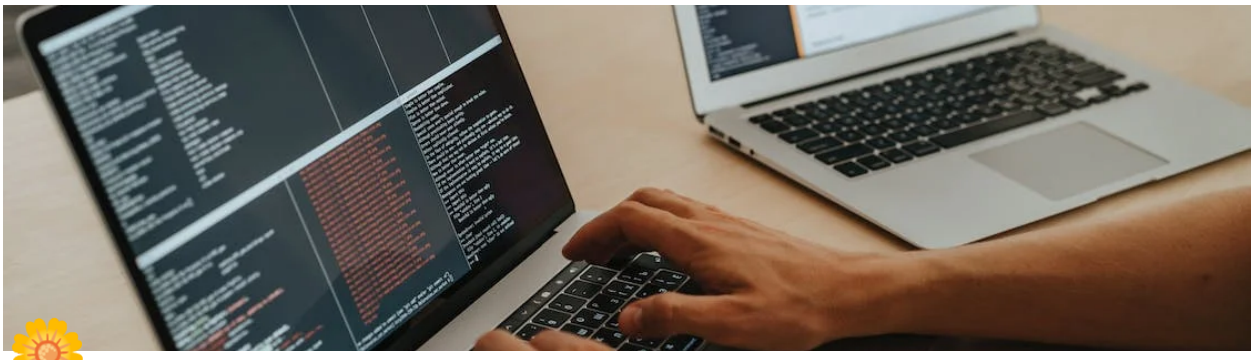


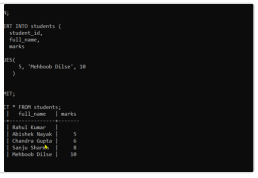
7 Funciones plpgSQL avanzadas

Links de interés:

PostgreSQL - BEGIN - GeeksforGeeks

A Computer Science portal for geeks. It contains well written, well thought and well explained computer science and

https://geeksforgeeks.org/postgresql-begin/



Capítulo 8 Modelo relacional | Base de Datos: Una aproximación a la inf

Manual de referencia en bases de datos, desde un aspecto introductorio y forma p una serie de libros sobre datos e información.

https://bookdown.org/paranedagarcia/database/modelo-relacional.html

MOVE

MOVE MOVE — position a cursor Synopsis MOVE [direction] [FROM | IN] cursor_name where direction can …

https://www.postgresql.org/docs/current/sql-move.html

PostgreSQL CHECK Constraint

In this tutorial, we will introduce you to PostgreSQL CHECK constraint to constrain the value of columns in a table based on a Boolean expression.

https://postgresqltutorial.com/postgresql-tutorial/postgresql-check-constraint/

CREATE TRIGGER

CREATE TRIGGER CREATE TRIGGER — define a new trigger Synopsis CREATE [OR REPLACE] [CONSTRAINT]

https://www.postgresql.org/docs/current/sql-createtrigger.html

Understanding PostgreSQL Triggers: A Comprehensive 101 Guide - Lea

This article provides a comprehensive guide on PostgreSQL Triggers, different ope associated with them and the example queries to implement them.

https://hevodata.com/learn/postgresql-triggers/

Ampliación de conceptos:

-Mirar lo que es una transacción en bases de datos (no lo ha explicado bien).

-Mirar lo que es una variable tipo cursor en bases de datos

-Vamos a dar hasta triggers

-Mirar como blindar la base de datos con triggers

-Si se elimina y se vuelve a crear una función, esta nueva función no es el mismo objeto que la vieja, por tanto, tendrán que eliminarse las reglas, vistas y disparadores existentes que hacían referencia a la antigua función.

FUNCIONES PLPGSQL

CREACIÓN DE FUNCIONES:

Cualquier función tiene esta estructura:

```
CREATE OR REPLACE FUNCTION nombreFunción( modoParámetro nombreParámetro/s tipoParámetro/s ) RETURNS tipoDatoQueDevuelve/queTablaDevuelve AS
$$
    Operación
$$LANGUAGE tipoFunción ;
```

Pero la estructura de las Operaciones en funciones plpgsql es la siguiente:

```
$$
DECLARE(Opcional)
    Se declaran variables que no son parámetros pero se usan en la operación
BEGIN
    Aquí van las operaciones
END;
$$
```

NOTA: Las sentencias se terminan con punto y coma al final de cada línea y para retornar el resultado se usa la palabra reservada RETURN

Ej: Crear una función sumar() que introduzcamos dos números enteros y nos devuelva su suma

```
CREATE OR REPLACE FUNCTION sumar(valor1 int, valor2 int) RETURNS int AS
$$
```

