Lenguaje PL/pgSQL

Estructura

- PL/pgSQL es un lenguaje estructurado por bloques.
- Cada unidad PL/pgSQL se compone de uno o más bloques. Cada bloque puede estar completamente separado del resto, o anidado dentro de otro bloque.

Tipos de programas que usan los bloques de PL/SQL

Tipo de programa	Descripción
Bloque anónimo	Bloque PL/pgSQL sin nombre que está inmerso en una aplicación o es llamado interactivamente.
Función almacenada	Bloque PL/pgSQL con nombre, que puede ser ejecutado repetidamente y acepta parámetros.
Trigger de Base de Datos	Bloque PL/pgSQL asociado con una tabla de la base de datos y que se dispara automáticamente al ejecutarse instrucciones de actualización.

Secciones de un Bloque

Un bloque está compuesto de 3 secciones:

- **DECLARE** – Opcional
  - Variables, cursores, excepciones definidos por el usuario
- **BEGIN** – Requerido
  - Instrucciones SQL
  - Instrucciones PL/pgSQL
- **EXCEPTION** – Opcional
  - Acciones a realizar cuando ocurren errores
- **END** – Requerido

Estructura de Bloques

```
DECLARE
    v_variable     VARCHAR(5);
BEGIN
    SELECT  nombre_columna
    INTO v_variable
    FROM nombre_tabla
EXCEPTION
    WHEN nombre_excepcion THEN
    …..
END;
```

Tipos de bloques y sintaxis

Anónimo	**Procedimiento**	**Función**
[**DECLARE**] ... **BEGIN** *Instrucciones* [**EXCEPTION**] ... **END**;	**FUNCTION** *nombre()* **RETURNS** *VOID AS $$* **BEGIN** *Instrucciones* **RETURN***;* [**EXCEPTION**] ... **END**; $$ LANGUAGE plpgsql;	**FUNCTION** *nombre()* **RETURNS** *tipoDeDatos* AS $$ **BEGIN** *Instrucciones* **RETURN** *valor*; [**EXCEPTION**] ... **END**; $$ LANGUAGE plpgsql;

Los procedimientos no existen como estructura en PL/pgSQL. Se pueden imitar con una función que retorna void.

Ejemplo procedimiento almacenado

```
/* Procedimiento que calcula numero total de creditos
   inscritos de los estudiantes */
   CREATE OR REPLACE FUNCTION calcula_credins()
   RETURNS VOID AS $$
      /* Declaración de variable para numero total de creditos inscritos
         de cada estudiante */
      /* Declaracion de cursor para la tabla de estudiantes*/
   BEGIN
      /* Para cada fila del cursor
            Calcular el numero total de creditos inscritos mediante consulta a
              SE_ASIGNA
            Actualizar la fila de la tabla de ESTUDIANTE */
   EXCEPTION
      /* Manejo de excepción – caso de cursor mal definido */
   END;
   $$ LANGUAGE plpgsql;
```

Lineamientos para la codificación de programas

- Documente el código con comentarios.
- Indente el código para facilitar su legibilidad.
- Siga algún estándar para los nombres.
- Use nombres mnemónicos.
- Inicialice los identificadores usando el operador de asignación (**:=**) o usando la palabra reservada **DEFAULT**.
- Declare a lo sumo una variable por línea.

Variables

Utilidad:

- Almacenamiento temporal de datos.
- Manipulación de valores almacenados.
- Reusabilidad.
- Facilidad de mantenimiento.

Lineamientos generales

- Declare e inicialice variables en la sección de declaración.
- Asigne nuevos valores a variables en la sección de ejecución.
- Pase valores a bloques PL/pgSQL a través de parámetros.
- Observe resultados a través de variables de salida.

Tipos

1) Variables PL/pgSQL
   a) Escalar
      -Un solo valor
      -Corresponden a los tipos de las columnas de las tablas:
         VARCHAR(*max long*), **NUMERIC**[(*precision, escala*)], **DATE**, **CHAR**([*max long*)], **LONG**, **LONG RAW**, **BOOLEAN**, **BINARY_INTEGER**, **PLS_INTEGER**
   b) Compuesto
      -Estructurados en varios atributos.
   c) Referencia
      -Apuntadores que referencian otros elementos del programa.
   d) LOB (*large objects*)
      -Contienen *locators* para especificar la ubicación de grandes objetos
2) Variables no PL/pgSQL
   a) Variables del ambiente

