Programación de bases de datos.

Caso práctico

Juan recuerda, de cuando estudió el Ciclo de Desarrollo de Aplicaciones Informáticos, que había muchas tareas que se podían automatizar dentro de la base de datos mediante el uso de un lenguaje de programación, e incluso que se podían programar algunas restricciones a la hora de manipular los datos. **Juan** se lo comenta a **María** y ésta se muestra ilusionada con dicha idea ya que muchas veces repiten el trabajo con la base de datos de juegos on-line que tienen entre manos (consultas, inserciones, etc. que son muy parecidas y que se podrían automatizar).

Ministerio de Educación (Uso educativo nc)

Para ello hablan con **Ada** y ésta les comenta que claro que se puede hacer y que precisamente eso es lo que les toca hacer ahora. **Ada** les dice que para ese propósito existen lenguajes de programación que permite hacer lo que ellos quieren. Por ejemplo, PostgreSQL tiene el llamado PL/pgSQL. Así, **Ada** les facilita un manual para que se lo vayan leyendo y se vayan poniendo manos a la obra con la base de datos de juegos on-line.

Ministerio de Educación (Uso educativo nc)

Ahora que ya dominas el uso de SQL para la manipulación y consulta de datos, es el momento de dar una vuelta de tuerca adicional para mejorar las aplicaciones que utilicen nuestra base de datos. Para ello nos vamos a centrar en la programación de bases de datos, utilizando el lenguaje PL/pgSQL. En esta unidad conoceremos qué es PL/pgSQL, cuál es su sintaxis y veremos cómo podemos sacarle el máximo partido a nuestra base de datos mediante su uso.

Debes conocer

La mayor parte de los ejemplos de esta unidad están basados en el modelo de datos extraído del siguiente caso de estudio:

Caso de estudio.

Ministerio de Educación y Formación Profesional. (Dominio público)

1.- Introducción.

Caso práctico

Juan y **María** se han puesto a repasar el manual de PL/pgSQL que les ha pasado Ada. Aunque no han avanzado mucho con el mismo, ya saben a qué se van a enfrentar y los beneficios que pueden obtener del uso del mismo para su aplicación de juegos on-line. Cuando hacen la primera parada de la mañana para tomarse un café, ambos se ponen a comentar las primeras conclusiones que han sacado después de su primer acercamiento a este lenguaje. Ambos están deseosos de seguir avanzando en su aprendizaje y saben que para ello cuentan con la inestimable ayuda de **Ada**.

ITE (Miguel Martínez Monasterio) (Uso educativo nc)

Estarás pensado que si no tenemos bastante con aprender SQL, sino que ahora tenemos que aprender otro lenguaje más que lo único que va a hacer es complicarnos la vida. Verás que eso no es cierto ya que lo más importante, que es el conocimiento de SQL, ya lo tienes. PL/pgSQL tiene una sintaxis muy sencilla y verás como pronto te acostumbras y luego no podrás vivir sin él.

Pero, **¿qué es realmente PL/SQL?**

PL/pgSQL es un lenguaje procedimental para el sistema de base de datos PostgreSQL.

PL/pgSQL le permite ampliar la funcionalidad del servidor de base de datos PostgreSQL creando objetos de servidor con lógica compleja.

PL/pgSQL fue diseñado para:

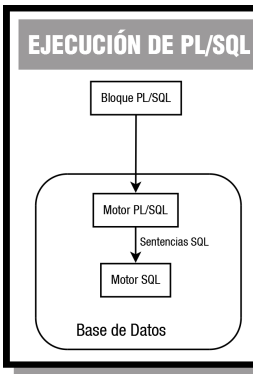
- Crear funciones definidas por el usuario, procedimientos almacenados y disparadores.
- Ampliar el SQL estándar agregando estructuras de control como `if`, `case` y `loop`.
- Heredar todas las funciones, operadores y tipos definidos por el usuario.

Desde PostgreSQL 9.0, PL/pgSQL está instalado de forma predeterminada.

Aunque PL/SQL fue creado por Oracle, hoy día todos los gestores de bases de datos utilizan un lenguaje procedimental muy parecido al ideado por Oracle para poder programar las bases de datos. Por ejemplo, PostgreSQL utiliza PL/pgSQL.

Como veremos, en PL/pgSQL podemos definir variables, constantes, funciones, procedimientos, capturar errores en tiempo de ejecución, anidar cualquier número de bloques, etc. como solemos hacer en cualquier otro lenguaje de programación. Además, por medio de PL/pgSQL programaremos los disparadores de nuestra base de datos, tarea que no podríamos hacer sólo con SQL.

El motor de PL/pgSQL acepta como entrada bloques PL/pgSQL o subprogramas, ejecuta sentencias procedimentales y envía sentencias SQL al servidor de bases de datos. En el esquema adjunto puedes ver su funcionamiento.



Ministerio de Educación (Uso educativo nc)

Para saber más

En el siguiente enlace podrás encontrar una breve historia de PL/SQL.

Breve historia de PL/SQL.

En estos enlaces podrás comprobar como los gestores de bases de datos incluyen hoy día un lenguaje procedimental para programar la base de datos muy parecido a PL/SQL.

Procedimientos almacenados.

Lenguaje procedimental en PostgreSQL.

Fuente

El contenido que aquí se presenta es una adaptación del que puedes encontrar en el siguiente tutorial de PL/pgSQL.

2.- Conceptos básicos.

Caso práctico

ITE (Uso educativo nc)

Juan ha avanzado muy rápido en la lectura del manual que le pasó **Ada**. **Juan** ya sabe programar en otros lenguajes de programación y por tanto la lectura de los primeros capítulos que se centran en cómo se estructura el lenguaje, los tipos de datos, las estructuras de control, etc. le han resultado muy fáciles de comprender. Sabe que lo único que tendrá que tener en cuenta son algunos aspectos en cuanto a las reglas de escritura y demás, pero la lógica es la de cualquier otro lenguaje de programación.

Como **María** está un poco más verde en el tema de la programación, **Juan** se ha ofrecido a darle un repaso rápido a todo lo que él ha visto y a explicarle todo aquello en lo que tenga dudas y a ver si pronto se pueden poner manos a la obra con la base de datos de juegos on-line.

En este apartado nos vamos a ir introduciendo poco a poco en los diferentes conceptos que debemos tener claros para programar en PL/pgSQL. Como para cualquier otro lenguaje de programación, debemos conocer las reglas de sintaxis que podemos utilizar, los diferentes elementos de que consta, los tipos de datos de los que disponemos, las estructuras de control que nos ofrece (tanto iterativas como condicionales) y cómo se realiza el manejo de los errores.

Como podrás comprobar, es todo muy sencillo y pronto estaremos escribiendo fragmentos de código que realizan alguna tarea particular. ¡Vamos a ello!

2.1.- Unidades léxicas (I).

En este apartado nos vamos a centrar en conocer cuáles son las unidades léxicas que podemos utilizar para escribir código en PL/pgSQL. Al igual que en nuestra lengua podemos distinguir diferentes unidades léxicas como palabras, signos de puntuación, etc. En los lenguajes de programación también existen diferentes unidades léxicas que definen los elementos más pequeños que tienen sentido propio y que al combinarlos de manera adecuada, siguiendo las reglas de sintaxis, dan lugar a sentencias válidas sintácticamente.

PL/pgSQL es un lenguaje no sensible a las mayúsculas, por lo que será equivalente escribir en mayúsculas o minúsculas, excepto cuando hablemos de literales de tipo cadena o de tipo carácter.

Cada unidad léxica puede estar separada por espacios (debe estar separada por espacios si se trata de 2 identificadores), por saltos de línea o por tabuladores para aumentar la legibilidad del código escrito.

```
IF A=CLAVE THEN ENCONTRADO:=TRUE;ELSE ENCONTRADO:=FALSE;END IF;
```

Sería equivalente a escribir la siguiente línea:

```
if a=clave then encontrado:=true;else encontrado:=false;end if;
```

Y también sería equivalente a este otro fragmento que es más legible y facilita su comprensión.

```
IF a = clave THEN
     encontrado := TRUE;
ELSE
     encontrado := FALSE;
END IF;
```

Las unidades léxicas se pueden clasificar en:

- Delimitadores.
- Identificadores.
- Literales.
- Comentarios.

Vamos a verlas más detenidamente.

Delimitadores.

PL/pgSQL tiene un conjunto de símbolos denominados delimitadores utilizados para representar operaciones entre tipos de datos, delimitar comentarios, etc. En la siguiente tabla puedes ver un resumen de los mismos.

Delimitadores en PL/pgSQL.

Delimitadores Simples.		Delimitadores Compuestos.	
Símbolo.	**Significado.**	**Símbolo.**	**Significado.**
+	Suma.	**	Exponenciación.

Delimitadores Simples.		Delimitadores Compuestos.	
%	Indicador de atributo.	<>	Distinto.
.	Selector.	¡=	Distinto.
/	División.	<=	Menor o igual.
(	Delimitador de lista.	>=	Mayor o igual.
)	Delimitador de lista.	..	Rango.
:	Variable host.	||	Concatenación.
,	Separador de elementos.	<<	Delimitador de etiquetas.
*	Producto.	>>	Delimitador de etiquetas.
"	Delimitador de identificador acotado.	--	Comentario de una línea.
=	Igual relacional.	/*	Comentario de varias líneas.
<	Menor.	*/	Comentario de varias líneas.
>	Mayor.	:=	Asignación.
@	Indicador de acceso remoto.	=>	Selector de nombre de parámetro.
;	Terminador de sentencias.		
-	Resta/negación.		

2.1.1.- Unidades léxicas (II).

Ya hemos visto qué son los delimitadores. Ahora vamos a continuar viendo el resto de unidades léxicas que nos podemos encontrar en PL/pgSQL.

Identificadores.

Los identificadores en PL/pgSQL, como en cualquier otro lenguaje de programación, son utilizados para nombrar elementos de nuestros programas. A la hora de utilizar los identificadores debemos tener en cuenta los siguientes aspectos:

- Un identificador es una letra seguida opcionalmente de letras, números, $, _, #.
- No podemos utilizar como identificador una palabra reservada.
  - Ejemplos válidos: **`X`**, **`A1`**, **`codigo_postal`**.
  - Ejemplos no válidos: **`rock&roll`**, **`on/off`**.
- PL/pgSQL nos permite además definir los identificadores acotados, en los que podemos usar cualquier carácter con una longitud máxima de 30 y deben estar delimitados por ". Ejemplo: **`"X*Y"`**.
- En PL/pgSQL existen algunos identificadores predefinidos y que tienen un significado especial ya que nos permitirán darle sentido sintáctico a nuestros programas. Estos identificadores son las palabras reservadas y no las podemos utilizar como identificadores en nuestros programas. Ejemplo: **`IF`**, **`THEN`**, **`ELSE`** ...
- Algunas palabras reservadas para PL/pgSQL no lo son para SQL, por lo que podríamos tener una tabla con una columna llamada '**`type`**' por ejemplo, que nos daría un error de compilación al referirnos a ella en PL/pgSQL. La solución sería acotarlos. **`SELECT "TYPE" …`**

Literales.

Los literales se utilizan en las comparaciones de valores o para asignar valores concretos a los identificadores que actúan como variables o constantes. Para expresar estos literales tendremos en cuenta que:

- Los literales numéricos se expresarán por medio de notación decimal o de notación exponencial. Ejemplos: 234, +341, 2e3, -2E-3, 7.45, 8.1e3.
- Los literales tipo carácter y tipo cadena se deben delimitar con unas comillas simples.
- Los literales lógicos son **`TRUE`** y **`FALSE`**.
- El literal **`NULL`** expresa que una variable no tiene ningún valor asignado.

Comentarios.

En los lenguajes de programación es muy conveniente utilizar comentarios en el código. Los comentarios no tienen ningún efecto sobre el código pero sí ayudan mucho al programador o la programadora a recordar qué se está intentando hacer en cada caso (más aún cuando el código es compartido entre varias personas que se dedican a mejorarlo o corregirlo o cuando el mismo autor ha de retomarlo tras mucho tiempo).

En PL/pgSQL podemos utilizar dos tipos de comentarios:

- Los comentarios de una línea se expresaran por medio del delimitador **`--`**. Ejemplo:

```
a:=b;  --asignación
```

- Los comentarios de varias líneas se acotarán por medio de los delimitadores /* y */. Ejemplo:

```
/* Primera línea de comentarios.
    Segunda línea de comentarios. */
```

Ejercicio resuelto

Dada la siguiente línea de código, haz su descomposición en las diferentes unidades léxicas que contenga.

```
IF A <> B
THEN iguales := FALSE; --No son iguales
```

Mostrar retroalimentación

La descomposición en unidades léxicas sería la siguiente:

- Identificadores: **`A`**, **`B`**, **`iguales`**.
- Identificadores (palabras reservadas): **`IF`**, **`THEN`**.
- Delimitadores: **`<>`**, **`:=`**, **`;`**.
- Comentarios: **`--No son iguales`**.

2.2.- Cadenas de caracteres delimitadas por dólar.

Resumen

En esta sección, aprenderemos cómo usar las constantes de cadena de caracteres delimitadas por los símbolos de dólar ($$) en funciones definidas por el usuario y procedimientos almacenados.

Introducción a la sintaxis de constantes de cadenas.