Empezamos creando una función *sumar()*, con *dos parámetros int* (CUIDADO CON EL ORDEN, ES A LA INVERSA DE JAVA) llamados valor1 y valor2 y nos va a *devolver un valor int que como no es un valor void, tenemos que poner si o si el operador RETURN. Como no tenemos que crear una variable nueva para la operación, no necesitamos DECLARE.*

```
BEGIN
        RETURN $1 + $2;
END;
$$ LANGUAGE plpgsql;
```

Como sabemos que no es una operación CRUD, el tipoFunción es LANGUAGE plpgsql, por lo que después de los primeros $$ hay que indicar cuando empieza la operación con **BEGIN** y antes de los últimos $$ hay que indicar que finaliza la operación con **END;**

Para hacer la operación entre BEGIN y END, tenemos que indicar en algún punto lo que devuelve con **RETURN**. En este ejemplo como es una operación muy simple, solo tenemos que devolver la propia suma.

Ej: Crea la misma función sumar() pero ahora con una variable suma

```sql
CREATE OR REPLACE FUNCTION sumar(num1 int, num2 int) RETURNS int AS
$$
DECLARE
    suma int;
BEGIN
    suma := num1 + num2;
    RETURN suma;
END;
$$ LANGUAGE plpgsql;
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 60 msec.
```

Ahora vemos que sí que necesitamos el DECLARE para crear la variable suma que es un entero. En la operación le asignamos el valor suma con "**:=**" a la suma del num1 y del num2; por último, hacemos que nos devuelva la suma.

ANTES DE COMENZAR CON LA SINTAXIS

COMENTARIOS

Para hacer un comentario en una función, se ponen dos guiones al principio:

```
-- Comentario
```

MENSAJES

En las funciones plpgsql utilizamos la cláusula **RAISE** para generar mensajes de error, advertencia o información en función de la necesidad. Podemos utilizar diferentes niveles de registro para los mensajes que generamos utilizando RAISE. La diferencia entre estos niveles es la gravedad del mensaje y cómo se manejan los mensajes en el registro de PostgreSQL. Aquí están los niveles organizados desde más a menos importancia:

- **NOTICE**: Este nivel se utiliza para mensajes informativos o notificaciones que no son necesariamente un problema o un error. Por ejemplo, podemos usar NOTICE para informar sobre el inicio de una función o un evento que no afecta la ejecución del programa.
- **DEBUG**: Este nivel se utiliza para mensajes de depuración que nos ayudan a entender mejor el código y encontrar posibles problemas o errores. Los mensajes DEBUG son muy detallados y suelen ser desactivados en producción.
- **LOG**: Este nivel se utiliza para mensajes de registro que no son necesariamente un error, pero que es importante registrar o dejar constancia del mensaje o error. Los mensajes de LOG se guardan en el registro de PostgreSQL, que se puede utilizar para analizar los problemas que ocurrieron en el sistema.
- **INFO**: Este nivel se utiliza para mensajes informativos más importantes que NOTICE. Por ejemplo, podemos usar INFO para informar sobre una operación importante que se acaba de completar o un cambio significativo en el sistema.
- **WARNING**: Este nivel se utiliza para mensajes que indican una situación potencialmente peligrosa o que pueden causar problemas en el futuro, es una advertencia. Por ejemplo, podemos usar WARNING para informar que se está utilizando una función obsoleta o que un parámetro de entrada es incorrecto.
- **EXCEPTION**: Este nivel se utiliza para generar errores que interrumpen la ejecución del programa. Cuando se genera una excepción, el programa se detiene, se devuelve el mensaje de error correspondiente al usuario y se cancelan todas las operaciones realizadas previamente en la función.

NOTA: Los más utilizados son RAISE NOTICE, RAISE WARNING y RAISE EXCEPTION

**Veremos ejemplos utilizando los mensajes más adelante*

SINTAXIS DE LAS DIFERENTES PARTES DE LA FUNCION PLPGSQL

DECLARE:

1.-Para crear una variable en declare la sintaxis es la siguiente (rojo imprescindible, violeta opcional):

`nombreVariable [CONSTANT] tipoVariable [NOT NULL] [{DEFAULT / := } valor] ;`

Mínimo se necesita: **nombreVariable tipoVariable;** por ejemplo: `suma int;`

Se puede añadir dependiendo de la variable que se quiera crear:

- **:=/DEFAULT valor:** asigna un valor a la variable que puede cambiar durante la operación de la función, por lo que es un valor inicial. Ej: `num4 numeric:=4;` o `num4 numeric DEFAULT 4;`
- **CONSTANT:** especifica que la variable es constante, por lo que el valor inicial asignado a la variable es el valor con que se mantendrá durante toda la función. De no ser especificada la variable es inicializada con el valor nulo. Ej: `pi CONSTANT float :=3.1415;`
- **NOT NULL:** especifica que la variable no puede tener asignado un valor nulo (generándose un error en tiempo de ejecución en caso de que ocurra). Todas las variables declaradas como no nulas deben tener especificado un valor no nulo. Ej: `fecha date NOT NULL :='2023-02-24'` este es un ejemplo válido y este es uno inválido: ~~`fecha date NOT NULL;`~~

