DELETE
nombre_tabla		especificar atributos	

Qué tipo de granularidad debe tener un trigger que controla una RI?

Qué tipo de tiempo de activación debe tener un trigger que controla una RI?

Miremos un poco el el orden de disparo de los trigger.....

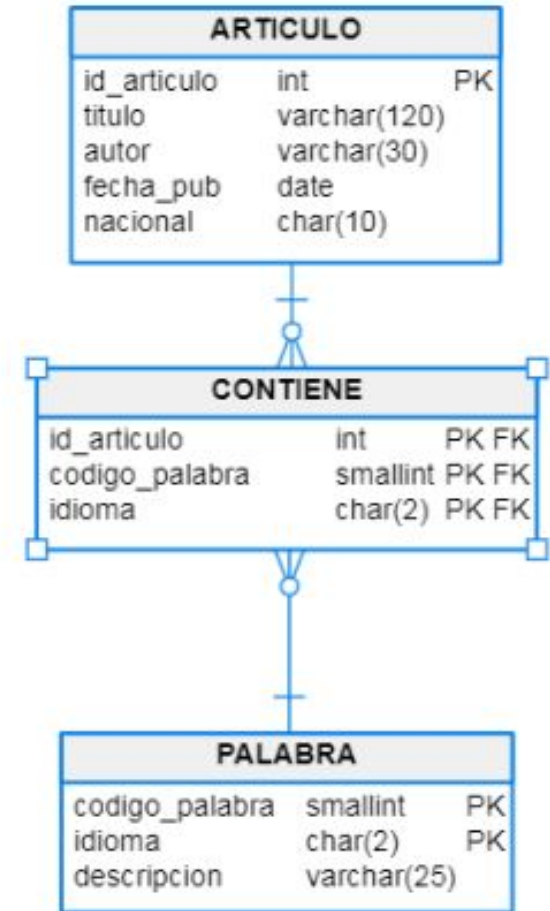
Ejecución de los Trigger



RI Declarativa



Ejercicio 3 - TP 3 - Los artículos pueden tener como máximo 15 palabras claves





Implementación de RI con Trigger

Ejercicio 3 - TP 3 - Los artículos pueden tener como máximo 15 palabras claves

```
ALTER TABLE CONTIENE  
ADD CONSTRAINT CK_MAXIMO_PL_CLAVES  
CHECK NOT EXISTS (  
    SELECT 1  
    FROM CONTIENE  
    GROUP BY id_articulo  
    HAVING COUNT(*) > 15);
```