Sintaxis de declaración y ejemplos

identificador tipo_de_datos [**DEFAULT**] [**NOT NULL**]
[:= expresion];

DECLARE

v_activo **BOOLEAN DEFAULT TRUE**;

v_fechaEgreso **DATE**;

v_numEmp **CHAR**(2);

v_ubicación **VARCHAR**(15) := ‘Caracas’;

Lineamientos para la declaración de variables

- Dos variables pueden tener el mismo nombre si se encuentran en bloques diferentes.
- El nombre de la variable no debe coincidir con el nombre de las columnas de tablas utilizadas en el bloque.
- Las variables son inicializadas cada vez que se ejecuta el bloque.
- Por defecto, el valor inicial es **NULL**.

Asignación

Para asignar el valor a una variable se utiliza el operador de asignación (**:=**) con la siguiente sintaxis:

Identificador **:=** *Expresion*

- *Identificador* es el nombre de la variable.
- *Expresion* puede ser una variable, literal, llamada a función, pero no una columna de tabla.

Otra manera de asignar valor a una variable es a través de la cláusula INTO, colocando valores de la Base de Datos dentro de la misma.

Ejemplo:

```
SELECT AVG(indice)
INTO v_promIndice
FROM ESTUDIANTE;
```

El Atributo **%TYPE**

- Se utiliza para declarar una variable de acuerdo a:
  - Una definición de columna de una tabla
  - Una variable creada anteriormente.
- %TYPE se coloca luego del atributo o variable precedido por un punto:
  - La columna y tabla de la base de datos.
  - La variable declarada previamente.

Ejemplo:

Variable del procedimiento calcula_credins

v_credins ESTUDIANTE.credins**.%TYPE**;

Funciones en PL/pgSQL

- Las funciones de una sola fila de SQL están disponibles en PL/pgSQL:
  - Numéricas
  - Caracteres
  - Conversión de datos
  - Fechas
- Las funciones de agregación de SQL no están disponibles en PL/pgSQL

Operadores en PL/pgSQL

- Los operadores de SQL están disponibles en PL/SQL:
  - Lógicos
  - Aritméticos
  - Concatenación
  - Paréntesis para controlar el orden de operaciones

Ejemplo:

```
v_balance := v_balance – v_retiro;
v_mesIngreso := TO_CHAR(v_fingreso,'mm');
```

Instrucciones de SQL en PL/pgSQL

- Extraiga una fila de datos mediante la instrucción **SELECT**. Sólo una fila puede ser retornada.
- En caso de que se quiera recuperar más de una fila deberá utilizarse un cursor.
- Se pueden hacer cambios a múltiples filas de la base de datos utilizando **INSERT**, **UPDATE** y **DELETE**.
- En un bloque PL/SQL no se puede tener instrucciones **CREATE TABLE**, **ALTER TABLE** ni **DROP TABLE**.
- Controle una transacción con las instrucciones **COMMIT**, **ROLLBACK** y **SAVEPOINT.**

Sintaxis de instrucciones SQL

SELECT lista_columnas
INTO [identificador_variable
[, identificador_variable,]
……. | nombre_registro]
FROM tabla
WHERE condicion;

Ejemplo de Instrucción en SQL para asignar una variable

Ejemplo:

“Obtenga el número total de créditos inscritos por el estudiante con carnet 99-12345”.

```
SELECT SUM(creditos) INTO v_credins
FROM SE_ASIGNA S, ASIGNATURA A
WHERE A.carnet = ‘99-123245’ S.codasig = A.codasig;
```

- Se pueden seleccionar varias columnas y asignarlas respectivamente mediante **INTO** a una serie de variables.

Actualización

Se puede ingresar, modificar y eliminar datos de las tablas de la Base de Datos desde un bloque PL/pgSQL

Ejemplo:

“Actualice el número total de créditos inscritos por el estudiante con carnet 99-12345”

UPDATE ESTUDIANTE

SET credins = v_credins

WHERE carnet = ‘99-12345’

Cursores

- Un cursor es un área de trabajo sobre una o más tablas que se crea cuando se ejecuta una instrucción SQL.
- Existen dos tipos de cursores:
  - Implícito: Se crea al ejecutarse un **SELECT** en PL/SQL.
  - Explícito: Se declaran explícitamente por el programador.