2.-Para crear una variable que haga referencia a un parámetro de la función (un alias) la sintaxis es:

`nombreAlias ALIAS FOR $número/nombreParametro;`

Ej: En la función sumar()

```
1  CREATE OR REPLACE FUNCTION sumar(num1 int, num2 int) RETURNS int AS
2  $$
3  DECLARE
4      numero1 ALIAS FOR $1;
5  BEGIN
6      RETURN numero1 + num2;
7  END;
8  $$ LANGUAGE plpgsql;
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 52 msec.
```

Ahora el primer parámetro de la función: $1 ahora se puede llamar num1 o numero1. (S*inceramente no lo usaría, pero está en los apuntes*)

3.-También se pueden definir cursores en el DECLARE, pero todo eso lo vamos a ver a continuación en el siguiente pdf

BEGIN … END;

Aquí es donde se hacen todas las operaciones y cuando decimos operaciones, no es solo entre parámetros sino que también son operaciones en una propia consulta

CREACIÓN:

Ej: Crear la funcion duplicarprecio en donde se le pase el precio de un producto y nos devuelva el precio multiplicado por dos

```
1  CREATE FUNCTION duplicarprecio(precio numeric) RETURNS numeric AS
2  $$
3  BEGIN
4      RETURN (SELECT buyprice FROM products WHERE buyprice = precio) * 2;
5  END;
6  $$ LANGUAGE plpgsql;
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 75 msec.
```

Creamos una función a la que le pasemos un parámetro numérico y nos devuelva un valor numérico. En el RETURN hacemos que la consulta “SELECT buyprice FROM products WHERE buyprice = precio” que devuelve el valor del buyprice (un número), se multiplique por 2

EJECUCIÓN:

```
10  SELECT duplicarprecio(15.91);
```

Data Output | Messages | Notifications

	duplicarprecio numeric
1	31.82

Ponemos el precio del producto más barato y vemos que nos devuelve su precio multiplicado por dos, pero al ser una consulta SELECT, el valor no cambia en el registro, solo se consulta cómo quedaría el precio duplicado. Si hiciéramos un UPDATE en la operación, el precio sí que se vería cambiado.

ESTRUCTURAS DE CONTROL

ESTRUCTURAS CONDICIONALES:

IF…THEN…END IF;

La sintaxis de esta estructura es:

`IF expresionBooleana THEN sentencia END IF;` Si la expresión booleana es true, se cumple la sentencia. Nivel 1

`IF expresionBooleana THEN sentencia1 ELSE sentencia2 END IF;` Si la expresión booleana es true se cumple la sentencia1 y si no, se cumple la sentencia2. Nivel 2

`IF expresionBooleana1 THEN sentencia1 [ELSIF expresionBooleana2 THEN sentencia2][ELSE sentencia3] END IF;` Si la expresión booleana 1 es true se cumple la sentencia 1, si no es true, se mira si la expresión booleana 2 es true (si lo es, se cumple la sentencia 2) y si es false, entonces se cumple la sentencia 3. Se pueden hacer tantas cláusulas ELSIF como se quiera. Nivel 3

Ejemplo Nivel 1: Crear una función que diga si un número es mayor que 10 o nos devuelva null si no es mayor que 10

```
1  CREATE OR REPLACE FUNCTION nivel1(num int) RETURNS text AS
2  $$
3  DECLARE
4      resultado text;
5  BEGIN
6      IF num > 10 THEN
7          resultado := 'El número es mayor que 10';
8      END IF;
9      RETURN resultado;
10 END;
11 $$ LANGUAGE plpgsql;
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 69 msec.
```

Ahora la ejecutamos con los dos casos (parámetro mayor que 10 y parámetro menor que 10)

```
13 SELECT nivel1(11);
14
```

Data Output | Messages | Notification

	nivel1 text
1	El número es mayor que 10

```
16 SELECT nivel1(5);
```

Data Output | Messages | Notifications

	nivel1 text
1	[null]

Ejemplo Nivel 2: Crea una función que al introducir un número nos de si es mayor que 10 o si es igual o menor que 10.

```sql
CREATE OR REPLACE FUNCTION tnivel2(num int) RETURNS text AS
$$
DECLARE
    resultado text;
BEGIN
    IF num > 10 THEN
        resultado := 'El número es mayor que 10';
    ELSE
        resultado := 'El número es menor o igual que 10';
    END IF;
    RETURN resultado;
END;
$$ LANGUAGE plpgsql;
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 81 msec.
```

Ahora la ejecutamos con los tres casos (mayor o menor o igual que 10)

```sql
SELECT tnivel2(11)
```