En PostgreSQL, usa comillas simples para una constante de cadena como esta:

```
select 'String constant';
```

Cuando una constante de cadena contiene una comilla simple ('), debe evitarla duplicando la comilla simple. Por ejemplo:

```
select 'I''m also a string constant';
```

El problema surge cuando la constante de cadena contiene muchas comillas simples y barras invertidas. Duplicar cada comilla y barra invertida hace que la constante de cadena sea más difícil de leer y mantener.

La versión 8.0 de PostgreSQL introdujo la función de delimitación por carácter **$**, para hacer que las constantes de cadena fueran más legibles.

A continuación se muestra la sintaxis de las constantes de cadena delimitadas por dólares:

```
$tag$<string_constant>$tag$
```

En esta sintaxis, el **tag** es opcional. Puede contener cero o muchos caracteres.

Entre **tag**, puede colocar cualquier cadena con comillas simples (') y barras invertidas (\). Por ejemplo:

```
select $$I'm a string constant that contains a backslash \$$;
```

En este ejemplo, no especificamos el **tag** entre los dos signos de dólar (**$**).

El siguiente ejemplo utiliza la sintaxis de constantes de cadena delimitadas por dólares con una etiqueta:

```
SELECT $message$I'm a string constant that contains a backslash \$message$;
```

En este ejemplo, usamos la cadena **`message`** como etiqueta entre los dos signos de dólar (**`$`**)

Usando constantes de cadena en bloques anónimos.

A continuación se muestra un bloque anónimo en PL/pgSQL:

```
do
'declare
   agentes_count integer;
begin
   select count(*) into agentes_count
   from agentes;
   raise notice ''El número de agentes es: %'', agentes_count;
end;';
```

```
NOTICE:  El número de agentes es: 39
```

Tenga en cuenta que aprenderá sobre el bloque anónimo en el tutorial de estructura de bloque PL/pgSQL . En este tutorial, puede copiar y pegar el código en cualquier herramienta cliente de PostgreSQL como pgAdmin o psql para ejecutarlo.

El código del bloque debe estar entre comillas simples. Si tiene una comilla simple, debe escaparla, duplicándola de esta manera:

```
raise notice ''El número de agentes es: %'', agentes_count;
```

Para evitar escapar de todas las comillas y barras invertidas, puede usar la cadena delimitada con caracteres dólar de la siguiente manera:

```
do
$$declare
   agentes_count integer;
begin
   select count(*) into agentes_count
   from agentes;
   raise notice 'El número de agentes es: %', agentes_count;
end;$$;
```

```
NOTICE:  El número de agentes es: 39
```

En este ejemplo, no es necesario escapar de las comillas simples y las barras invertidas.

Bloques anónimos

Tenga en cuenta que aprenderá sobre el bloque anónimo en el tutorial de estructura de bloque PL/pgSQL . En este tutorial, puede copiar y pegar el código en cualquier herramienta cliente de PostgreSQL como DBeaver o psql para ejecutarlo.

2.3.- Estructura de bloque PL/pgSQL

Resumen

En esta sección, aprenderemos sobre la estructura de bloques de PL/pgSQL y cómo escribir y ejecutar su primer bloque PL/pgSQL.

Organización en bloques de PL/pgSQL

PL/pgSQL es un lenguaje estructurado en bloques, por lo tanto, una función PL/pgSQL o procedimiento almacenado se organiza en bloques.

Lo siguiente ilustra la sintaxis de un bloque completo en PL/pgSQL:

```
[ <<label>> ]
[ declare
    declarations ]
begin
    statements;
    ...
end [ label ];
```

Examinemos la estructura del bloque con más detalle:

- Cada bloque tiene dos secciones: *declaración* y *cuerpo*. La sección de *declaración* es opcional, mientras que la sección de *cuerpo* es obligatoria. Un bloque termina con un punto y coma (**;**) después de la palabra clave **END**.
- Un bloque puede tener una etiqueta opcional ubicada al principio y al final. Utilice la etiqueta de bloque cuando desee especificarla en la declaración **EXIT** del cuerpo del bloque o cuando desee calificar los nombres de las variables declaradas en el bloque.
- La sección de *declaración* es donde declaras todas las variables utilizadas dentro de la sección del cuerpo. Cada declaración en la sección de declaración termina con un punto y coma (**;**).
- La sección del *cuerpo* es donde colocas el código. Cada declaración en la sección del cuerpo también termina con un punto y coma (**;**).

Ejemplo de estructura de bloque PL/pgSQL.

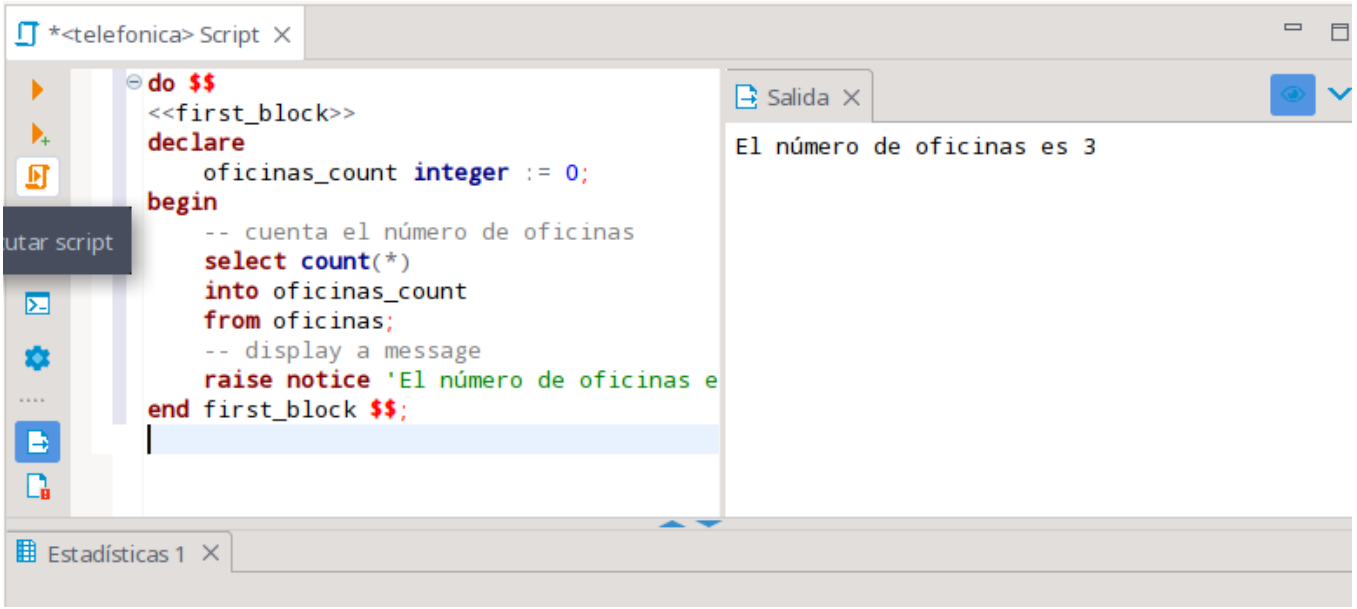
El siguiente ejemplo ilustra un bloque muy simple. Se llama bloque anónimo.

```
do $$
```

```
<<first_block>>
declare
    oficinas_count integer := 0;
begin
    -- cuenta el número de oficinas
    select count(*)
    into oficinas_count
    from oficinas;
    -- display a message
    raise notice 'El número de oficinas es %', oficinas_count;
end first_block $$;
```

```
NOTICE:  The current value of counter is 3
```

Podemos ejecutar el bloque directamente desde **`psql`**. También lo podemos ejecutar desde DBeaver:



Observe que la instrucción **`DO`** no pertenece al bloque. Se utiliza para ejecutar un bloque anónimo. PostgreSQL introdujo la declaración **`DO`** desde la versión 9.0.

También debemos fijarnos en que el bloque anónimo lo delimitamos por carácteres **`$$`**, para evitar tener que encerrarlo entre comillas simples y permitir una mayor legibilidad.

En la sección de *declaración*, declaramos una variable **`oficinas_count`** y establecimos su valor en cero.

```
    oficinas_count integer := 0;
```

Dentro de la sección del *cuerpo*, usamos una declaración **`select into`** con la función **`count()`** para obtener el número de oficinas de la tabla **`oficinas`** y asignar el resultado a la variable **`oficinas_count`**.

```
    select count(*)
    into oficinas_count
    from oficinas;
```

Después de eso, mostramos un mensaje usando una declaración **`raise notice`**:

```
        raise notice 'El número de oficinas es %', oficinas_count;
```

El **`%`** es un marcador de posición que se reemplaza por el contenido de la variable **`oficinas_count`**.

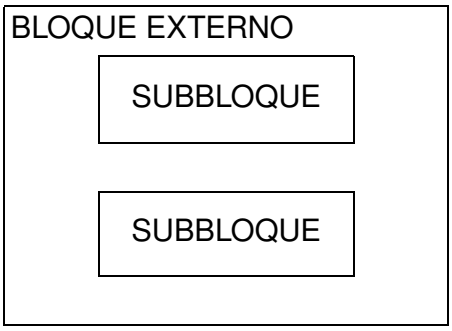
Tenga en cuenta que la etiqueta **`first_block`** es solo para fines de demostración. No hace nada en este ejemplo.

Subbloques PL/pgSQL

PL/pgSQL le permite colocar un bloque dentro del cuerpo de otro bloque.

El bloque anidado dentro de otro bloque se denomina *subbloque*. El bloque que contiene el *subbloque* se denomina bloque externo.

La siguiente imagen ilustra el bloque exterior y el subbloque:



Por lo general, divide un bloque grande en subbloques más pequeños y lógicos. Las variables del subbloque pueden tener los mismos nombres que las del bloque exterior, aunque no es una buena práctica.

3.- Variables y constantes

En esta sección, aprenderá:

- técnicas para declarar variables PL/pgSQL.
- cómo usar la sentencia **`SELECT INTO`** para asignar valores de la base de datos a una variable.
- a usar los **`ROWTYPE`** de PL/pgSQL para declarar variables de fila que contienen una fila completa de un conjunto de resultados.
- los tipos **`RECORD`** de PL/pgSQL, que le permiten definir variables que pueden contener una sola fila de un conjunto de resultados.
- el uso de las constantes PL/pgSQL cuyos valores no se pueden cambiar.

3.1.- Variables

Introducción a las variables en PL/pgSQL.

Una *variable* es un nombre significativo de una ubicación de memoria. Una *variable* tiene un valor que se puede cambiar a través del bloque . Una *variable* siempre está asociada con un *tipo de datos* en particular.

Antes de usar una *variable*, debe declararla en la sección de *declaración* del bloque PL/pgSQL .

A continuación se ilustra la sintaxis de declaración de una variable.

```
variable_name data_type [:= expression];
```

En esta sintaxis:

- Primero, especifique el *nombre de la variable*. Es una buena práctica asignar un nombre significativo a una variable. Por ejemplo, en lugar de nombrar una variable **`i`**, debe usar **`index`** o **`counter`**.
- En segundo lugar, asocie un *tipo de datos* específico con la variable. El tipo de datos puede ser cualquier tipo de datos válido, como **`integer`**, **`numeric`** , **`varchar`** y **`char`**.
- Tercero, asigne opcionalmente un valor predeterminado a una variable. Si no lo hace, el valor inicial de la variable es **`NULL`**.

El siguiente ejemplo ilustra cómo declarar e inicializar variables:

```
do $$
declare
    counter integer := 1;
    first_name varchar(50) := 'John';
    last_name varchar(50) := 'Doe';
    payment numeric(11,2) := 20.5;
begin
    raise notice '% % % has been paid % USD',
        counter,
        first_name,
        last_name,
        payment;
end $$;
```

La variable **`counter`** es un **`integer`** que se inicializa a 1

Las variables **`first_name`** y **`last_name`** son **`varchar(50)`** y se inicializan en las constantes de cadena 'John' y 'Doe'.

El tipo de **`payment`** es **`numeric`** y su valor se inicializa a 20.5

Tenga en cuenta que puede usar el operador de asignación `:=` o `=` para inicializar y asignar un valor a una variable.

Momento de inicialización de una variable.

PostgreSQL evalúa el valor predeterminado de una variable y lo asigna a la variable cuando se ingresa al bloque. Por ejemplo:

```
do $$
declare
    created_at time := now();
begin
    raise notice '%', created_at;
    perform pg_sleep(10);
    raise notice '%', created_at;
end $$;
```

Aquí está la salida correspondiente:

```
NOTICE:  19:33:03.988743
    (10 segundos más tarde)
NOTICE:  19:33:03.988743
```

En este ejemplo:

- Primero, se declara una variable cuyo valor predeterminado se inicialice a la hora actual.
- Segundo, imprime el valor de la variable y pausa la ejecución en 10 segundos usando la función **`pg_sleep()`**.
- Tercero, imprime el valor de la variable **`created_at`** nuevamente.

Como se muestra en la salida, el valor de **`created_at`** solo se inicializa una vez cuando se ingresa el bloque.

Copiar tipos de datos.

`%type` proporciona el tipo de datos de una columna de tabla u otra variable. Por lo general, se usa **`%type`** para declarar una variable que contiene un valor de la base de datos u otra variable.