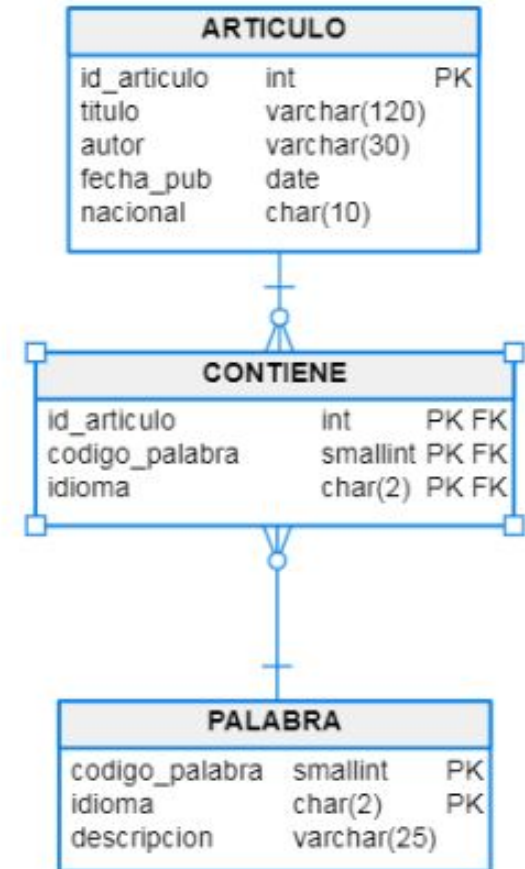


Tabla / Evento	INSERT	UPDATE	DELETE



Implementación de RI con Trigger

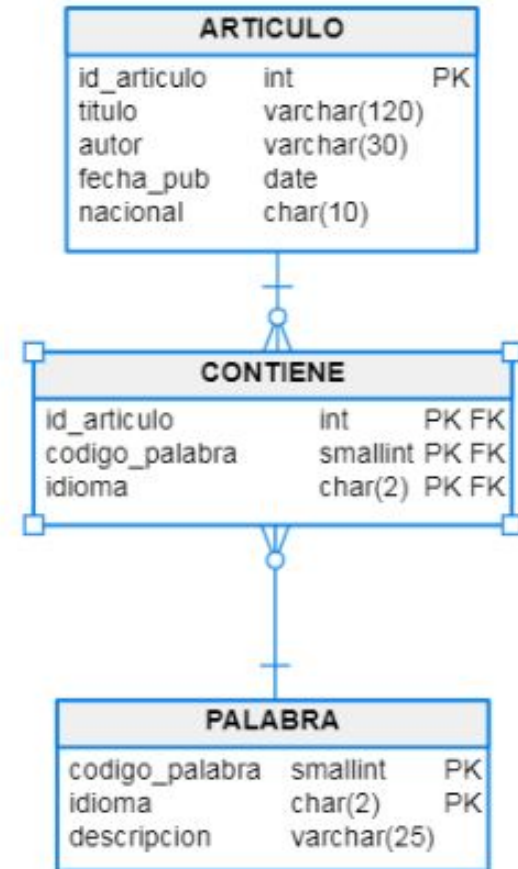
```
CREATE OR REPLACE FUNCTION FN_MAXIMO_PL_CLAVES() RETURNS Trigger AS $$
BEGIN
    IF ((SELECT count(*) FROM CONTIENE
        WHERE id_articulo = NEW.id_articulo) > 14 ) THEN
        RAISE EXCEPTION 'Superó la cantidad de palabras claves en el artículo %',
            NEW.id_articulo
    END IF;
    RETURN NEW;
END $$
LANGUAGE 'plpgsql';

CREATE TRIGGER TR_MAXIMO_PL_CLAVES
BEFORE INSERT OR UPDATE OF id_articulo
ON CONTIENE
FOR EACH ROW EXECUTE PROCEDURE FN_MAXIMO_PL_CLAVES();
```

RI Declarativa



Ejercicio 4 - TP 3 - Sólo se pueden publicar artículos argentinos que contengan hasta 10 palabras claves.



Implementación de RI con Trigger



Ejercicio 4 - TP 3 - Sólo se pueden publicar artículos argentinos que contengan hasta 10 palabras claves.

CREATE ASSERTION

ASS_ART_ARG_MAXIMO_PL_CLAVES

CHECK NOT EXISTS (

SELECT 1

FROM ARTICULO A JOIN CONTIENE C

ON (a.id_articulo = c.id_articulo)

WHERE nacionalidad = 'Argentino'

GROUP BY a.id_articulo

HAVING COUNT(*) > 10);