Data Output | Messages | Notificati

	tnivel2 text
1	El número es mayor que 10

```sql
SELECT tnivel2(10)
```

Data Output | Messages | Notification

	tnivel2 text
1	El número es menor o igual que 10

```sql
SELECT tnivel2(0)
```

Data Output | Messages | Notifications

	tnivel2 text
1	El número es menor o igual que 10

Ejemplo Nivel 3: Crear una función que nos diga si el número introducido es mayor de 10, menor de 10 o igual a 10.

```
1  CREATE OR REPLACE FUNCTION nivel3(num int) RETURNS text AS
2  $$
3  DECLARE
4      resultado text;
5  BEGIN
6      IF num > 10 THEN
7          resultado := 'El número es mayor que 10';
8      ELSIF num < 10 THEN
9          resultado := 'El número es menor que 10';
10     ELSE
11         resultado := 'El número es igual a 10';
12     END IF;
13     RETURN resultado;
14 END;
15 $$ LANGUAGE plpgsql;
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 72 msec.
```

Ahora la ejecutamos con los tres casos (mayor, menor o igual que 10)

```
21 SELECT nivel3(11)
```

Data Output | Messages | Notifications

	nivel3 text
1	El número es mayor que 10

```
1 SELECT nivel3(10)
```

Data Output | Messages | Notifications

	nivel3 text
1	El número es igual a 10

```
1 SELECT nivel3(1)
```

Data Output | Messages | Notifications

	nivel3 text
1	El número es menor que 10

CASE…WHEN…END CASE;

Hay dos tipos de CASE y su sintaxis de estructura es:

CASE SIMPLE: para realizar comparaciones directas en una expresión

```
CASE expresión/expresion := CASE
WHEN valor1 THEN resultado1
WHEN valor2 THEN resultado2
...
ELSE resultadoPorDefecto
END CASE;
```

CASE DE BÚSQUEDA: se utiliza para buscar un valor dentro de una tabla o subconsulta y devolver un resultado correspondiente

```
CASE
WHEN condición1 THEN resultado1
WHEN condición2 THEN resultado2
...
ELSE resultadoPorDefecto
END CASE;
```

Ejemplo CASE SIMPLE: Crea una función en la que introduzca un número y me de si es positivo, negativo o 0

```
1  CREATE OR REPLACE FUNCTION casesimple(num int) RETURNS text AS
2  $$
3  DECLARE
4      resultado text;
5  BEGIN
6      resultado := CASE
7          WHEN num > 0 THEN 'El número es positivo'
8          WHEN num < 0 THEN 'El número es negativo'
9          ELSE 'El número es cero'
10     END;
11     RETURN result;
12 END;
13 $$ LANGUAGE plpgsql;
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 68 msec.
```

Ejemplo CASE SIMPLE: Dime si un empleado está en las oficinas de San Francisco o Boston y si no, dímelo también. (El officecode de San Francisco es 1 y el de Boston es 2)

```
1  CREATE OR REPLACE FUNCTION casesimple2(numeroOficina int) RETURNS VOID AS
2  $$
3  BEGIN
4      CASE numeroOficina
5          WHEN 1 THEN
6              RAISE NOTICE 'El empleado es de la oficina de San Francisco';
7          WHEN 2 THEN
8              RAISE NOTICE 'El empleado es de la oficina de Boston';
9          ELSE
10             RAISE NOTICE 'El empleado no es ni de la oficina de San Francisco ni la
11     END CASE;
12 END;
13 $$ LANGUAGE plpgsql;
14
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 54 msec.
```

Ahora lo ejecutamos

SELECT casesimple2((select e.officecode::integer from employees e where e.employeenumber=1002))

*NOTA: Ponemos **e.officecode::integer** porque el officecode no es un int, así que tenemos que hacer el casting*

NOTA: Para hacer un SELECT dentro de la función, hay que poner doble paréntesis

No nos devuelve nada porque devuelve un void

```
15  SELECT casesimple2((select e.officecode::integer from employees e where e.employeen
16
```

Data Output | Messages | Notifications

	casesimple2 void
1	

Pero si vamos a “Messages” veremos lo que nos dice:

Data Output | Messages | Notifications

```
NOTICE:  El empleado es de la oficina de San Francisco
```

Si sustituyéramos los RASE NOTICE por RAISE WARNING el resultado nos daría esto:

Data Output | Messages | Notifications

```
WARNING:  El empleado es de la oficina de San Francisco
```

Ejemplo CASE DE BÚSQUEDA: Haz la misma función pero utilizando el CASE de búsqueda

```
1  CREATE OR REPLACE FUNCTION casebusqueda(numeroOficina int) RETURNS VOID AS
2  $$
3  BEGIN
4      CASE
5          WHEN (numeroOficina = 1) THEN
6              RAISE NOTICE 'El empleado es de la oficina de San Francisco';
7          WHEN (numeroOficina = 2) THEN
8              RAISE NOTICE 'El empleado es de la oficina de Boston';
9          ELSE
10             RAISE NOTICE 'El empleado no es ni de la oficina de San Francisco ni la
11     END CASE;
12 END;
13 $$ LANGUAGE plpgsql;
```