A continuación se ilustra cómo declarar una variable con el tipo de datos de una columna de tabla:

```
variable_name table_name.column_name%type;
```

Y lo siguiente muestra cómo declarar una variable con el tipo de datos de otra variable:

```
variable_name variable%type;
```

Consulte la siguiente tabla de oficinas de la base de datos del caso de estudio:

```
telefonica=> \d oficinas

                          Tabla «public.oficinas»
    Columna    |         Tipo          | Ordenamiento | Nulable  | Por omisión
---------------+-----------------------+--------------+----------+-------------
 identificador | numeric(6,0)          |              | not null |
 nombre        | character varying(40) |              | not null |
 domicilio     | character varying(40) |              |          |
 localidad     | character varying(20) |              |          |
 codigo_postal | character varying(5)  |              |          |
```

Este ejemplo utiliza la técnica de copia de tipos para declarar variables que contienen valores que provienen de la tabla oficinas:

```
do $$
declare
    oficina_nombre oficinas.nombre%type;
    oficina_domicilio oficinas.nombre%type;
begin
    -- obtener nombre de la oficina con id 2
    select nombre
    from oficinas
    into oficina_nombre
    where identificador = 2;
    -- muestra el nombre de la oficina
    raise notice 'Nombre de la oficina con id 2: %', oficina_nombre;
end; $$;
```

Este ejemplo declaró dos variables:

- La variable **`oficina_nombre`** tiene el mismo tipo de datos que la columna **`nombre`** de la tabla **`oficinas`** de la base de datos de ejemplo .
- La variable **`oficina_domicilio`** tiene el mismo tipo de datos que el tipo de datos de la variable **`oficina_nombre`**.

Al utilizar la función de copia de tipos, obtiene las siguientes ventajas:

- Primero, no necesita saber el tipo de columna o referencia a la que hace referencia.
- En segundo lugar, si cambia el tipo de datos del nombre de la columna (o variable) a la que se hace referencia, no necesita cambiar la definición de la función.

Variables en bloque y subbloque.

Cuando declara una variable en un subbloque que tiene el mismo nombre que otra variable en el bloque exterior, la variable en el bloque exterior se oculta en el subbloque.

En caso de que desee acceder a una variable en el bloque exterior, use la etiqueta del bloque para calificar su nombre como se muestra en el siguiente ejemplo:

```
do $$
<<outer_block>>
declare
  counter integer := 0;
begin
   counter := counter + 1;
   raise notice 'The current value of the counter is %', counter;

   declare
       counter integer := 0;
   begin
       counter := counter + 10;
       raise notice 'Counter in the subblock is %', counter;
       raise notice 'Counter in the outer block is %', outer_block.counter;
   end;

   raise notice 'Counter in the outer block is %', counter;

end outer_block $$;
```

```
NOTICE: The current value of the counter is 1
NOTICE: Counter in the subblock is 10
NOTICE: Counter in the outer block is 1
NOTICE: Counter in the outer block is 1
```

Conversión de tipos.

Aunque en PL/pgSQL existe la conversión implícita de tipos para tipos parecidos, siempre es aconsejable utilizar la conversión explícita de tipos por medio de funciones de conversión (**`TO_CHAR`**, **`TO_DATE`**, **`TO_NUMBER`**, …) y así evitar resultados inesperados.

3.2.- SELECT INTO

Introducción a la sentencia SELECT INTO de PL/pgSQL

La sentencia **SELECT INTO** nos permite seleccionar datos de la base de datos y asignarlos a una variable.

A continuación se ilustra la sintaxis de la sentencia **SELECT INTO**:

```
select select_list
into variable_name
from table_expression;
```

En esta sintaxis, se coloca la variable después de la palabra clave **INTO**. La sentencia **SELECT INTO** asignará los datos devueltos por la cláusula **SELECT** a la variable.

La sentencia **SELECT INTO** admite cualquier cláusula, incluyendo **JOIN**, **GROUP BY** y **HAVING**.

Ejemplo de sentencia SELECT INTO de PL/pgSQL

Vea el siguiente ejemplo:

```
do $$
declare
    agentes_count integer;
begin
    -- select el número de agentes de la tabla agentes
    select count(*)
    into agentes_count
    from agentes;

    -- muestra el número de agentes
    raise notice 'El número de agentes es: %', agentes_count;
end;
$$;
```

```
NOTICE: El número de agentes es: 39
```

En este ejemplo:

- Primero, declare una variable llamada **`agentes_count`** que almacenará el número de agentes de la tabla **`agentes`**.
- En segundo lugar, use la sentencia **`SELECT INTO`** para asignar el número de agentes a la variable **`agentes_count`**.
- Finalmente, muestre un mensaje con el valor de la variable **`agentes_count`** usando la sentencia **`raise notice`**.

3.3.- ROWTYPE

Introducción al ROWTYPE de PL/pgSQL

Para almacenar una fila completa devuelta por una sentencia **`select into`**, utilice una **`ROWTYPE`** variable o variable de fila.

Puede declarar una variable que tenga el mismo tipo de datos que el tipo de datos de la fila en una tabla usando la siguiente sintaxis:

```
row_variable table_name%ROWTYPE;
row_variable view_name%ROWTYPE;
```

Para acceder ana columna individual de la variable de fila, utilice la notación de punto (**.**) de esta manera:

```
row_variable.field_name
```

Ejemplo de tipos de fila PL/pgSQL

Usaremos la tabla **`agentes`** de la base de datos para mostrar cómo funcionan los tipos de fila:

```
telefonica=> \d agentes
                           Tabla «public.agentes»
    Columna    |         Tipo          | Ordenamiento | Nulable  | Por omisión
---------------+-----------------------+--------------+----------+-------------
 identificador | numeric(6,0)          |              | not null |
 nombre        | character varying(60) |              | not null |
 usuario       | character varying(20) |              | not null |
 clave         | character varying(20) |              | not null |
 habilidad     | numeric(1,0)          |              | not null |
 categoria     | numeric(1,0)          |              | not null |
 familia       | numeric(6,0)          |              |          |
 oficina       | numeric(6,0)          |              |          |
```

El siguiente ejemplo muestra el nombre y la categoría del agente con identificador 11:

```
do $$
declare
    agente_seleccionado agentes%rowtype;
begin
-- selecciona al agente con identificador 11
    select *
    from agentes
    into agente_seleccionado
    where identificador = 11;

-- muestra la información del agente
```

```
    raise notice 'El nombre del agente es %',
    agente_seleccionado.nombre;

    raise notice 'La categoría del agente es %',
    agente_seleccionado.categoria;
end; $$;
```

Cómo funciona.

- Primero, declare una variable de fila **`agente_seleccionado`**, cuyo tipo de datos sea el mismo que la fila de la tabla **`agentes`**.
- Segundo, asigne la fila cuyo valor en la columna **`identificador`** es 11 a la variable **`agente_seleccionado`** usando la sentencia **`SELECT INTO`**.
- Tercero, muestre el nombre y apellido del actor seleccionado usando la raise noticedeclaración. Accedió a los campos first_namey last_nameusando la notación de punto.

3.4.- Tipo RECORD

Introducción a los tipos RECORD de PL/pgSQL

PostgreSQL proporciona un "tipo" llamado **RECORD** que es similar al **ROWTYPE**.

Para declarar una variable **RECORD**, usa un nombre de variable seguido de la palabra clave **RECORD** como esta:

```
nombre_variable record;
```

Una variable **RECORD** solo puede contener una fila de un conjunto de resultados.

A diferencia de una variable **ROWTYPE**, una variable **RECORD** no tiene una estructura predefinida. La estructura de una variable **RECORD** se determina cuando la instrucción **select** o **for** le asigna una fila real.

Para acceder a un campo en el registro, utiliza la sintaxis de notación de punto (**.**) como esta:

```
nombre_variable.nombre_columna;
```

Si intenta acceder a un campo en una variable **RECORD** antes de que se asigne, obtendrá un error.

De hecho, un registro no es un verdadero tipo de datos. Es solo un marcador de posición. Además, una variable de registro puede cambiar su estructura cuando la reasigna.

Ejemplos de RECORD PL/pgSQL

Tomemos algunos ejemplos del uso de las variables **RECORD**.

1. Uso de registro con la instrucción **select into**

   El siguiente ejemplo ilustra cómo usar la variable de registro con la sentencia **select into**:

```
do $$
declare
    rec record;
begin
    -- seleccionar el agente
    select nombre, categoria, oficina
    into rec
    from agentes
    where identificador = 11;
```

```
    raise notice '% % %', rec.nombre, rec.categoria, rec.oficina;

end;$$;
```

```
NOTICE:  Narciso Jáimez Toro 2 1
```

Cómo funciona:

- Primero, declare una variable **RECORD** llamada **rec** en la sección de declaración.
- En segundo lugar, use la sentencia **select into** para seleccionar una fila cuyo valor **identificador** sea 11 en la variable **rec**.
- Tercero, imprima la información del agente a través de la variable **RECORD**.

2. Usar variables **RECORD** en la declaración de bucle **for**

A continuación se muestra cómo utilizar una variable **RECORD** en una instrucción **for loop**:

```
do $$
declare
    rec record;
begin
    -- seleccionar los agentes de la categoria 1
    for rec in select nombre, categoria, habilidad
        from agentes
        where categoria = 1
    loop
        raise notice '% % %', rec.nombre, rec.categoria, rec.habilidad;
    end loop;
end;$$;
```

```
NOTICE:  Diosdado Sánchez Hernández 1 8
NOTICE:  Jesús Baños Sancho 1 8
NOTICE:  Salvador Romero Villegas 1 7
NOTICE:  José Javier Bermúdez Hernández 1 7
NOTICE:  Alfonso Bonillo Sierra 1 7
NOTICE:  Silvia Thomas Barrós 1 7
```

Tenga en cuenta que aprenderá más sobre la instrucción for loop en la sección correspondiente.

Cómo funciona:

- Primero, declare una variable llamada **rec** con el tipo **record**.
- En segundo lugar, use la instrucción **for loop** para obtener filas de la tabla **agentes** (en la base de datos de ejemplo) cuya categoría sea la 1. La instrucción **for loop** asigna la fila que consiste en **nombre**, **categoria** y **habilidad** a la variable **rec** en cada iteración.
- Tercero, muestre el contenido de las columnas de la variable **RECORD** usando la notación de punto (**rec.nombre, rec.categoria, rec.habilidad**)

3.5.- Constantes PL/pgSQL

Introducción a las constantes PL/pgSQL

A diferencia de una variable, el valor de una constante no se puede cambiar una vez que se inicializa.

Las siguientes son las razones para usar constantes.

- Primero, las constantes hacen que el código sea más legible y mantenible, por ejemplo, imagine que tiene la siguiente fórmula:

```
precio_venta := precio_neto + precio_neto * 0.1;
```

  ¿Qué significa 0.1? Se puede interpretar como cualquier cosa.

  Pero cuando usas la siguiente fórmula, todos conocen el significado del cálculo del precio de venta que es igual al precio neto más el impuesto al valor agregado (IVA).

```
precio_venta := precio_neto + precio_neto * iva;
```

- En segundo lugar, las constantes reducen el esfuerzo de mantenimiento.

  Suponga que tiene una fórmula que calcula el precio de venta en todos los lugares de una función. Cuando el IVA cambia, por ejemplo, de 0,1 a 0,5, debe cambiar todos estos valores codificados.

  Al usar una constante, solo necesita cambiar su valor en un lugar donde define la constante.

Definición de constantes

Para definir una constante en PL/pgSQL, utilice la siguiente sintaxis:

```
constant_name constant data_type := expression;
```

En esta sintaxis:

- Primero, especifique el nombre de la constante. El nombre debe ser lo más descriptivo posible.
- En segundo lugar, agregue la palabra clave **`constant`** después del nombre y especifique el tipo de datos de la constante.
- Tercero, inicialice un valor para la constante después del operador de asignación (**`:=`**).

Ejemplo de constantes PL/pgSQL

El siguiente ejemplo declara un nombre constante **`iva`** que almacena el impuesto al valor añadido y calcula el precio de venta a partir del precio neto:

```
do $$
declare
    iva constant numeric := 0.1;
    precio_neto numeric := 20.5;
begin
    raise notice 'El precio de venta es %', precio_neto * ( 1 + iva );
end $$;
```

```
El precio de venta es 22.55
```

Ahora, si intenta cambiar el valor de la constante de la siguiente manera:

```
do $$
declare
    iva constant numeric := 0.1;
    precio_neto numeric := 20.5;
begin
    raise notice 'El precio de venta es %', precio_neto * ( 1 + iva );
    iva := 0.05;
end $$;
```

Recibirá el siguiente mensaje de error:

```
ERROR:  la variable «iva» esta declarada como CONSTANT
```

Similar al valor predeterminado de una variable , PostgreSQL evalúa el valor de la constante cuando se ingresa el bloque en tiempo de ejecución, no en tiempo de compilación. Por ejemplo:

```
do $$
declare
    inicio constant time := now();
begin
    raise notice 'El bloque se ha empezado a ejecutar a las %', inicio;
end $$;
```

```
NOTICE:  El bloque se ha empezado a ejecutar a las 12:49:53.161632
```

PostgreSQL evalúa la función **`now()`** cada vez que se llama al bloque. Para ver su efecto, puedes ejecutar el bloque repetidamente:

```
NOTICE:  El bloque se ha empezado a ejecutar a las 12:49:57.160391
```

4.- Informes de mensajes y errores

En esta sección, aprenderá

- a enviar mensajes informativos y generar mensajes de error utilizando la sentencia **`raise`**.
- a usar la instrucción **`assert`** para insertar comprobaciones de depuración en bloques PL/pgSQL.