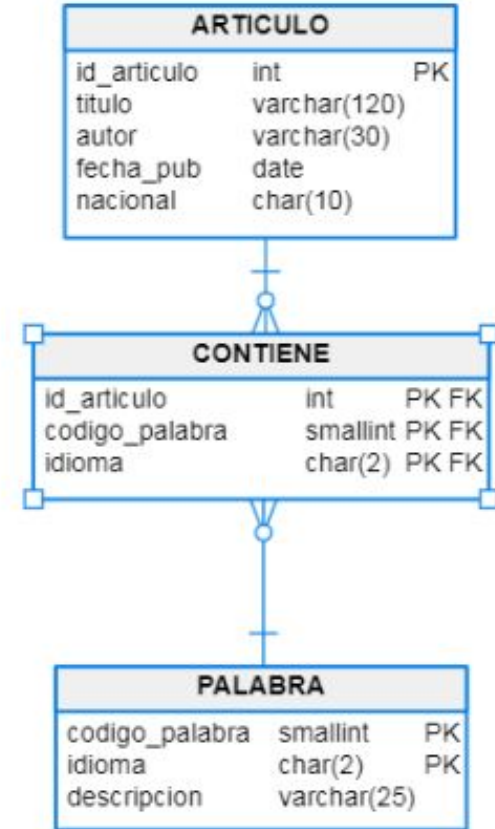


Tabla / Evento	INSERT	UPDATE	DELETE

Implementación de RI con Trigger



Ejercicio 4 - TP 3 - Sólo se pueden publicar artículos argentinos que contengan hasta 10 palabras claves.

CREATE ASSERTION

ASS_ART_ARG_MAXIMO_PL_CLAVES

CHECK NOT EXISTS (

SELECT 1

FROM ARTICULO A JOIN CONTIENE C

ON (a.id_articulo = c.id_articulo)

WHERE nacionalidad = 'Argentino'

GROUP BY a.id_articulo

HAVING COUNT(*) > 10);

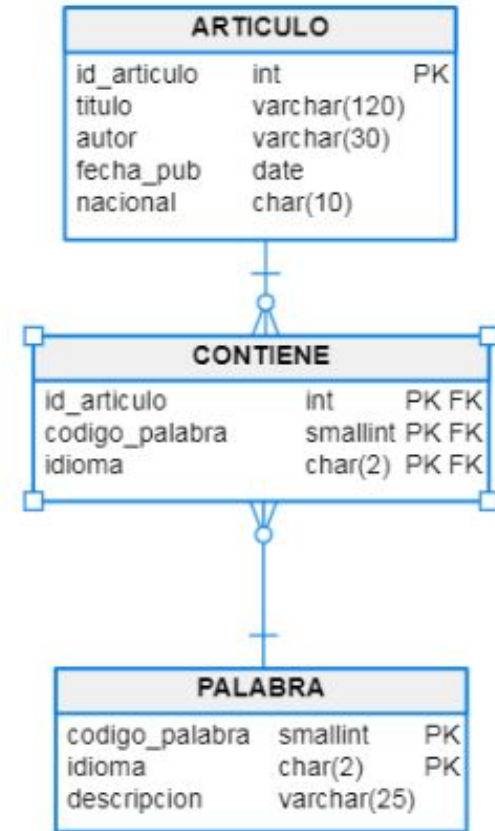


Tabla / Evento	INSERT	UPDATE	DELETE
ARTICULO	NO	SI nacionalidad	NO
CONTIENE	SI	SI id_articulo	NO



Implementación de RI con Trigger

```
CREATE OR REPLACE FUNCTION FN_ART_ARG_MAXIMO_PL_CLAVES_CNT() RETURNS
Trigger AS $$
DECLARE
    cant          integer;
    v_nacional    articulo.nacional%type;
BEGIN
    --cuando el trigger se despierta por un insert or update en contiene
    SELECT nacional INTO v_nacional
    FROM ARTICULO A
    WHERE a.id_articulo = NEW.id_articulo;
    IF (v_nacional = 'Argentina') THEN
        SELECT count(*) INTO cant
        FROM CONTIENE
        WHERE id_articulo = NEW.id_articulo;
        IF (cant > 9) THEN
            RAISE EXCEPTION 'Superó la cantidad % palabras claves', cant;
        END IF;
    END IF;
    RETURN NEW;
END $$
LANGUAGE 'plpgsql';
```



Implementación de RI con Trigger

```
CREATE OR REPLACE FUNCTION FN_ART_ARG_MAXIMO_PL_CLAVES_ART()  
RETURNS Trigger AS $$  
BEGIN  
    --cuando el trigger se despierta por un update en artículo  
    IF ( (SELECT count(*)  
        FROM CONTIENE  
        WHERE id_articulo = NEW.id_articulo) > 9) THEN  
        RAISE EXCEPTION 'Superó la cantidad % palabras claves', cant;  
    END IF;  
    RETURN NEW;  
END $$  
LANGUAGE 'plpgsql';
```