Cursores implícitos – Atributos

SQL%ROWCOUNT	Número de filas afectadas por la instrucción SQL más reciente.
SQL%FOUND	Atributo booleano que es TRUE si la última instrucción SQL afectó una o más filas.
SQL%NOTFOUND	Atributo booleano que es TRUE si la última instrucción SQL no afectó filas.

Estos atributos se utilizan para evaluar lo que sucede al ejecutarse cualquier instrucción SQL.

Estructuras de control

Se puede cambiar el flujo lógico de las instrucciones utilizando la instrucción condicional IF y estructuras de control para *loops*.

Las instrucciones condicionales son:

IF – THEN – END IF

IF – THEN – ELSE – END IF

IF – THEN – ELSIF – END IF

Sintaxis de Condicional

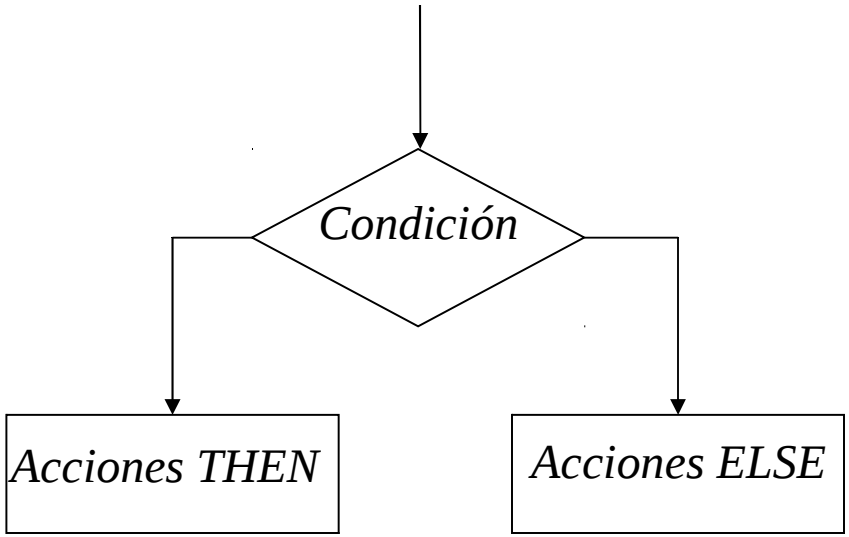
IF *condicion* **THEN**

instrucciones;

[**ELSE**

instrucciones;]

END IF



Ejemplo de condicional

“Dar la condición del estudiante de acuerdo al índice”

```
DECLARE
   v_indice ESTUDIANTE.indice.%TYPE;
   v_condicion VARCHAR(15);
BEGIN
   SELECT indice INTO v_indice
   FROM ESTUDIANTE WHERE carnet = ‘99-12345’
   IF v_indice < 2 THEN
      v_condicion = ‘retirado’;
   ELSE IF v_indice >= 2 AND v_indice <= 2.39 THEN
            v_condicion = ‘periodo prueba’;
         ELSE
            v_condicion = ‘activo’;
         END IF;
   END IF;
END;
```

Instrucciones de repetición

Los *LOOPs* repiten una secuencia de instrucciones varias veces.

Los tipos son:

- *Loop* básico: Acciones repetitivas sin condición.
- *Loop* **FOR**: Acciones repetitivas basadas en un contador.
- *Loop* **WHILE**: Acciones repetitivas basadas en una condición.

LOOP básico

LOOP

instrucción1;

instrucción2;

…..

EXIT [**WHEN** *condicion*];

END LOOP;

Sin la instrucción **EXIT**, el *loop* sería infinito.

Ejemplo de LOOP Básico

"Insertar 5 secciones para la asignatura 'AA0002' "

```
DECLARE
   v_contador NUMERIC(2) := 1;
BEGIN
   LOOP
      INSERT INTO SECCION(codasig,numsec)
    VALUES('AA0002', v_contador);
      v_contador := v_contador + 1;
      EXIT WHEN v_contador > 5;
   END LOOP;
END;
```

Loop **FOR**

FOR *contador* **IN** [**REVERSE**]
limite_inferior ... *limite_superior* **LOOP**
instruccion1;
instruccion2;
...
END LOOP;

- Utilice un loop FOR para “ahorrarse” la prueba del número de repeticiones.
- No declare explícitamente el índice; éste es declarado implícitamente.