4.1.- Mensajes y errores de PL/pgSQL

Informes de mensajes

Para generar un mensaje, utilice la sentencia **raise** de la siguiente manera:

```
raise nivel formato;
```

Examinemos la sentencia **raise** con más detalle.

Nivel

Después de la sentencia **raise** está la opción **nivel** que especifica la gravedad del error.

PostgreSQL proporciona los siguientes niveles:

- **debug**
- **log**
- **notice**
- **info**
- **warning**
- **exception**

Si no especifica el **nivel**, de forma predeterminada, la sentencia **raise** utilizará **exception** como nivel que genera el error y detendrá la transacción actual. Hablaremos de **raise exception** más adelante.

Formato

Es una cadena que especifica el mensaje formateado. Los símbolos de porcentaje (**%**) se usan como marcadores de posición, que serán sustituidos por los argumentos establecidos en la instrucción **raise**.

La cantidad de marcadores de posición debe ser la misma que la cantidad de argumentos; de lo contrario, PostgreSQL generará un error:

```
ERROR:  se especificaron demasiados parámetros a RAISE
```

El siguiente ejemplo ilustra la sentencia **raise** que informa diferentes mensajes en el momento actual.

```
do $$
begin
    raise info 'mensaje informativo %', now() ;
    raise log 'mensaje de log %', now();
    raise debug 'mensaje de depuración %', now();
    raise warning 'mensaje de advertencia %', now();
    raise notice 'mensaje de aviso %', now();
```

```
end $$;
```

```
INFO:  mensaje informativo 2022-12-07 13:45:23.082403+01
WARNING:  mensaje de advertencia 2022-12-07 13:45:23.082403+01
NOTICE:  mensaje de aviso 2022-12-07 13:45:23.082403+01
```

Tenga en cuenta que no todos los mensajes se envían al cliente. PostgreSQL solo informa al cliente de los mensajes de nivel **`info`**, **`warning`** y **`notice`**. Esto es controlado por los parámetros de configuración **`client_min_messages`** y **`log_min_messages`**.

Lanzando errores

Para generar un error, use el nivel **`exception`** después de la sentencia **`raise`**. Tenga en cuenta que la sentencia **`raise`** utiliza el nivel **`exception`** de forma predeterminada.

Además de generar un error, puede agregar más información utilizando la siguiente cláusula adicional:

```
using option = expression
```

`option` puede ser:

- **`message`**: establecer mensaje de error.
- **`hint`**: proporcione el mensaje de sugerencia para que la causa raíz del error sea más fácil de descubrir.
- **`detail`**: proporciona información detallada sobre el error.
- **`errcode`**: identifique el código de error, que puede ser el nombre de la condición o directamente un código **`SQLSTATE`** de cinco caracteres. Consulte la tabla de códigos de error y nombres de condiciones .

`expression` es una expresión de cadena.

El siguiente ejemplo genera un mensaje de error de correo electrónico duplicado:

```
do $$
declare
    email varchar(255) := 'info@postgresqltutorial.com';
begin
    -- check email duplicado
    -- ...
    -- informar del email duplicado
    raise exception 'email duplicado: %', email
    using hint = 'revise el email';
end $$;
```

```
ERROR:  email duplicado: info@postgresqltutorial.com
SUGERENCIA:  revise el email
CONTEXTO:  función PL/pgSQL inline_code_block en la línea 8 en RAISE
```

Los siguientes ejemplos ilustran cómo generar un SQLSTATEy su condición correspondiente:

```
Edo $$
begin
    --...
    raise sqlstate '2201B';
end $$;
```

```
ERROR:  2201B
CONTEXTO:  función PL/pgSQL inline_code_block en la línea 4 en RAISE
```

```
do $$
begin
    --...
    raise invalid_regular_expression;
end $$;
```

```
ERROR:  invalid_regular_expression
CONTEXTO:  función PL/pgSQL inline_code_block en la línea 4 en RAISE
```

4.2.- Sentencia ASSERT

Introducción a la sentencia ASSERT.

La sentencia **assert** es una abreviatura útil para insertar comprobaciones de depuración en el código PL/pgSQL.

A continuación se ilustra la sintaxis de la sentencia **assert** :

```
assert condición [, mensaje];
```

En esta sintaxis:

1. condición
   condición es una expresión booleana que se espera que siempre devuelva **true**.

   Si **condición** se evalúa como true, la sentencia **assert** no hace nada.

   En caso de **condición** se evalúe como **false** o **null**, PostgreSQL genera una excepción **assert_failure**.

2. mensaje
   El **mensaje** es opcional.

   Si no se facilita un **mensaje**, PostgreSQL usa el mensaje **assertion failed " "** por defecto. En caso de que se facilita un **mensaje** a la sentencia **assert**, se usará en lugar del mensaje predeterminado.

Tenga en cuenta que debe usar la sentencia **assert** únicamente para detectar errores, no para informar. Para informar un mensaje o un error, utilice la sentencia **raise** en su lugar.

Habilitar/Deshabilitar aserciones.

PostgreSQL proporciona el parámetro de configuración **plpgsql.check_asserts** para *habilitar* o *deshabilitar* las pruebas de aserción. Si establece este parámetro en **off**, las sentencias **assert** no harán nada.

Ejemplo de assert de PostgreSQL

El siguiente ejemplo usa la sentencia **assert** para verificar si la tabla **oficinas** de la base de datos de muestra tiene datos:

```
do $$
declare
    oficinas_count integer;
begin
    select count(*)
    into oficinas_count
    from oficinas;

    assert oficinas_count > 0, 'No se han encontrado oficinas. Compruebe la tabla oficinas';
end$$;
```

Debido a que la tabla de **oficinas** tiene datos, el bloque no emitió ningún mensaje.

El siguiente ejemplo genera un error porque el número de oficinas de la tabla **oficinas** no es mayor que 20.

```
do $$
declare
    oficinas_count integer;
begin
    select count(*)
    into oficinas_count
    from oficinas;

    assert oficinas_count > 20, 'No se hay suficientes oficinas. Compruebe la tabla oficinas
end$$;
```

```
ERROR:  No se hay suficientes oficinas. Compruebe la tabla oficinas
CONTEXTO:  función PL/pgSQL inline_code_block en la línea 9 en ASSERT
```

4.3.- Manejo de errores.

ITE (Félix Vallés Calvo) (Uso educativo nc)

Muchas veces te habrá pasado que surgen situaciones inesperadas con las que no contabas y a las que tienes que hacer frente. Pues cuando programamos con PL/SQL pasa lo mismo, que a veces tenemos que manejar errores debidos a situaciones diversas. Vamos a ver cómo tratarlos.

Cualquier situación de error es llamada **excepción** en PL/SQL. Cuando se detecta un error, una excepción es lanzada, es decir, la ejecución normal se para y el control se transfiere a la parte de manejo de excepciones. La parte de manejo de excepciones es la parte etiquetada como **`EXCEPTION`** y constará de sentencias para el manejo de dichas excepciones, llamadas **manejadores de excepciones**.

Manejadores de excepciones

Sintaxis:

```
WHEN nombre_excepcion THEN
     <sentencias para su manejo>
     ....
WHEN OTHERS THEN
     <sentencias para su manejo>
```

Ejemplo:

```
DECLARE
     supervisor agentes%ROWTYPE;
BEGIN
     SELECT * INTO supervisor FROM agentes
     WHERE categoria = 2 AND oficina = 3;
          ...
EXCEPTION
     WHEN NO_DATA_FOUND THEN
     --Manejamos el no haber encontrado datos
WHEN OTHERS THEN
     --Manejamos cualquier error inesperado
END;
/
```

La parte **`OTHERS`** captura cualquier excepción no considerada anteriormente.

Debes conocer

En el siguiente enlace podrás ver las diferentes excepciones predefinidas en PostgreSQL, junto a su código de error asociado (que luego veremos lo que es) y una explicación de cuándo son lanzadas.

Excepciones predefinidas en PostgreSQL.

5.- Estructuras de control.

En la vida constantemente tenemos que tomar decisiones que hacen que llevemos a cabo unas acciones u otras dependiendo de unas circunstancias o repetir una serie de acciones un número dado de veces o hasta que se cumpla una condición. En PL/SQL también podemos imitar estas situaciones por medio de las estructuras de control que son sentencias que nos permiten manejar el flujo de control de nuestro programa.

Las sentencias de control de flujo las dividimos en:

- condicionales
  - **IF**
  - **CASE**
- iterativas
  - **LOOP**
  - **WHILE**
  - **FOR**

5.1.- Sentencia IF

La sentencia **`if`** determina qué declaraciones ejecutar en función del resultado de una expresión booleana.

PL/pgSQL le proporciona tres formas de sentencia **`if`**.

- **`if then`**
- **`if then else`**
- **`if then elsif`**

Sentencia if-then

A continuación se ilustra la forma más simple de la sentencia **`if`**:

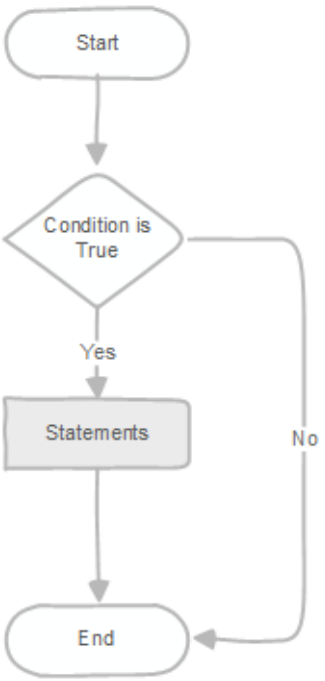
```
if condición then
    sentencias;
end if;
```

La sentencia **`if`** ejecutará las **`sentencias`** si la **`condición`** es verdadera. Si **`condición`** se evalúa como falsa, el control se pasa a la siguiente sentencia después de la **`end if`**.

`condición` es una expresión *booleana* que se evalúa como **`true`** o **`false`**.

Las **`sentencias`** que se ejecutarán si la **`condición`** se evalúa como **`true`** puede ser una única o un bloque de sentencias. Puede ser cualquier sentencia válida, incluso otra sentencia **`if`** anidada.

El siguiente diagrama de flujo ilustra la sentencia **`if`** simple.



Vea el siguiente ejemplo:

```
do $$
```

```
declare
    oficina_seleccionada oficinas%rowtype;
    id_oficina oficinas.identificador%type := 0;
begin

    select * from oficinas
    into oficina_seleccionada
    where identificador = id_oficina;

    if not found then
        raise notice'La oficina % no puede ser encontrada.',
            id_oficina;
    end if;
end $$;
```

En este ejemplo, seleccionamos una oficina mediante un **identificador** específico (0).

La variable global **FOUND** está disponible en el lenguaje de procedimientos PL/pgSQL. Si la sentencia **select into** consigue asignar una fila para la variable **oficina_seleccionada**, entonces **FOUND** será **true**; en caso contrarío, **FOUND** será **false**.

En el ejemplo, usamos la sentencia **if** para verificar si la oficina con **identificador** (0) existe y enviar un aviso si no existe.

```
NOTICE:  La oficina 0 no puede ser encontrada.
```

Si cambia el valor de la variable **id_oficina** a algún valor que existe en la tabla de **oficinas**, como 1, no verá ningún mensaje.

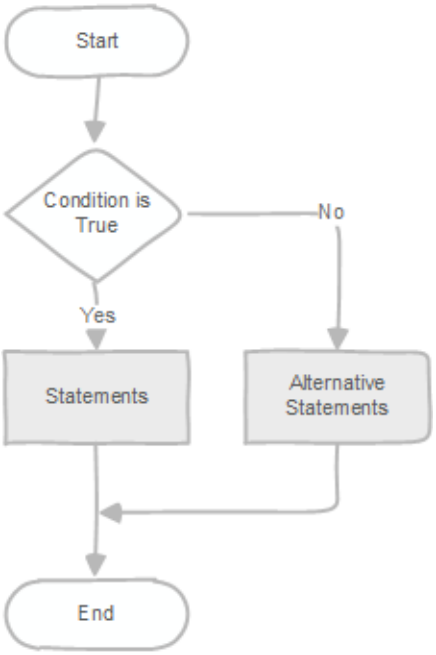
Sentencia if-then-else

A continuación se ilustra la sintaxis de la sentencia **if-then-else**:

```
if condición then
    sentencias;
else
    sentencias_alternativas;
end if;
```

La sentencia **if-then-else** ejecuta las **sentencias** en la rama **if** si **condición** se evalúa como verdadera; de lo contrario, ejecuta las **sentencias_alternativas** de la rama **else**.

El siguiente diagrama de flujo ilustra la sentencia **if-then-else**.



Vea el siguiente ejemplo:

```
do $$
declare
    oficina_seleccionada oficinas%rowtype;
    id_oficina oficinas.identificador%type := 1;
begin

    select * from oficinas
    into oficina_seleccionada
    where identificador = id_oficina;

    if not found then
        raise notice'La oficina % no ùede ser encontrada.',
            id_oficina;
    else
        raise notice'El nombre de la oficina es %.',
            oficina_seleccionada.nombre;
    end if;
end $$;
```

En este ejemplo, la identificación de la oficina 1 existe en la tabla de oficinas, por lo que la variable **`FOUND`** se estableció en verdadero. Por lo tanto, la instrucción en la rama **`else`** se ejecutó.