NOTA: Ten cuidado de no poner los atributos de la tabla en vez de los parámetros

Lo ejecutamos buscando un employee con officecode=2 para que salga Boston

```
15  SELECT casebusqueda((select e.officecode::integer from employees e where e.employee
16
```

Data Output | Messages | Notifications

```
NOTICE:  El empleado es de la oficina de Boston
```

Ejemplo de CASES más complejos:

1.- Función que pasa los códigos de estado a los nombres completos del estado

```
CREATE OR REPLACE FUNCTION nombreCompletoEstado(codigoestado varchar(50))
RETURNS varchar(50) AS $$
DECLARE
    nombrestado varchar(50);
BEGIN
    SELECT CASE
              WHEN codigoestado = 'CA' THEN 'California'
              WHEN codigoestado = 'MA' THEN 'Massachusetts'
              WHEN codigoestado = 'NY' THEN 'Nueva York'
              ELSE 'No lo sé'
           END CASE
    INTO nombrestado;

    RETURN nombrestado;
END;
$$ LANGUAGE plpgsql;
```

1. Se define una nueva función llamada "nombreCompletoEstado" que recibe un parámetro llamado "codigoestado" de tipo varchar(50) y que devuelve un resultado de tipo varchar(50).
2. Se definen las variables locales que se van a utilizar en la función. En este caso, se define una variable llamada "nombrestado" de tipo varchar(50).
3. Se utiliza la cláusula "SELECT CASE" para definir las diferentes opciones que se pueden dar para el parámetro "codigoestado". En este caso, si el valor es "CA", se asigna a "nombrestado" el valor "California". Si el valor es "MA", se asigna a "nombrestado" el valor "Massachusetts". Si el valor es "NY", se asigna a "nombrestado" el valor "Nueva York". Si el valor es cualquier otro, se asigna a "nombrestado" el valor "No lo sé".
4. Se devuelve el valor de la variable "nombrestado".
5. Se indica que el lenguaje utilizado en la función es "plpgsql".

En PostgreSQL, la sintaxis de CASE se utiliza en consultas SQL regulares para evaluar una expresión y devolver un valor condicionalmente, por ejemplo:

```
SELECT column1, column2,
       CASE WHEN column3 > 0 THEN 'positive'
            WHEN column3 < 0 THEN 'negative'
            ELSE 'zero'
       END AS sign
FROM my_table;
```

Por otro lado, en una función PL/pgSQL, **para asignar el resultado de una expresión condicional CASE a una variable local, es necesario utilizar la sintaxis SELECT CASE,** como se muestra en el siguiente ejemplo:

```
SELECT CASE
          WHEN variable = 1 THEN 'one'
          WHEN variable = 2 THEN 'two'
          ELSE 'unknown'
       END CASE
INTO result_variable;
```

NOTA: Si queremos hacer un INTO, tenemos que usar SELECT CASE.

NOTA: Cuidado con los ; porque en END CASE no se puede poner, si no, se entiende que SELECT también se cierra y entonces INTO nombrestado; no tiene sentido

```
20  SELECT nombreCompletoEstado('CA')
```

Data Output | Messages | Notifications

	nombrecompletoestado character varying
1	California

2.-Función a la que le de un número y me diga el día de la semana al que le corresponde

```
CREATE OR REPLACE FUNCTION diaSemana(num int) RETURNS text AS
$$
BEGIN
    RETURN CASE
        WHEN num = 1 THEN 'Lunes'
        WHEN num = 2 THEN 'Martes'
        WHEN num = 3 THEN 'Miércoles'
        WHEN num = 4 THEN 'Jueves'
        WHEN num = 5 THEN 'Viernes'
        WHEN num = 6 THEN 'Sábado'
        WHEN num = 7 THEN 'Domingo'
        ELSE 'Número de día no válido'
    END CASE;
END;
$$ LANGUAGE plpgsql;
```

Es muy importante **RETURN CASE** porque se deben comparar los valores de "num" con números enteros en lugar de cadenas de texto, así que cuando hacemos el THEN, si no ponemos RETURN, le damos el valor de una cadena de texto en vez de un número. Si añadimos "RETURN" antes de la cláusula "CASE" devuelve el resultado del caso (el valor que le damos cuando usamos THEN)

NOTA: Cuidado con las '' ya que si usamos '' entre los números, no funciona la función

```
17  SELECT diaSemana('1')
18
```

Data Output | Messages | Notifications

	diasemana text
1	Lunes

```
17  SELECT diaSemana('10')
18
```

Data Output | Messages | Notifications

	diasemana text
1	Número de día no válido

ESTRUCTURAS ITERATIVAS:

Existen 4 tipos de bucles:

LOOP Simple: `LOOP sentencia END LOOP;` es un ciclo incondicional que es repetido indefinidamente hasta que encuentra una sentencia EXIT o RETURN.