Ejemplo de *Loop* **FOR**

Usando el ejemplo del *loop* básico:

```
BEGIN
   FOR v_contador IN 1..5 LOOP
      INSERT INTO SECCION(codasig, numsec)
      VALUES('AA0002', v_contador);
...
   END LOOP;
END;
```

Loop **WHILE**

WHILE *condicion* **LOOP**
instruccion1;
instruccion2;
…
END LOOP;

- Use el *loop* WHILE para repetir un conjunto de instrucciones mientras una condición sea verdad.
- El *loop* termina cuando la condición sea falsa.
- Si la condición es falsa al iniciar el *loop*, no se realizan repeticiones.
- La condición es evaluada al principio de cada iteración.

Ejemplo de *Loop* **WHILE**

“Inserte secciones nuevas para la asignatura ‘AA0002’ mientras no se sobrepase un total de 8 secciones. (Suponga que se pueden haber ingresado secciones de las asignatura anteriormente) ”

```
DECLARE
   v_numsecciones NUMERIC(2);
BEGIN
   SELECT COUNT(*) INTO v_numsecciones
   FROM SECCION WHERE codasig = ‘AA0002’;
   WHILE v_numsecciones <= 8 LOOP
    v_numsecciones := v_numsecciones + 1;
    INSERT INTO SECCION(codasig, numsec)
      VALUES('AA0002', v_numsecciones);
   END LOOP;
END;
```

Tipos de datos estructurados **RECORD**

- Un **RECORD** es un grupo de elementos de datos relacionados, cada uno con su nombre y tipo de datos.
- Una de sus utilidades principales es la recuperación y procesamiento de filas de una tabla.

nombre_tipo **RECORD**

(*declaracion_campo* [, d*eclaracion_campo*]…);

identificador *nombre_tipo*;

Ejemplo de **RECORD**

```
DECLARE
   tipo_reg_emp RECORD
      (nombre VARCHAR(10),
       cargo VARCHAR(12),
       sueldo NUMERIC(7,2));
      reg_emp  tipo_reg_emp;
BEGIN
   reg_emp.cargo := ‘Programador’;
END;
```

Tipos de datos estructurados **%ROWTYPE**

- Se puede declarar un registro del mismo tipo que la fila de una tabla o vista de la Base de Datos.
- Se coloca ‘nombre_tabla.**%ROWTYPE’**.
- Los campos del registro toman sus nombres y tipos de datos de las columnas de la tabla o vista.

Ejemplo de **%ROWTYPE**

```
DECLARE
   reg_est   ESTUDIANTE.%ROWTYPE;
BEGIN
   SELECT * INTO reg_est
   FROM ESTUDIANTE
   WHERE carnet = ‘9912345’;
      …..
END;
```

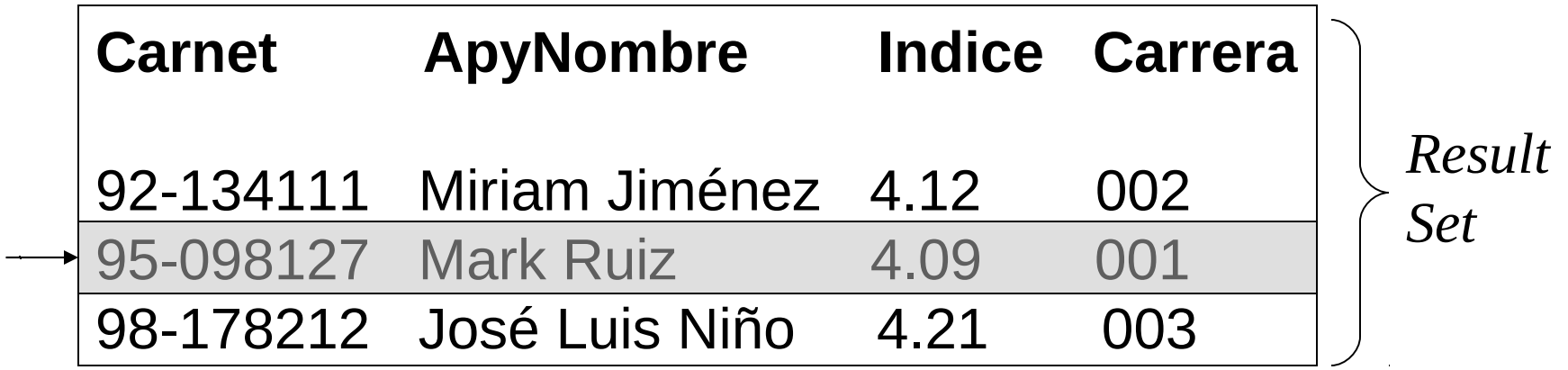
Cursores explícitos

- Son utilizados para consultas que retornan más de una fila. Al conjunto de filas se le llama *result set*.
- El tamaño del *result set* es el número de filas que satisfacen el criterio de selección de la consulta
  - Son declarados por el programador y manipulados a través de determinadas instrucciones.
  - El cursor “apunta” a una determinada fila del *result set* llamada *current row*.