Aquí está la salida:

```
NOTICE:  El nombre de la oficina es Madrid.
```

Sentencia if-then-elsif

A continuación se ilustra la sintaxis de la sentencia if then elsif:

```
if condición_1 then
    sentencias_1;
elsif condición_2 then
    sentencias_2
    ...
elsif condición_n then
    sentencias_n;
else
    sentencias-else;
end if;
```

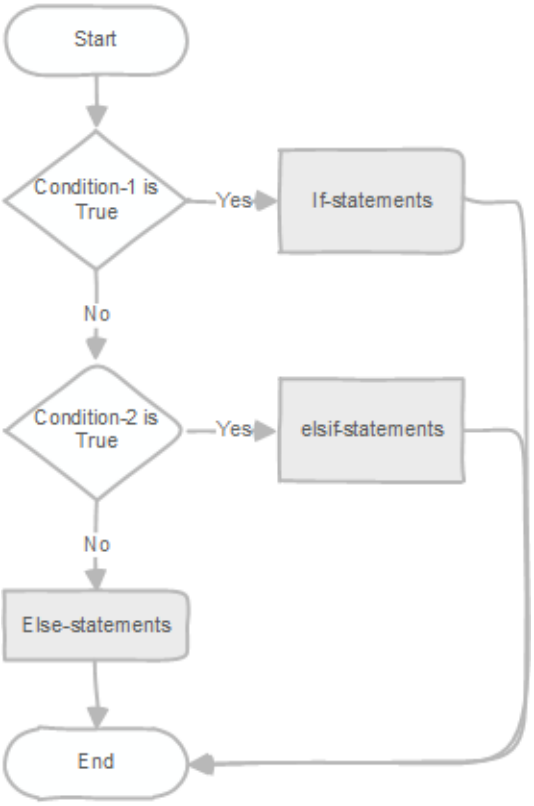
Las sentencias **`if`** y **`if then else`** evalúan una condición. Sin embargo, la sentencia **`if then elsif`** evalúa múltiples condiciones.

Si una **`condición`** se evalúa como cierta, se ejecuta las **`sentencias`** correspondiente en esa rama.

Por ejemplo, si **`condición_1`** es **`true`** entonces se ejecutarán las **`sentencias_1`** y se deja de evaluar el resto de condiciones.

Si todas las **`condiciones`** se evalúan como **`false`**, se ejecutarán las sentencias en la rama **`else`**.

El siguiente diagrama de flujo ilustra la sentencia **`if then elsif`**:



Veamos el siguiente ejemplo:

```
do $$
declare
    oficina_seleccionada oficinas%rowtype;
    id_oficina oficinas.identificador%type := 1;
    provincia varchar(20);
begin

    select * from oficinas
    into oficina_seleccionada
    where identificador = id_oficina;
```

```
    if not found then
        raise notice'La oficina % no puede ser encontrada.',
            id_oficina;
    elsif trunc(oficina_seleccionada.codigo_postal::int / 1000) = 28 then
        provincia := 'Madrid';
    elsif trunc(oficina_seleccionada.codigo_postal::int / 1000) = 30 then
        provincia := 'Murcia';
    elsif trunc(oficina_seleccionada.codigo_postal::int / 1000) = 27 then
        provincia := 'Jaén';
    elsif trunc(oficina_seleccionada.codigo_postal::int / 1000) = 36 then
        provincia := 'Granada';
    else
        provincia := 'Desconocida';
    end if;
    raise notice'La oficina con dirección en % está en la provincia de %.',
        oficina_seleccionada.domicilio, provincia;
end $$;
```

```
NOTICE:  La oficina con dirección en Gran vía, 37 está en la provincia de Madrid.
```

Cómo funciona:

Primero, seleccione la oficina con identificador 1. Si la oficina no existe, envíe un aviso de que no se encuentra la oficina.En segundo lugar, use la sentencia **`if then elsif`** para asignar a la oficina una provincia en función del código postal de la oficina.

5.2.- Sentencia CASE

La sentencia **case** ejecuta las sentencias asociadas a una sección **when** de una lista de secciones **when**, en función de una condición.

La sentencia **case** tiene dos formas:

- sentencia case sencilla
- sentencia case buscada

Sentencia CASE simple

Comencemos con la sintaxis de la sentencia case simple:

```
case expresión-buscada
    when expresión_1 [, expresión_2, ...] then
        sentencias-when
[ ... ]
[else
    sentencias-else
]
END case;
```

La **expresión-buscada** una expresión que se evalúa como un resultado.

La sentencia **case** compara el resultado de **expresión-buscada** con la **expresión** de cada rama **when** usando el operador igual (**=**) de arriba a abajo.

Si la sentencia **case** encuentra una coincidencia, ejecutará la sección **when** correspondiente. Además, deja de comparar el resultado de **expresión-buscada** con el resto de expresiones.

Si la sentencia **case** no puede encontrar ninguna coincidencia, ejecutará la sección **else**.

La sección **else** es opcional. Si el resultado de **expresión-buscada** no coincide con ninguna de las **expresión** en las secciones **when** y la sección **else** no existe, la sentencia **case** generará una excepción **case_not_found**.

El siguiente es un ejemplo de la declaración simple case.

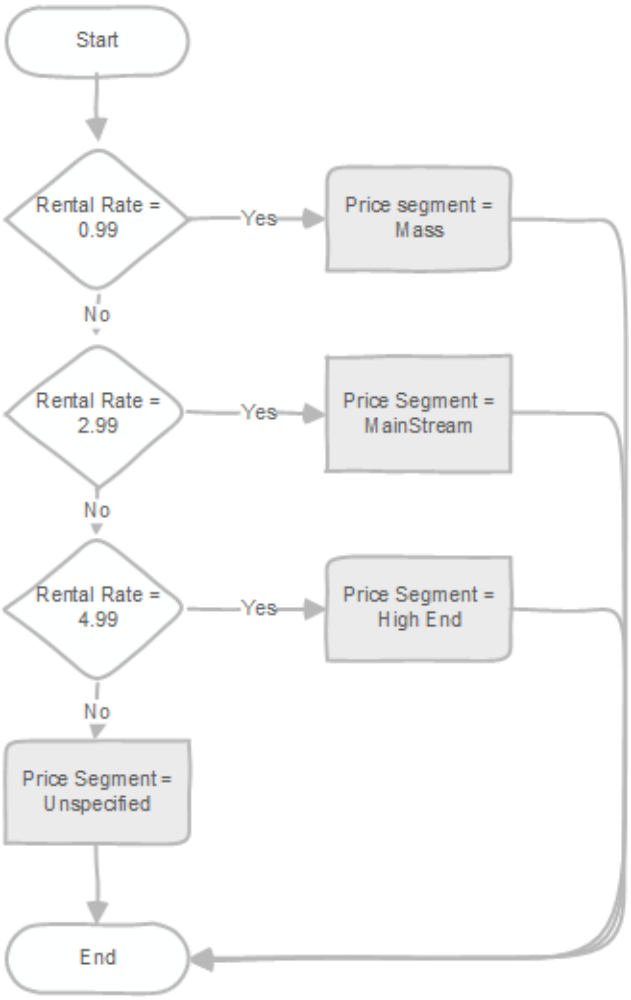
```
do $$
declare
    oficina_seleccionada oficinas%rowtype;
    id_oficina oficinas.identificador%type := 1;
    provincia varchar(20);
begin

    select * from oficinas
    into oficina_seleccionada
    where identificador = id_oficina;

    if not found then
        raise notice'La oficina % no puede ser encontrada.',
            id_oficina;
```

```
    else
        case trunc(oficina_seleccionada.codigo_postal::int / 1000)
            when 28 then
                provincia := 'Madrid';
            when 30 then
                provincia := 'Murcia';
            when 27 then
                provincia := 'Jaén';
            when 36 then
                provincia := 'Granada';
            else
                provincia := 'Desconocida';
        end case;
        raise notice'La oficina con dirección en % está en la provincia de %.',
            oficina_seleccionada.domicilio, provincia;
    end if;
end $$;
```

```
NOTICE:  La oficina con dirección en Gran vía, 37 está en la provincia de Madrid.
```

En este ejemplo, primero selecciona la oficina con **`identificador`** 1. En función de su **`codigo_postal`**, asigna una provincia diferente. En caso de que el **`codigo_postal`** no comience por 28, 30, 27 o 36, la sentencia **`case`** asigna a la **`provincia`** el valor **`'Desconocida'`**.

El siguiente diagrama de flujo ilustra la sentencia **`case`** simple:



Sentencia CASE buscada

La siguiente sintaxis muestra la sintaxis de la sentencia **case** buscada:

```
case
    when expresión_booleana_1 then
        sentencias    [ when expresión_booleana_2 then        sentencias     ... ]
[ ... ]
[else
    sentencias-else
]
END case;
```

En esta sintaxis, la sentencia **case** evalúa las expresiones booleanas secuencialmente de arriba a abajo hasta que encuentra una expresión que se evalúa como **true**.

Una vez que encuentra una expresión que se evalúa como **true**, la instrucción **case** ejecuta la sección **when** correspondiente e inmediatamente deja de buscar las expresiones restantes.

En caso de que ninguna expresión se evalúe como **true**, la sentencia **case** ejecutará la sección **else**.

La sección **else** es opcional. Si omite la sección **else** y no hay una expresión que se evalúe como **true**, la sentencia **case** generará la excepción **case_not_found**.

El siguiente ejemplo ilustra cómo usar una sentencia **case** buscada:

```
do $$
declare
    oficina_seleccionada oficinas%rowtype;
    id_oficina oficinas.identificador%type := 2;
    provincia varchar(20);
begin

    select * from oficinas
    into oficina_seleccionada
    where identificador = id_oficina;

    if not found then
        raise notice'La oficina % no puede ser encontrada.',
            id_oficina;
    else
        case
            when trunc(oficina_seleccionada.codigo_postal::int / 1000)= 28 then
                provincia := 'Madrid';
            when trunc(oficina_seleccionada.codigo_postal::int / 1000) = 30 then
                provincia := 'Murcia';
            when trunc(oficina_seleccionada.codigo_postal::int / 1000) = 27 then
                provincia := 'Jaén';
            when trunc(oficina_seleccionada.codigo_postal::int / 1000) = 36 then
                provincia := 'Granada';
            else
                provincia := 'Desconocida';
        end case;
        raise notice'La oficina con dirección en % está en la provincia de %.',
            oficina_seleccionada.domicilio, provincia;
    end if;
```

```
end $$;


NOTICE:  La oficina con dirección en Camino Ronda, 50 está en la provincia de Granada.
```

Cómo funciona:

El funcionamiento es muy similar al del ejemplo de la sentencia **case** simple.

5.3.- Bucle LOOP

Introducción a la sentencia LOOP de PL/pgSQL

LOOP define un bucle incondicional, que ejecuta un bloque de código repetidamente hasta que lo termina una instrucción **exit** o **return**.

A continuación se ilustra la sintaxis de la sentencia **loop**:

```
<<label>>
loop
    sentencias;
end loop;
```

Por lo general, se usa una sentencia **if** dentro del bucle para terminarlo en función de una condición como la siguiente:

```
<<label>>
loop
    sentencias;
    if condición then
        exit;
    end if;
end loop;
```

Es posible colocar una instrucción de bucle dentro de otra instrucción de bucle. Cuando una sentencia **loop** se coloca dentro de otra sentencia **loop**, se denomina bucle anidado:

```
<<outer>>
loop
    statements;
    <<inner>>
    loop
        /* ... */
        exit <<inner>>
    end loop;
end loop;
```

Cuando tiene bucles anidados, necesita usar la etiqueta del bucle para que pueda especificarla en la declaración exity continuepara indicar a qué bucle se refieren estas declaraciones.

Ejemplo de bucle LOOP PL/pgSQL

El siguiente ejemplo muestra cómo usar la instrucción **`loop`** para calcular el número de secuencia de Fibonacci.

```
do $$
declare
    n integer:= 10;
    fib integer := 0;
    counter integer := 0 ;
    i integer := 0 ;
    j integer := 1 ;
begin
    if (n < 1) then
        fib := 0 ;
    end if;
    loop
        exit when counter = n ;
        counter := counter + 1 ;
        select j, i + j into i, j ;
    end loop;
    fib := i;
    raise notice '%', fib;
end; $$;
```

```
NOTICE: 55
```

El bloque calcula el enésimo número de Fibonacci de un entero (**`n`**).

Por definición, los números de Fibonacci son una secuencia de números enteros que comienzan con 0 y 1, y cada número posterior es la suma de los dos números anteriores, por ejemplo, 1, 1, 2 (1+1), 3 (2+1), 5 (3+2), 8 (5+3), …

En la sección de declaración, la variable **`counter`** se inicializa a cero (**`0`**). El bucle termina cuando **`counter`** es igual a **`n`**. La siguiente declaración de selección intercambia valores de dos variables **`i`** y **`j`**:

```
SELECT j, i + j INTO i, j ;
```

¿Cómo salir de un LOOP?

Para salir de un bucle o de otros tipos de bloques, se puede utilizar la sentencia **`exit`**.

La documentación sobre su uso la puedes encontrar aquí.

5.4.- Bucle WHILE

La sentencia **`while`** ejecuta un bloque de código hasta que una **`condición`** se evalúa como **`false`**.