Implementación de RI con Trigger

```
CREATE TRIGGER TR_ART_ARG_MAXIMO_PL_CLAVES_ART  
BEFORE UPDATE OF nacionalidad  
ON ARTICULO  
FOR EACH ROW  
WHEN (NEW.nacional = 'Argentina')  
EXECUTE PROCEDURE FN_ART_ARG_MAXIMO_PL_CLAVES_ART();
```

```
CREATE TRIGGER TR_ART_ARG_MAXIMO_PL_CLAVES_CONT  
BEFORE INSERT OR UPDATE OF id_articulo  
ON CONTIENE  
FOR EACH ROW  
EXECUTE PROCEDURE FN_ART_ARG_MAXIMO_PL_CLAVES_CNT();
```

TRIGGERS – EJEMPLOS

✓ Forzado de reglas de integridad

create trigger *<nombre>*

before *<operación crítica sobre la BD>*

when *<condición por la que una RI es incumplida>*

< acción(es) del trigger > → rechazó (acción pasiva) /reparación (acción activa)

Ejemplo

Verificar que el sueldo de un empleado no se reduzca

```
CREATE TRIGGER sueldo_no_se_reduce
  BEFORE UPDATE OF sueldo ON Empleado
  FOR EACH ROW
  WHEN (old.sueldo > new.sueldo)
  EXECUTE PROCEDURE funcion_error();
```


TRIGGERS – EJEMPLOS

✓ Actualización (automática) de datos derivados

Ejemplo

Mantener *automáticamente* la cantidad total de empleados del Area
(ante altas, bajas o modificaciones en Empleado)

EMPLEADO(idE, nombre, ..., AreaT)

AREA(idArea, ... CantEmp)

```
CREATE TRIGGER Incrementar_EmpArea
AFTER INSERT OR UPDATE OF AreaT OR DELETE
ON Empleado
FOR EACH ROW
EXECUTE PROCEDURE cant_total_empleados();
```