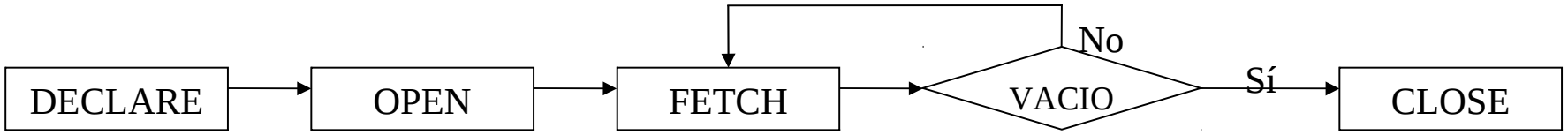
Ejemplo de cursores explícitos

Ejemplo:

"Estudiantes con índice mayor que 4"



Cursores explícitos



Crear un cursor mediante una consulta

Ejecutar la consulta y crear el *result set*

Cargar *el current row* en una variable

Mover el cursor a la siguiente fila

Preguntar si el cursor apunta a alguna fila

Cerrar el cursor

Sintaxis

Las diferentes instrucciones para el manejo de cursores son:

CURSOR *nombre_cursor* **IS**

instruccion_select;

OPEN *nombre_cursor*;

FETCH *nombre_cursor* **INTO**

[*variable* [,*variable*, …….] | *nombre_registro*];

CLOSE *nombre_cursor* ;

Ejemplo

“Asigne a todos los estudiantes de la carrera ‘001’ a la sección 1 de la asignatura ‘AA0001’”

```
DECLARE
   CURSOR e1 IS SELECT * FROM ESTUDIANTE
      WHERE idcarrera = ‘001’;
   reg_est ESTUDIANTE.%ROWTYPE;
BEGIN
   OPEN e1;
   FETCH e1 INTO reg_est;
   WHILE e1.%FOUND LOOP
      INSERT INTO ASIGNA VALUES (e1.carnet, ‘AA0001’, 1);
      FETCH e1 INTO reg_est;
   END LOOP;
   CLOSE e1;
END;
```

Atributos

Atributo	Tipo	Descripción
%**ISOPEN**	Boolean	Es TRUE si el cursor está abierto.
%**NOTFOUND**	Boolean	Es TRUE si el *fetch* más reciente no retorna fila.
%**FOUND**	Boolean	Es TRUE si el *fetch* más reciente retorna una fila.
%**ROWCOUNT**	Number	Es el número total de filas retornadas.

Loop de cursor

FOR *nombre_registro* **IN** *nombre_cursor* **LOOP**
instruccion1;
instruccion2;
….
END LOOP;

- Forma resumida de procesar cursores.
- El *open*, *fetch* y *close* están implícitos.
- El *record* no debe declararse.

Ejemplo de *Loop* de cursor

Usando el ejemplo anterior: “Asigne a todos los estudiantes de la carrera ‘001’ a la sección 1 de la asignatura ‘AA0001’”

```
DECLARE
  CURSOR e1 IS SELECT * FROM ESTUDIANTE
  WHERE idcarrera = ‘001’;
BEGIN
  FOR reg_est IN e1 LOOP
    INSERT INTO ASIGNA VALUES (e1.carnet,‘AA0001’, 1);
  END LOOP;
END;
```

Cursores con parámetros

CURSOR *nombre_cursor*

[(*nombre_parametro tipo_datos*

[,*nombre_parametro tipo_datos*….])]