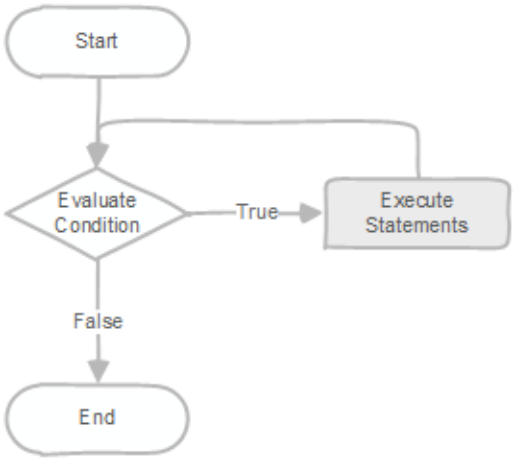
```
[ <<label>> ]
while condición loop
    sentencias;
end loop;
```

En esta sintaxis, PostgreSQL evalúa el **`condición`** antes de ejecutar el bloque de **`sentencias`**.

Si la **`condición`** es verdadera, ejecuta el bloque de **`sentencias`**. Después de cada iteración, el bucle **`while`** evalúa de nuevo **`condición`**.

Dentro del cuerpo del bucle **`while`**, debe cambiar los valores de algunas variables para hacer **`condición false`** o **`null`** en algunos puntos. De lo contrario, tendrá un bucle infinito.

El siguiente diagrama de flujo ilustra el bucle **`while`**.



El siguiente ejemplo utiliza un bucle **`while`** para mostrar el valor de un **`contador`**:

```
do $$
declare
    contador integer := 0;
begin
    while contador < 5 loop
        raise notice 'Contador %', contador;
        contador := contador + 1;
    end loop;
end$$;
```

```
NOTICE: Contador 0
NOTICE: Contador 1
NOTICE: Contador 2
NOTICE: Contador 3
NOTICE: Contador 4
```

Cómo funciona.

- Primero, declare la variable **`contador`** e inicialice su valor a **`0`**.
- En segundo lugar, utilice la sentencia **`while`** para mostrar el valor actual de **`contador`** siempre que sea inferior a 5. En cada iteración, aumente el valor de **`contador`** en uno. Después de 5 iteraciones, el **`contador`** es **`5`**, por lo tanto, el bucle **`while`** finaliza.

5.5.- Bucle FOR

El bucle **`for loop`** sirve para iterar sobre un rango de enteros.

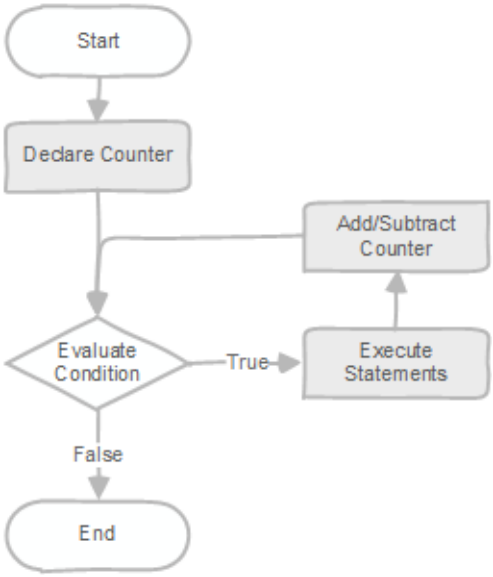
A continuación, se ilustra la sintaxis de la sentencia **`for loop`** que recorre un rango de enteros:

```
[ <<label>> ]
for contador in [ reverse ] from .. to [ by step ] loop
    statements
end loop [ label ];
```

En esta sintaxis:

- Primero, **`for loop`** crea la variable entera **`contador`** a la que solo se puede acceder dentro del bucle.
- En segundo lugar, **`from`** y **`to`** son expresiones que especifican el límite inferior y superior del rango. El bucle **`for`** evalúa estas expresiones antes de entrar en el bucle.
- Tercero, el **`step`** que sigue a la palabra clave **`by`** especifica el paso de iteración. El valor predeterminado es 1.
- De forma predeterminada, **`for loop`** incrementa **`contador`** en la cantidad indicada por **`step`** después de cada iteración. Sin embargo, cuando usa la opción **`reverse`**, lo que se hace es decrementar **`contador`**.

El siguiente diagrama de flujo ilustra la sentencia **`for loop`**:



El siguiente ejemplo usa la sentencia **`for loop`** para iterar sobre cinco números del 1 al 5 y mostrar cada uno de ellos en cada iteración:

```
do $$
begin
    for contador in 1..5 loop
        raise notice 'contador: %', contador;
    end loop;
end; $$;
```

```
NOTICE:  contador: 1
NOTICE:  contador: 2
NOTICE:  contador: 3
```

```
NOTICE:  contador: 4
NOTICE:  contador: 5
```

El siguiente ejemplo itera sobre 5 números del 5 al 1 y muestra cada número en cada iteración:

```
do $$
begin
    for contador in reverse 5..1 loop
        raise notice 'contador: %', contador;
    end loop;
end; $$;
```

```
NOTICE:  contador: 5
NOTICE:  contador: 4
NOTICE:  contador: 3
NOTICE:  contador: 2
NOTICE:  contador: 1
```

El siguiente ejemplo usa la sentencia **`for loop`** para iterar sobre seis números del 1 al 6. Agrega 2 al contador después de cada iteración:

```
do $$
begin
    for contador in 1..6 by 2 loop
        raise notice 'contador: %', contador;
    end loop;
end; $$;
```

```
NOTICE:  contador: 1
NOTICE:  contador: 3
NOTICE:  contador: 5
```

Uso de for loop para iterar sobre un conjunto de resultados

La siguiente declaración muestra cómo usar la sentencia **`for loop`** para iterar sobre un conjunto de resultados de una consulta:

```
[ <<label>> ]
for variable in consulta loop
    sentencias
end loop [ label ];
```

La siguiente declaración usa el bucle **`for`** para mostrar los nombres de los agentes con la habilidad 7.

```
do $$
declare
    agente record;
begin
    for agente in select nombre
        from agentes
        where habilidad = 7
    loop
        raise notice '%', agente.nombre;
    end loop;
end;
$$;
```

```
NOTICE:  Salvador Romero Villegas
NOTICE:  José Javier Bermúdez Hernández
NOTICE:  Alfonso Bonillo Sierra
NOTICE:  Silvia Thomas Barrós
```

Uso de for loop para iterar sobre el conjunto de resultados de una consulta dinámica.

La siguiente forma de la sentencia **`for loop`** le permite ejecutar una consulta dinámica e iterar sobre su conjunto de resultados:

```
[ <<label>> ]
for fila in execute consulta [ using parámetro [, ... ] ]
loop
    sentencias
end loop [ label ];
```

En esta sintaxis:

- La **`consulta`** es una instrucción SQL.
- La cláusula **`using`** se utiliza para pasar parámetros a la consulta.

El siguiente bloque muestra cómo usar la sentencia **`for loop`** para recorrer una consulta dinámica. Tiene dos variables de configuración:

- **`tipo_orden`**: 1 para ordenar los agentes por nombre, 2 para ordenar los agentes por habilidad.
- **`contador_registros`**: es el número de filas a consultar de la tabla **`agentes`**. Lo usaremos en la cláusula **`using`** del bucle **`for`**.

Este bloque anónimo compone la consulta en función de la variable **`tipo_orden`** y utiliza el bucle **`for`** para iterar sobre la fila del conjunto de resultados.

```
do $$
declare
    -- orden 1: nombre, 2: habilidad
```

```
    tipo_orden smallint := 1;
    -- devuelve el número de agentes
    contador_registros int := 5;
    -- usado para iterar sobre el agente
    fila record;
    -- consulta dinámica
    consulta text;
begin

    consulta := 'select nombre, habilidad from agentes ';

    if tipo_orden = 1 then
        consulta := consulta || 'order by nombre';
    elsif tipo_orden = 2 then
        consulta := consulta || 'order by habilidad';
    else
        raise 'orden no establecido correctamente %', consulta;
    end if;

    consulta := consulta || ' limit $1';

    for fila in execute consulta using contador_registros
    loop
        raise notice '% - %', fila.nombre, fila.habilidad;
    end loop;
end;
$$;
```

```
NOTICE:  Alfonso Bonillo Sierra - 7
NOTICE:  Antonio Álvarez Palomeque - 5
NOTICE:  Antonio Casado Fernández - 5
NOTICE:  Antonio Fernández Ruíz - 5
NOTICE:  Cayetano García Herrera - 5
```

Si cambia **`tipo_orden`** a 2, obtendrá el siguiente resultado:

```
NOTICE:  Pedro Fernández Arias - 5
NOTICE:  Vanesa Sánchez Rojo - 5
NOTICE:  Francisco Javier García Escobedo - 5
NOTICE:  Pilar Ramirez Pérez - 5
NOTICE:  José Luis García Martínez - 5
```

Saltar a la siguiente iteración.

Con la sentencia **`continue`** puedes obviar el resto de la iteración actual y comenzar la siguiente iteración.

Puedes aprender a utilizar la sentencia **`continue`** en la documentación oficial.

6.- Rutinas.

Las **rutinas** son bloques de código PL/pgSQL, referenciados bajo un nombre, que realizan una acción determinada. Les podemos pasar parámetros y los podemos invocar. Además, las rutinas pueden estar almacenados en la base de datos o estar encerrados en otros bloques. Si el programa está almacenado en la base de datos, podremos invocarlo si tenemos permisos suficientes y si está encerrado en otro bloque lo podremos invocar si tenemos visibilidad sobre el mismo.

Hay dos clases de rutinas: las funciones y los procedimientos.

6.1.- Funciones.

La sentencia **`CREATE FUNCTION`** le permite definir una nueva función definida por el usuario.

A continuación se ilustra la sintaxis de la sentencia **`CREATE FUNCTION`**:

```
create [or replace] function nombre_función(lista_parámetros)
returns tipo_retorno
language plpgsql
as
$$
declare
    -- declaración de variables
begin
    -- lógica de la función
end;
$$
```

En esta sintaxis:

- Primero, especifique el nombre de la función después de las palabras clave **`CREATE FUNCTION`**. Si desea reemplazar la función existente, puede utilizar las palabras clave **`OR REPLACE`**.
- Luego, especifique la lista de parámetros de la función entre paréntesis después del nombre de la función. Una función puede tener cero o muchos parámetros.
- A continuación, especifique el tipo de datos del valor devuelto después de la palabra clave **`RETURNS`**.
- Después de eso, use **`language plpgsql`** para especificar el lenguaje de procedimiento de la función. Tenga en cuenta que PostgreSQL admite muchos lenguajes de procedimiento, no solo **`plpgsql`**.
- Finalmente, coloque un bloque delimitado por dólar.

Ejemplos de declaraciones de funciones.

La siguiente instrucción crea una función que cuenta los agentes cuya categoría se encuentra entre los parámetros **`cat_desde`** y **`cat_hasta`**:

```
create function get_agentes_count(cat_desde int, cat_hasta int)
returns int
language plpgsql
as
$$
declare
    contador_agentes integer;
begin
    select count(*)
    into contador_agentes
    from agentes
    where categoria between cat_desde and cat_hasta;

    return contador_agentes;
end;
$$;
```

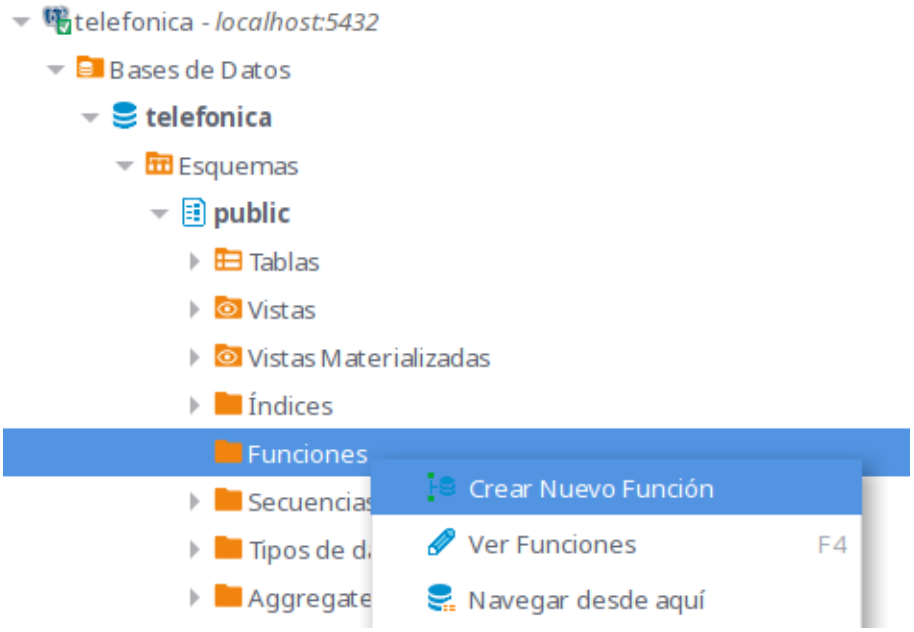
La función `get_film_count()` tiene dos secciones principales: **encabezado** y **cuerpo**.