WHILE LOOP: `WHILE expresiónBooleana LOOP sentencia END LOOP;` repite una secuencia de sentencias mientras la expresiónBooleana sea verdadera, esta expresión es chequeada antes de cada entrada en el ciclo LOOP.

FOR LOOP: `FOR nombre IN [REVERSE] expresión [BY expresión ] LOOP sentencia END LOOP;` itera sobre un rango de valores enteros. La variable nombre es automáticamente definida de tipo entero y existe solamente dentro del ciclo. Hay 2 expresiones que delimitan los límites inferior y superior del rango; la cláusula BY especifica el incremento en cada iteración (por defecto 1); la cláusula REVERSE especifica que en lugar de incrementarse el valor a iterar se decrementa. Las expresiones son evaluadas en cada entrada al ciclo LOOP.

FOR EACH LOOP: `FOR variable IN consulta LOOP sentencia END LOOP;` itera por el resultado de una consulta, y manipular los datos de cada tupla. La variable es de tipo RECORD, fila o un listado de variables escalares separadas por coma.

Y también hay dos estructuras iterativas:

CONTINUE: se utiliza dentro de un bucle para saltar a la siguiente iteración del bucle sin ejecutar el código restante dentro de este para la iteración actual. Es como decir "*olvida esta iteración, salta a la siguiente*". Por ejemplo, supongamos que queremos imprimir los números del 1 al 10, pero solo queremos imprimir los números pares. Podemos usar CONTINUE para saltar los números impares y seguir con los pares.

EXIT: se utiliza dentro de un bucle para salir del bucle antes de que se hayan completado todas las iteraciones. Es como decir "*sal del bucle, ya no necesito continua*r". Por ejemplo, supongamos que queremos imprimir los números del 1 al 10, pero solo queremos imprimir los números hasta el número 5. Podemos usar EXIT para salir del bucle cuando lleguemos al número 5.

Ejemplos de bucles del libro:

LOOP SIMPLE

```
CREATE FUNCTION cambiar_precio(id integer, cant integer) RETURNS numeric AS
$$
DECLARE
        contador integer := 0;
        precio numeric;
BEGIN
        LOOP
                UPDATE products SET price = price * 2 WHERE prod_id = id RETURNING
                price INTO precio;
                contador := contador + 1;
                IF contador = cant THEN
                        EXIT;
                END IF;
        END LOOP;
        RETURN precio;
END;
$$ LANGUAGE plpgsql;
```

Crea una función cambiar_precio a la que se meten dos números enteros, uno que es el id del producto y otro que es la cantidad de veces que queremos que se multipliquen los precios por dos, y se le devuelve un número. Declaramos una variable contador que empieza en 0 y una variable precio. Empezamos el LOOP simple que hace un Update de los productos para que el precio sea

el doble (ya que se hace primero el SET para cambiar los precios y luego se hace el RETURNING price INTO precio para darle el valor del nuevo price a la variable precio), hacemos que el contador vaya sumando 1 cada vez que se repite el LOOP (con la estructura contador:=contador+1;) y luego usamos una cláusula IF de nivel 1 que dice que si el contador es igual al parámetro cantidad, entonces salimos del LOOP, si no, continuamos. Salimos del IF, salimos del LOOP y devolvemos el precio.

En classicmodels podríamos hacer lo mismo con esta función:

```
CREATE FUNCTION cambiar_precio(id varchar(15), cant integer) RETURNS numeric
$$
DECLARE
    contador integer := 0;
    precio numeric;
BEGIN
    LOOP
        UPDATE products SET buyprice = buyprice * 2 WHERE productcode = id
        RETURNING buyprice INTO precio;

        contador := contador + 1;
        IF contador = cant THEN
            EXIT;
        END IF;
    END LOOP;
RETURN precio;
END;
$$ LANGUAGE plpgsql;
```

Si ejecutamos la función así:

SELECT cambiar_precio('S10_1678', '2')

El producto con ese código valía 48.81€ y ahora vale 195.28€ ya que lo hemos multiplicado dos veces por dos

WHILE LOOP

```
CREATE FUNCTION cambiar_precio(id integer, cant integer) RETURNS numeric AS
$$
DECLARE
        contador integer := 0;
        precio numeric;
BEGIN
        WHILE contador < cant LOOP
                UPDATE products SET price = price * 2 WHERE prod_id = id RETURNING
                price INTO precio;
                contador := contador + 1;
        END LOOP;
        RETURN precio;
END;
$$ LANGUAGE plpgsql;
```

Es la misma función pero ahora con un bucle WHILE. Todo se mantiene igual excepto la sentencia de WHILE. Ahora hacemos que mientras el contador sea menor que la cantidad de veces que queramos repetir el bucle, se va a hacer un bucle donde se va a

hacer un update y se va a instanciar el contador como en el anterior bucle. Pero ahora ya no necesitamos la estructura IF porque el propio WHILE nos dice cuando acaba el bucle.