IS

instruccion_select;

Ejemplo de uso de cursores

“Asigne a todos los estudiantes de las carrera ‘001’ y ‘002’ a la sección 1 de la asignatura ‘AA0001’”

```
DECLARE
  CURSOR e1 (v_idcarrera CHAR) IS
  SELECT * FROM ESTUDIANTE WHERE idcarrera =
  v_idcarrera;
BEGIN
  FOR reg_est IN e1(‘001’) LOOP
    INSERT INTO ASIGNA VALUES (e1.carnet, ‘AA0001’, 1);
  END LOOP;
  FOR reg_est IN e1(‘002’) LOOP
    INSERT INTO ASIGNA VALUES (e1.carnet, ‘AA0001’, 1);
  END LOOP;
END;
```

Actualización

Para actualizar (**UPDATE**) alguna fila de un cursor se debe incluir la cláusula **FOR UPDATE** en la instrucción **SELECT** del cursor y utilizar la cláusula **WHERE CURRENT OF** para referenciar el *current row* de un cursor explícito.

Ejemplo:

"Actualice el número de créditos inscritos de todos los estudiantes"

Ejemplo de Actualización

```
DECLARE
  v_credins ESTUDIANTE.credins.%TYPE;
  CURSOR e1 IS SELECT * FROM ESTUDIANTE FOR UPDATE;
BEGIN
  FOR reg_est IN e1 LOOP
    SELECT SUM(creditos) INTO v_credins
    FROM SE_ASIGNA S, ASIGNATURA A
    WHERE S.carnet = e1.carnet AND S.codasig = A.codasig;
    IF SQL%FOUND THEN
      UPDATE ESTUDIANTE SET credins = v_credins
      WHERE CURRENT OF e1;
    END IF;
  END LOOP;
END;
```

Procedimientos

El procedimiento es un bloque con nombre que acepta parámetros.

CREATE OR REPLACE FUNCTION

nombre_proc([*parametro tipo_dato*

[,*parametro tipo_dato* ...]]

RETURNS VOID AS $$

[declaraciones definidas por usuario]

BEGIN

….

[**EXCEPTION**]

…

END;

$$ LANGUAGE plpgsql;

NOTA: La separación impuesta por $$ seguido por LANGUAGE plpgsql indica al manejador que el bloque es de este tipo

Funciones

La función es un bloque con nombre que acepta parámetros y retorna un valor.

CREATE OR REPLACE FUNCTION
nombre_func([*parametro tipo_dato*
[,*parametro tipo_dato ...*]]
RETURNS *tipo_datos* AS $$
[declaraciones definidas por usuario]
BEGIN
….
RETURN valor;
[**EXCEPTION**]
…
END;
$$ LANGUAGE plpgsql;

Activación de Triggers en PostgreSQL

- Eventos: Operaciones de actualización
  - INSERT, UPDTE, DELETE
- Propagación: Inmediata con dos opciones
  - Antes de la operación (BEFORE)
  - Luego de la operación (AFTER)
- Granularidad: Dos opciones
  - Por Instrucción, una vez para toda la operación
  - Por fila, una vez por cada registro (FOR EACH ROW)
- Resolución de conflictos
  - Arbitraria por el sistema

Sintaxis de Triggers en PostgreSQL

```
CREATE FUNCTION nombreFuncion() RETURNS trigger AS $nombreTrigger$
BEGIN
.....
END;
$nombreTrigger$ LANGUAGE plpgsql;
```

```
CREATE [OR REPLACE] TRIGGER nombreTrigger [BEFORE|AFTER] <suceso>
ON TABLA
[FOR EACH ROW] EXECUTE PROCEDURE nombreFuncion();
```

Triggers en PostgreSQL (ejemplo)

```
CREATE Function Reorder() RETURNS TRIGGER AS $Reorder$
DECLARE NUMERIC x
BEGIN
      SELECT COUNT (*) INTO x
      FROM PendingOrders
      WHERE Part=New.Part;
      IF x=0 THEN
            INSERT INTO PendingOrders VALUES
              (New.Part,New.ReorderQuantity,SYSDATE)
      ENDIF;
END;
$Reorder$ LANGUAGE plpgsql;

CREATE TRIGGER Reorder AFTER UPDATE OF PartOnHand
    ON InventoryWHEN (New.PartOnHand<New.ReorderPoint)
        FOR EACH ROW EXECUTE PROCEDURE Reorder();
```

Sobre la Ejecución de Triggers en PostgreSQL

- El cuerpo del trigger puede tener operaciones que disparen otros triggers
- En ese caso se suspende la ejecución del trigger actual y se anida la ejecución del otro trigger
- No hay manera de predecir si la cadena de anidamientos es finita, por lo que se limita a 32
- La resolucción de conflicto es arbitraria, por lo que debe hacerse una programación que no dependa del orden de ejecución entre triggers en conflicto