- En la sección de encabezado:
    1. Primero, el nombre de la función es `get_agentes_count` que sigue a las palabras clave `create function`.
    2. En segundo lugar, la función `get_agentes_count()` acepta dos parámetros `cat_desde` y `cat_hasta`, con el tipo de datos entero.
    3. En tercer lugar, la función `get_agentes_count()` devuelve un número entero especificado por la cláusula `returns int`.
    4. Finalmente, el idioma de la función se indica mediante `language plpgsql`.
- En el cuerpo de la función:
    - Use la sintaxis constante de cadena delimitada por dólares que comienza `$$` y termina con `$$`. Entre estos `$$`, puede colocar un bloque que contenga la *declaración* y la *lógica de la función*.
    - En la sección de *declaración*, declare una variable llamada `contador_agentes` que almacena el número de agentes seleccionados de la tabla `agentes`.
    - En el *cuerpo del bloque*, utiliza la sentencia `select into` para seleccionar el número de agentes cuya categoría se encuentra entre `cat_desde` y `cat_hasta` y asigne el resultado a la variable `contador_agentes`.
    - Al final del bloque, use la instrucción `return` para devolver el `contador_agentes`.

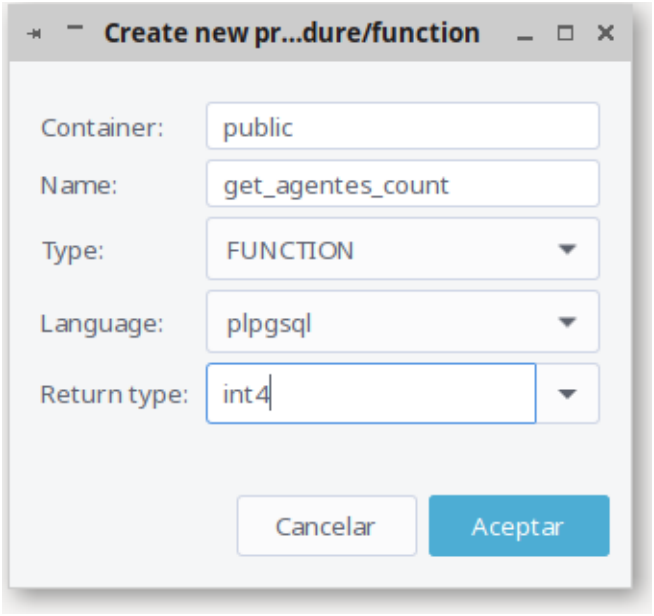
Para ejecutar la declaración de función de creación, puede usar cualquier herramienta de cliente de PostgreSQL, incluidos `psql` y DBeaver.

Crear una función usando DBeaver.

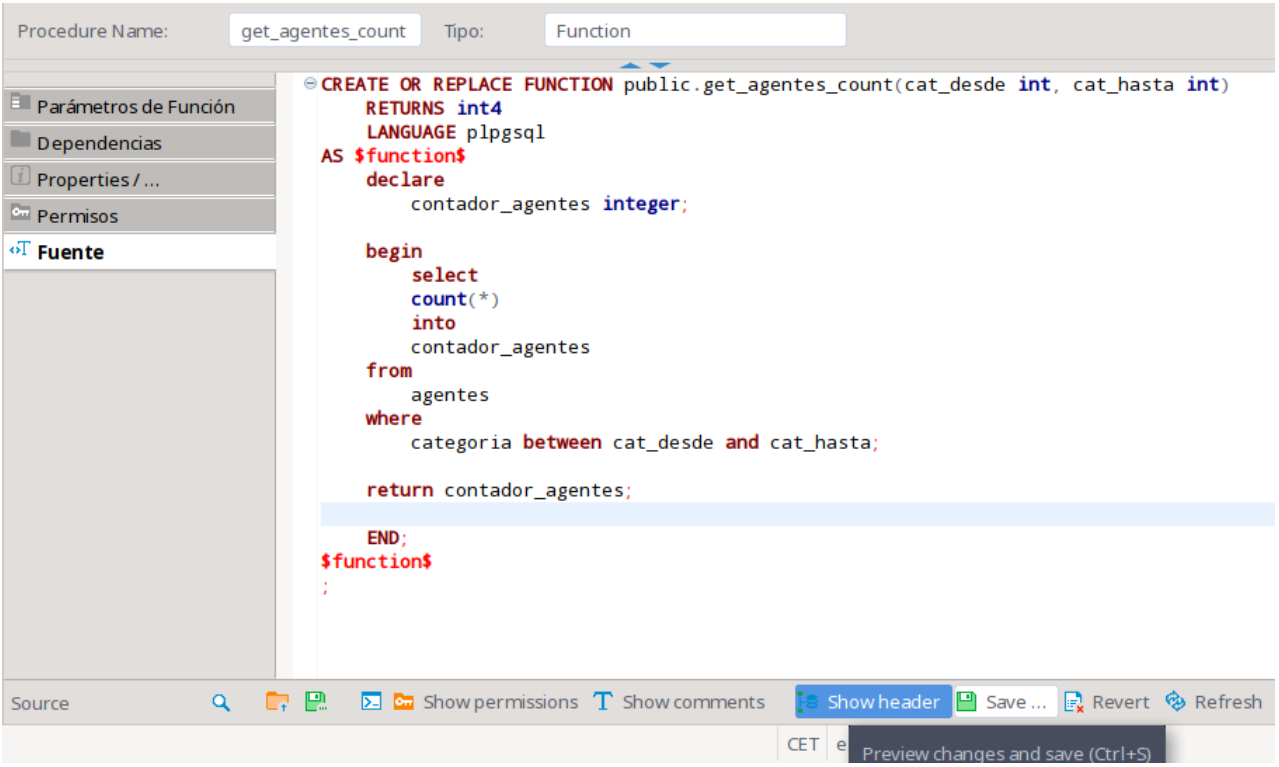
1. En primer lugar, desplegamos la base de datos de ejemplo a través del navegador de DBeaver y seleccionamos **"Crear Nueva Función"**.



2. En la ventana flotante, rellenamos las propiedades de la cabecera.



3. Seleccionamos la pestaña **Fuente** de la izquierda y rellenamos el cuerpo de la función, así como sus parámetros.



   Al finalizar, pulsamos sobre el botón inferior **Save** para guardar la función en la base de datos.

Fíjate que DBeaver incluye la etiqueta **`function`** en el delimitador del bloque de la función, estableciéndose el delimitador como **`$function$`**.

Llamar a la función definida por el usuario.

PostgreSQL le proporciona tres formas de llamar a una función definida por el usuario. No obstante, aquí se explicará la notación posicional, en la que se deben especificar los argumentos en el mismo orden en el que se establecieron los parámetros.

En el caso de la función get_agentes_count(), la invocaremos de la siguiente forma:

```
select get_agentes_count(1, 2);
```

```
 get_agentes_count
-------------------
                 9
(1 fila)
```

o:

```
select get_agentes_count(2, 3);
```

```
 get_agentes_count
-------------------
                 3
(1 fila)
```

6.1.1.- Modos de los parámetros.

Los modos de parámetro determinan el comportamiento de los parámetros. PL/pgSQL admite tres modos de parámetros: **`in`**, **`out`** y **`inout`**. Un parámetro toma el modo **`in`** por defecto si no lo especifica OTRO explícitamente.

La siguiente tabla ilustra los tres modos de parámetros:

IN	**OUT**	**INOUT**
El valor por defecto	Especificado explícitamente	Especificado explícitamente
Pasar un valor a la función	Devolver un valor de una función	Pase un valor a una función y devuelva un valor actualizado.
los parámetros **`in`** actúan como constantes	los parámetros actúan **`out`** como variables no inicializadas	los parámetros **`inout`** actúan como variables inicializadas
No se le puede asignar un valor	Debe asignar un valor	Se le debe asignar un valor

El modo IN

La siguiente función encuentra una oficina por su **`identificador`** y devuelve su **`domicilio`**:

```
create or replace function find_oficina_by_id(p_oficina_id int)
returns varchar
language plpgsql
as $$
declare
    oficina_domicilio oficinas.domicilio%type;
begin
    -- encuentra el domicilio de la oficina por su identificador
    select domicilio
    into oficina_domicilio
    from oficinas
    where identificador = p_oficina_id;

    if not found then
        raise 'Oficina con el identificador % no encontrada', p_oficina_id;
    end if;

    return oficina_domicilio;

end;$$;
```

Debido a que no especificamos el modo para el parámetro **`p_oficina_id`**, toma el modo **`in`** por defecto.

```
select find_oficina_by_id(1);
```

```
 find_oficina_by_id
--------------------
 Gran vía, 37
(1 fila)
```

```
select find_oficina_by_id(5);
```

```
ERROR:  Oficina con el identificador 5 no encontrada
CONTEXTO:  función PL/pgSQL find_oficina_by_id(integer) en la línea 12 en RAISE
```

El modo OUT

Los parámetros **`out`** se definen como parte de la lista de argumentos y se devuelven como parte del resultado.

Los parámetros **`out`** son muy útiles en funciones que necesitan devolver múltiples valores.

Para definir parámetros **`out`**, hay que anteponer la palabra clave **`out`** al parámetro de la siguiente forma:

```
out nombre_parámetro tipo
```

La siguiente función utiliza, además del parámetro correspondiente al **`identificador`** de la oficina, 3 parámetros **`out`** para devolver su **`nombre`**, su **`domicilio`** y su **`codigo_postal`**:

```
create or replace function get_datos_oficina(
    in p_oficina_id int,
    out p_nombre varchar(40),
    out p_domicilio varchar(40),
    out p_codigo_postal varchar(5)
)
language plpgsql
as $$
begin
    -- encuentra los datos de la oficina por su identificador
    select nombre, domicilio, codigo_postal
    into p_nombre, p_domicilio, p_codigo_postal
    from oficinas
    where identificador = p_oficina_id;

    if not found then
        raise 'Oficina con el identificador % no encontrada', p_oficina_id;
    end if;

end;$$;
```

```
select get_datos_oficina(1);
```

```
       get_datos_oficina
------------------------------
 (Madrid,"Gran vía, 37",28000)
(1 fila)
```

La salida de la función es un registro. Para hacer que la salida se separe como columnas, use la siguiente sentencia:

```
select * from get_datos_oficina(1);
```

```
 p_nombre | p_domicilio  | p_codigo_postal
----------+--------------+-----------------
 Madrid   | Gran vía, 37 | 28000
(1 fila)
```

El modo INOUT

El modo **`inout`** es la combinación de los modos **`in`** y **`out`**.

Significa que la persona que invoca la función puede pasar un argumento a una función; la función, posteriormente, cambia el valor de dicho argumento y devuelve el valor actualizado.

La siguiente función **`swap`** acepta dos enteros e intercambia sus valores:

```
create or replace function swap(
    inout x int,
    inout y int
)
language plpgsql
as $$
begin
   select x,y into y,x;
end; $$;
```

```
select * from swap(10,20);
```

```
 x  | y
----+----
 20 | 10
(1 fila)
```

6.2.- Procedimientos.

Las explicaciones sobre utilización de funciones definidas por el usuario también se aplicarán a los procedimientos, excepto por lo siguiente:

- Los procedimientos se definen con el comando **`CREATE PROCEDURE`**, no **`CREATE FUNCTION`**.
- Los procedimientos no devuelven un valor; por lo tanto, **`CREATE PROCEDURE`** carece de una cláusula **`RETURNS`**. Sin embargo, los procedimientos pueden, en cambio, devolver datos a través de parámetros **`output`**.
- Mientras que una función se llama como parte de una consulta o un comando **`DML`**, un procedimiento se llama de forma aislada usando el comando **`CALL`**
- Un procedimiento puede utilizar transacciones. Una función no puede hacer eso.

CREATE PROCEDURE

Para definir un nuevo procedimiento almacenado, utilice la sentencia **`create procedure`**, según la siguiente sintaxis

```
create [or replace] procedure nombre_procedimiento(lista_parámetros)
language plpgsql
as $$
declare
    -- declaración de variables
begin
    -- cuerpo del procedimiento almacenado
end; $$
```

En esta sintaxis:

- Primero, especifique el nombre del procedimiento después de las palabras clave **`CREATE PROCEDURE`**. Si desea reemplazar un procedimiento existente, puede utilizar las palabras clave **`OR REPLACE`**.
- Luego, especifique la lista de parámetros del procedimiento entre paréntesis, después de su nombre. Un procedimiento almacenado puede tener cero o muchos parámetros.
- En tercer lugar, use **`language plpgsql`** para especificar el lenguaje de procedimiento del procedimiento. Tenga en cuenta que PostgreSQL admite muchos lenguajes de procedimiento, no solo **`plpgsql`**.
- Finalmente, coloque un bloque delimitado por dólar como cuerpo del procedimiento almacenado.

Un procedimiento almacenado no devuelve un valor. **No** puede usar la sentencia **`return`** con un valor dentro de un procedimiento.

Ejemplo de creación de procedimiento almacenado.

Vamos a crear un sencillo procedimiento almacenado que permita insertar un nuevo agente en la tabla **agentes**:

```
create or replace procedure inserta_agente(
    p_identificador int,
    p_nombre varchar(60),
    p_usuario varchar(20),
    p_clave varchar(20),
    p_habilidad int,
    p_categoria int
)
language plpgsql
as $$
begin
    insert into agentes (identificador,nombre, usuario, clave, habilidad, categoria)
    values (p_identificador, p_nombre, p_usuario, p_clave, p_habilidad, p_categoria);

end;$$;
```

Si queremos invocarlo, una vez creado, utilizaremos la sentencia **CALL** seguido del nombre del procedimiento:

```
ccall inserta_agente(2222, 'Federico Fernández', 'fedfer', 'ferfed2', 1, 1);
```

El resultado será la inserción del nuevo agente con los datos facilitados. Evidentemente, dentro de este procedimiento se podrían/deberían realizar comprobaciones previas a la inserción (que la clave fuera segura p.e.), así como la captura de posibles problemas en la inserción, como la inserción de duplicados.