FOR LOOP

```
CREATE FUNCTION datos_producto_setof(integer) RETURNS SETOF products AS
$$
DECLARE
        resultado products;
BEGIN
        FOR resultado IN SELECT * FROM products where prod_id > $1 LOOP
                RETURN NEXT resultado;
        END LOOP;
        RETURN; -- Opcional
END;
$$ LANGUAGE plpgsql;
```

Función que devuelve todos los productos registrados en la base de datos que su identificador sea mayor que el pasado por parámetro. Declaramos la variable resultado que contendrá los valores de una fila de la tabla products, ya que el tipo de datos products se refiere a la estructura de la tabla products, de esta manera, la variable resultado se utilizará dentro del bucle FOR para almacenar el resultado de cada iteración, y luego se devolverá mediante la instrucción RETURN NEXT, que devuelve la siguiente fila en el conjunto de resultados. Al declarar la variable resultado como tipo products, se asegura que la variable tenga la misma estructura que las filas de la tabla products, lo que permite que la variable almacene y devuelva correctamente los valores de las filas de la tabla.

Luego en el FOR se usa la variable resultado para almacenar el resultado de cada vuelta, e itera a través de todas las filas de la tabla products donde el valor de prod_id es mayor que el valor del parámetro de entrada. Usamos RETURN NEXT para que la función devuelva un conjunto de resultados fila por fila a medida que el bucle itera a través de las filas de la tabla.

RETORNO DE VALORES

En las funciones SQL los valores se devolvían o bien con una consulta SELECT o bien con un RETURNING. En las funciones plpgsql hay tres comandos para devolver valores (aunque RETURN NEXT y RETURN QUERY se consideran muy similares):

- **RETURN + valor/variable**: se utiliza para devolver un solo valor de la función. Es útil cuando la función debe devolver un valor como un número o una cadena. El RETURN finaliza la ejecución de la función, por lo que no se ejecutará ningún código después del RETURN. Ej:

```
CREATE OR REPLACE FUNCTION square(number integer) RETURNS integer AS $$
BEGIN
  RETURN number * number;
END;
$$ LANGUAGE plpgsql;
```

- **RETURN NEXT + valor/variable**: se utiliza para devolver una fila completa de una consulta. Esta instrucción es útil cuando se está generando un conjunto de resultados en la función y se quiere devolver una fila a la vez. La función puede llamar a la instrucción RETURN NEXT varias veces para devolver múltiples filas. La función debe finalizar llamando a la instrucción

RETURN para indicar que se han devuelto todos los resultados. Ej: La función utiliza el RETURN NEXT para devolver el siguiente número de la secuencia cada vez que se llama a la función.

```
CREATE OR REPLACE FUNCTION next_number(start integer, end integer) RETURNS integer AS $$
DECLARE
  i integer := start;
BEGIN
  IF i <= end THEN
    RETURN NEXT i;
    i := i + 1;
  END IF;
  RETURN i;
END;
$$ LANGUAGE plpgsql;
```

- **RETURN QUERY + consulta**: se utiliza para devolver múltiples filas de una consulta. Esta instrucción es útil cuando se está generando un conjunto de resultados en la función y se quiere devolver todas las filas al mismo tiempo. RETURN QUERY toma como argumento una consulta que devuelve un conjunto de resultados. La función debe finalizar llamando a la instrucción RETURN para indicar que se han devuelto todos los resultados. Ej: la función utiliza SELECT COUNT(*) INTO user_count para obtener el número total de usuarios en la tabla users. Luego, utiliza la instrucción RETURN QUERY para devolver el valor de user_count.

```
CREATE OR REPLACE FUNCTION count_users() RETURNS integer AS $$
DECLARE
  user_count integer;
BEGIN
  SELECT COUNT(*) INTO user_count FROM users;
  RETURN QUERY SELECT user_count;
END;
$$ LANGUAGE plpgsql;
```

RETURNS VS RETURNS SETOF

- **RETURNS** se utiliza para especificar el tipo de valor que devuelve la función. Por ejemplo, si una función devuelve un valor entero, se utilizaría la cláusula RETURNS INTEGER.
- **RETURNS SETOF** se utiliza para especificar que una función devuelve un conjunto de valores. Es decir, que la función retornará varias filas de datos, como si fuera una tabla. En este caso, se utiliza la cláusula SETOF seguida del tipo de datos que conforman cada fila.