TRIGGERS – EJEMPLOS

✓ Actualización (automática) de datos derivados

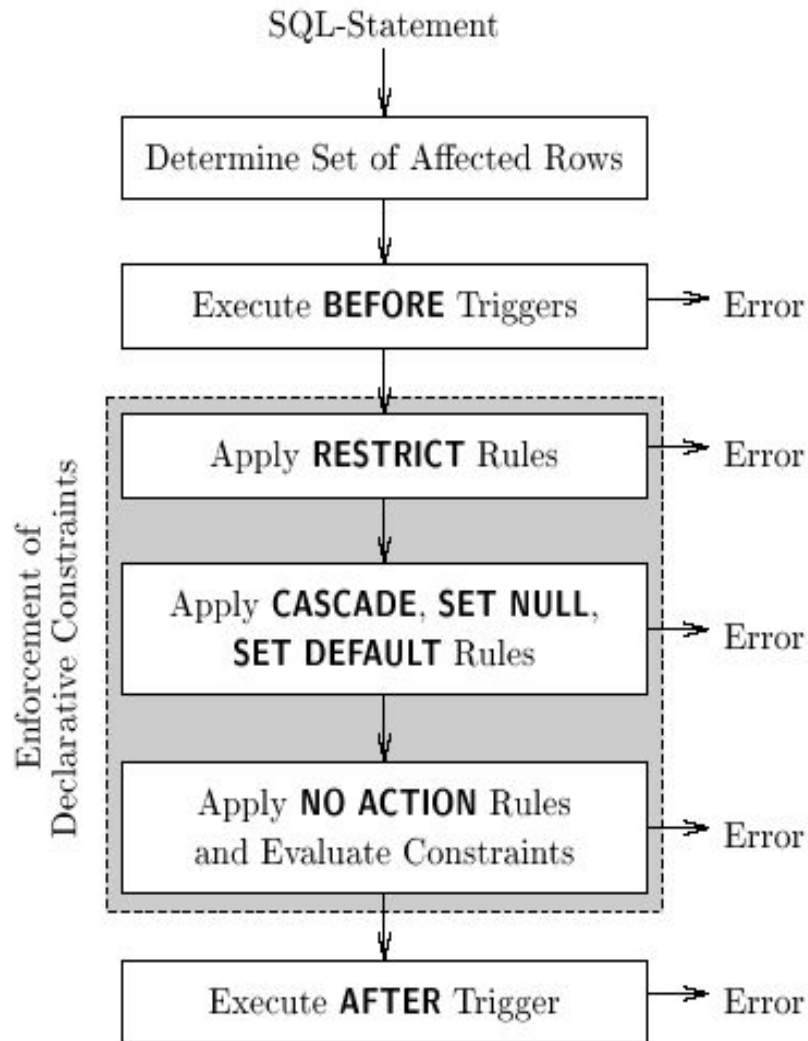
Ejemplo

```
CREATE FUNCTION cant_total_empleados ( )
RETURNS trigger AS $body$
BEGIN
    IF TG_OP = 'INSERT' THEN
        UPDATE area set CantEmp = CantEmp + 1 where IdArea = new.AreaT;
        RETURN NEW;
    END IF;
    IF TG_OP = 'UPDATE' THEN
        UPDATE area set CantEmp = CantEmp - 1 where IdArea = old.AreaT;
        UPDATE area set CantEmp = CantEmp + 1 where IdArea = new.AreaT;
        RETURN NEW;
    END IF;
    IF TG_OP = 'DELETE' THEN
        UPDATE area set CantEmp = CantEmp - 1 where IdArea = old.AreaT;
        RETURN OLD;
    END IF;
END; $body$
LANGUAGE 'plpgsql'
```

TRIGGERS vs. RI DECLARATIVAS

- Triggers → permiten *definir y forzar* reglas de integridad, pero *NO son una restricción de integridad*
- Un trigger definido para forzar una RI no verifica su cumplimiento para los datos ya almacenados en la BD (una RI declarativa verifica la *carga existente* en la BD)
- Los triggers deberían usarse sólo cuando una RI no puede ser expresada mediante una cláusula declarativa

MODELO DE EJECUCIÓN SQL-99 (+ TRIGGERS)



1. Ejecuta todos los triggers **BEFORE-statement**
2. Realiza un ciclo por todas las filas afectadas por la sentencia SQL
 - a. Ejecuta todos los triggers **BEFORE-row**
 - b. Bloquea y actualiza cada fila y ejecuta los chequeos de integridad declarat. *(El bloqueo no se levanta hasta el final de la transacción)*
 - c. Ejecuta todos los triggers **AFTER-row**
3. Completa las acciones correspondientes a la verificación diferida de integridad expresada declarativamente
4. Ejecuta todos los triggers **AFTER-statement**

BIBLIOGRAFÍA

Capítulo 42 del Manual de PostgreSQL Versión 13:

www.postgresql.org que se encuentra [aquí](#)

PL/PgSQL y otros lenguajes procedurales en Postgresql:

https://www.postgresql.org/message-id/attachment/92321/pl_pgsql_y_otros_lenguajes_procedurales_en_postgresql.pdf



Implementación de RI con Trigger

```
CREATE OR REPLACE FUNCTION FN_ART_ARG_MAXIMO_PL_CLAVES_ART()  
RETURNS trigger AS $$  
DECLARE  
    cant      integer;  
    v_nacional articulo.nacional%type;  
BEGIN  
    --cuando el trigger se despierta por un update en articulo  
    SELECT count(*) INTO cant  
    FROM CONTIENE  
    WHERE articulo_id_articulo = NEW.id_articulo;  
    IF (cant > 2 ) THEN  
        RAISE EXCEPTION 'Superó la cantidad % palabras claves', cant;  
    END IF;  
    RETURN NEW;  
END $$  
LANGUAGE 'plpgsql';
```