Por ejemplo, si una función retorna una tabla con dos columnas (nombre y edad), se podría utilizar la cláusula RETURNS SETOF seguida de la definición del tipo de datos de la fila:

`...RETURNS SETOF TABLE (nombre VARCHAR, edad INTEGER)`

`...RETURNS SETOF customers`

E incluso esto:

```
CREATE TYPE tipo_aviso AS (email VARCHAR,adeudo decimal);
...RETURNS SETOF tipo_aviso
```

DIFERENTES TIPOS DE modoParámetro

Como hablamos en el anterior FP, hay varios modos de parámetros

`CREATE OR REPLACE FUNCTION nombreFunción( modoParámetro nombreParámetro/s tipoParámetro/s ) RETURNS tipoDatoQueDevuelve/queTablaDevuelve AS`

`$$`

`Operación`

`$$LANGUAGE tipoFunción ;`

El modoParámetro puede ser de 4 tipos:

- **IN:** Es el modo por defecto por lo que no hay que escribirlo. Especifica que el parámetro es de entrada, o sea, forma parte de la lista de parámetros con que se invoca a la función y que son
  necesarios para el procesamiento definido en la función. Ej: Una función normal con todos los parámetros en modo IN (que son por defecto con los que hemos trabajado en clase). *Supongamos que queremos crear una función que reciba dos números enteros y los multiplique. Los valores de estos números se pasan a la función como parámetros de entrada. La función no modifica estos valores, solo los utiliza para realizar la operación.*

```
CREATE OR REPLACE FUNCTION multiplicar_numeros(num1 INT, num2 INT) RETURNS INT AS
$$
DECLARE
  resultado INT;
BEGIN
  resultado := num1 * num2;
  RETURN resultado;
END;
$$
LANGUAGE plpgsql;
```

- **OUT:** Es el parámetro de salida, forma parte del resultado de la función y no se incluye en la ejecución de la función. Ej: *Supongamos que queremos crear una función que calcule el cuadrado de un número y lo devuelva como resultado. En este caso, el valor devuelto no se puede leer dentro de la función, solo se puede establecer y devolver como parámetro de salida.*

```
CREATE OR REPLACE FUNCTION cuadrado_numero(num INT, OUT resultado INT) AS
$$
BEGIN
  resultado := num * num;
END;
$$
LANGUAGE plpgsql;
```

Y para ejecutar la función, tendríamos que poner solo el valor IN:

```
SELECT cuadrado_numero(5);
```

- **INOUT:** Es el parámetro de entrada/salida, puede ser empleado indistintamente para que forme parte de la lista de parámetros de entrada y que sea parte luego del resultado. Ej: *Supongamos que queremos crear una función que intercambie los valores de dos números enteros. En este caso, los valores de los dos números se pasan a la función como parámetros de entrada y se intercambian dentro de la función, para luego ser devueltos como parámetros de salida.*

```
CREATE OR REPLACE FUNCTION intercambiar_numeros(INOUT num1 INT, INOUT num2 INT) AS
$$
DECLARE
  temp INT;
BEGIN
  temp := num1;
  num1 := num2;
  num2 := temp;
END;
$$
LANGUAGE plpgsql;
```

Para invocar/ejecutar a la función, podemos hacerlo así ya que al ser parámetros INOUT se comportan como parámetros IN:

```
SELECT intercambiar_numeros(5, 10);
```

- **VARIADIC:** Es el parámetro de entrada con un tratamiento especial, que permite definir un arreglo para especificar que la función acepta un conjunto variable de parámetros, los que lógicamente deben ser del mismo tipo. Este tipo no se ha visto en clase con profundiad, por lo que no habrá un ejemplo.

%TYPE

%TYPE se utiliza para definir una variable o parámetro de función con el mismo tipo de datos que otra variable o columna existente.

Por ejemplo, si se tiene una tabla llamada "clientes" con una columna "nombre" de tipo VARCHAR(50), y se quiere declarar una variable en una función que tenga el mismo tipo de datos que la columna "nombre", se puede utilizar la cláusula %TYPE de la siguiente manera:

```
...DECLARE
nombre_cliente clientes.nombre%TYPE;
```

%ROWTYPE

%ROWTYPE es una cláusula que se utiliza para definir una variable o un registro que tenga el mismo tipo de estructura que una fila de una tabla existente.

Por ejemplo, si se tiene una tabla llamada "clientes" con las columnas "id_cliente", "nombre", "apellido" y "email", y se quiere definir una variable en una función que tenga la misma estructura que una fila de la tabla "clientes", se puede utilizar la cláusula %ROWTYPE de la siguiente manera:

```
…DECLARE
cliente_row clientes%ROWTYPE;
```