6.3.- Eliminar rutinas.

DROP FUNCTION

Para eliminar una función definida por el usuario, utilice la sentencia **`drop function`**:

```
drop function [if exists] function_name(argument_list)
[cascade | restrict]
```

En esta sintaxis:

1. Primero, especifique el nombre de la función que desea eliminar después de las palabras clave **`drop function`**.
2. En segundo lugar, utilice la opción **`if exists`** si desea indicar a PostgreSQL que emita un aviso en lugar de un error en caso de que la función no exista.
3. Tercero, especifique la lista de argumentos de la función. Dado que las funciones se pueden sobrecargar, PostgreSQL necesita saber qué función desea eliminar comprobando la lista de argumentos. Si una función es única dentro del esquema, no necesita especificar la lista de argumentos.

Cuando una función tiene objetos dependientes, como operadores o disparadores, no se puede borrar esa función sin utilizar la opción **`cascade`**. En el caso de utilizar **`cascade`**, se eliminará recursivamente la función, sus objetos dependientes y los objetos que dependen de esos objetos, y así sucesivamente. De forma predeterminada, la sentencia **`drop function`** utiliza la opción **`restrict`** que rechaza la eliminación de una función cuando tiene objetos dependientes.

Ejemplo:

```
drop function if exists get_agentes_count cascade;
```

DROP PROCEDURE

Para eliminar un procedimiento almacenado, utilice la sentencia **`drop procedure`**:

```
drop procedure [if exists] procedure_name (argument_list)
[cascade | restrict]
```

En esta sintaxis:

1. Primero, especifique el nombre del procedimiento almacenado que desea eliminar después

de las palabras clave **`drop procedure`**.
2. En segundo lugar, utilice la opción **`if exists`** si desea indicar a PostgreSQL que emita un aviso en lugar de un error en caso de que el procedimiento no exista.
3. Tercero, especifique la lista de argumentos del procedimiento. Dado que los procedimientos se pueden sobrecargar, PostgreSQL necesita saber qué procedimiento desea eliminar comprobando la lista de argumentos. Si un procedimiento es único dentro del esquema, no necesita especificar la lista de argumentos.

Ejemplo:

```
drop procedure inserta_agente;
```

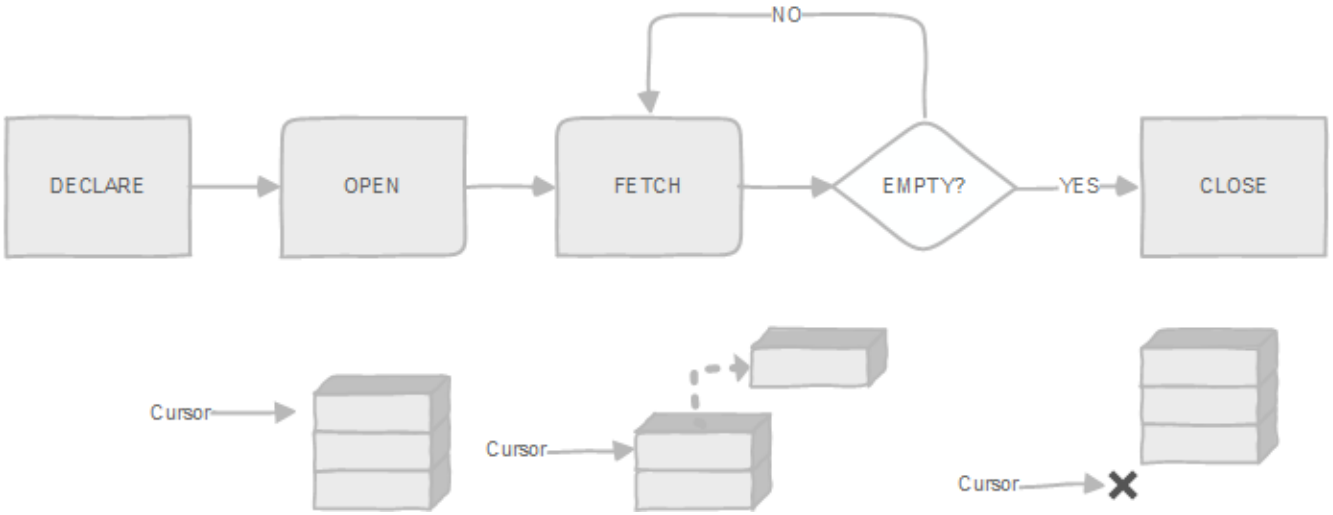
7.- Cursores

Un cursor PL/pgSQL le permite encapsular una consulta y procesar cada fila individual a la vez.

Por lo general, utiliza cursores cuando desea dividir un gran conjunto de resultados en partes y procesar cada parte individualmente. Si lo procesa a la vez, es posible que tenga un error de desbordamiento de memoria.

Además de eso, puede desarrollar una función que devuelva una referencia a un cursor. Esta es una forma efectiva de devolver un gran conjunto de resultados de una función. La persona que llama a la función puede procesar el conjunto de resultados en función de la referencia del cursor.

El siguiente diagrama ilustra cómo usar un cursor en PostgreSQL:



1. Primero, declare un cursor.
2. A continuación, abra el cursor.
3. Luego, obtenga filas del conjunto de resultados en un destino.
4. Después de eso, verifique si queda más fila para buscar. En caso afirmativo, vaya al paso 3, de lo contrario, vaya al paso 5.
   Finalmente, cierre el cursor.

Examinaremos cada paso con más detalle en las siguientes secciones.

Declarar un cursor.

Para acceder a un cursor, debe declarar una variable de tipo cursor en la sección de declaración de un bloque. PostgreSQL le proporciona un tipo especial llamado **`REFCURSOR`** para declarar una variable de cursor.

```
declare mi_cursor refcursor; -- En PL/SQL se conocen como cursor variable.
```

También puede declarar un cursor que se limite a una consulta usando la siguiente sintaxis:

```
nombre_cursor [ [no] scroll ] cursor [( nombre tipo_dato, nombre tipo_dato, ...)] for consul
```

1. Primero, especifique un nombre de variable (**`nombre_cursor`**) para el cursor.
2. A continuación, especifique si el cursor se puede desplazar hacia atrás utilizando el **`scroll`**. Si usa **`no scroll`**, el cursor no se puede desplazar hacia atrás.
3. Luego, coloca la palabra clave **`cursor`** seguida de una lista de argumentos separados por comas (**`nombre tipo_dato`**) que definen los parámetros para la consulta. Estos argumentos serán sustituidos por valores cuando se abra el cursor.
4. Después de eso, especifica una consulta después de la palabra clave **`for`**. Puede usar cualquier instrucción **`SELECT`** válida aquí.

El siguiente ejemplo ilustra cómo declarar cursores:

```
declare    -- cursor sin parámetros
    cur_oficinas cursor for
        select *
        from oficinas;    -- cursor con parámetros
    cur_oficinas2 cursor (p_localidad varchar) for
        select *
        from oficinas
        where localidad = p_localidad;
```

`cur_oficinas` es un cursor que encapsula todas las filas de la tabla **`oficinas`**.

`cur_oficinas2` es un cursor que encapsula las oficinas de una determinada localidad.

Apertura de cursores.

Los cursores deben abrirse antes de que puedan usarse para consultar filas. PostgreSQL proporciona la sintaxis para abrir un cursor enlazado y no enlazado.

Apertura de cursores independientes

Un cursor independiente se abre usando la siguiente sintaxis:

```
OPEN cursor_independiente [ [ NO ] SCROLL ] FOR consulta;
```

Debido a que la variable de cursor independiente no está enlazada a ninguna consulta cuando la declaramos, debemos especificar la consulta cuando la abrimos. Vea el siguiente ejemplo:

```
open mi_cursor for
    select * from oficinas
        where localidad = p_localidad
```

PostgreSQL le permite abrir un cursor y vincularlo a una consulta dinámica. Aquí está la sintaxis:

```
open cursor_independiente[ [ no ] scroll ]
for execute consulta [using expresión [, ... ] ];
```

En el siguiente ejemplo, construimos una consulta dinámica que ordena las filas en función de un parámetro **`tipo_orden`** y abrimos el cursor que ejecuta la consulta dinámica.

```
consulta := 'select * from oficinas order by $1';

open cur_oficinas for execute consulta using tipo_orden;
```

Apertura de cursores enlazados

Debido a que un cursor enlazado ya está enlazado a una consulta cuando lo declaramos, entonces, cuando lo abrimos, solo necesitamos pasar los argumentos a la consulta si es necesario.

```
open nombre_cursor[ (nombre:=valor,nombre:=valor,...)];
```

En el siguiente ejemplo, abrimos los cursores enlazados cur_oficinas y cur_oficinas2 que declaramos anteriormente:

```
open cur_oficinas;
open cur_oficinas2(p_localidad = 'Granada');
```

Uso de cursores

Después de abrir un cursor, podemos manipularlo usando las instruccies **`FETCH`**, **`MOVE`**, **`UPDATE`** o **`DELETE`** .

Obtener la siguiente fila

```
fetch [ dirección { from | in } ] nombre_cursor into variable;
```

La instrucción **fetch** obtiene la siguiente fila del cursor y la asigna una **variable**, que podría ser un registro, una variable de fila o una lista de variables separadas por comas. Si no se encuentran más filas, se establece la **variable** a **NULL**(s).

De forma predeterminada, un cursor obtiene la siguiente fila si no especifica la dirección explícitamente.

```
fetch cur_oficinas into fila_oficina;
```

Desplazamiento en los cursores.

Para el ámbito de estos contenidos, vamos a utilizar el movimiento por defecto de los cursores, utilizando FETCH INTO. No obstante, PL/pgSQL permite desplazarse casi libremente por los cursores. Puede encontrar más información en la sección Using cursors del tutorial de cursores de PostgreSQL.

Eliminar o actualizar la fila.

Una vez que nos posicionamos en una fila de un cursor, podemos eliminarla o actualizarla usando las sentencias **DELETE WHERE CURRENT OF** o **UPDATE WHERE CURRENT OF** de la siguiente manera:

```
update table_name
set column = value, ...
where current of cursor_variable;

delete from table_name
where current of cursor_variable;
```

Vea el siguiente ejemplo.

```
update oficinas
set codigo_postal = 18000
where current of cur_oficinas;
```

Cerrar un cursor.

Para cerrar un cursor, usaremos la instrucción **CLOSE** de la siguiente manera:

```
close nombre_cursor;
```

La instrucción **CLOSE** libera recursos o libera la variable del cursor para permitir que se abra nuevamente usando la sentencia **OPEN**.

Ejemplo completo de un cursor.

La siguiente función **get_agentes_nombres** acepta un argumento que representa la **habilidad** de un agente. Dentro de la función, consultamos todos los agentes que tienen la habilidad pasada a la función. Usamos el cursor para recorrer las filas y concatenar el nombre y la habilidad de aquellos agentes que tiene la **categoría** 2.

```
create or replace function get_agentes_nombres(p_habilidad integer)
    returns text
    language plpgsql
as $$
declare
    nombres text default '';
    fila_agente record;
    cur_agentes cursor(p_habilidad integer)
        for select nombre, categoria, habilidad
        from agentes
        where habilidad = p_habilidad;
begin
    -- apertura del cursor
    open cur_agentes(p_habilidad);

    loop
        -- fetch la fila en el registro de agente
        fetch cur_agentes into fila_agente;
        -- salir cuando no haya más filas a recorrer
        exit when not found;

        -- construir el mensaje
        if fila_agente.categoria = 1 then
            nombres := nombres || ',' || fila_agente.nombre || ':' || fila_agente.habilidad;
        end if;
    end loop;

    -- cerrar el cursor
    close cur_agentes;

    return nombres;
end;
$$;
```

```
select get_agentes_nombres(7);
```

```
                                                get_agentes_nombres
-------------------------------------------------------------------------------------------
 ,Salvador Romero Villegas:7,José Javier Bermúdez Hernández:7,Alfonso Bonillo Sierra:7
```

(1 fila)

8.- Disparadores.

Un disparador de PostgreSQL es una función que se invoca automáticamente cada vez que ocurre un evento como insertar, actualizar o eliminar registros de una tabla. En esta sección, aprenderá acerca de los factores desencadenantes y cómo administrarlos de manera efectiva.

Introducción a los disparadores en PostgreSQL.

Un disparador de PostgreSQL es una función que se invoca automáticamente cada vez que ocurre un evento asociado con una tabla. Un evento podría ser cualquiera de los siguientes: **`INSERT`**, **`UPDATE`**, **`DELETE`** o **`TRUNCATE`** .

Un disparador es una función especial definida por el usuario asociada con una tabla. Para crear un nuevo disparador, primero definiremos una función de disparador y luego vincularemos esta función de disparador a una tabla. La diferencia entre un disparador y una función definida por el usuario es que un disparador se invoca automáticamente cuando ocurre un evento desencadenante.

PostgreSQL proporciona dos tipos principales de disparadores:

- disparadores a nivel de fila.
- disparadores a nivel de sentencia.

Las diferencias entre los dos tipos son cuántas veces se invoca el disparador y en qué momento.

Por ejemplo, si ejecutamos una sentencia **`UPDATE`** que afecta a 20 filas, el disparador de nivel de fila se invocará 20 veces, mientras que el disparador a nivel de sentencia se invocará 1 vez.

Podemos especificar si el disparador se invocará **`before`** o **`after`** de un evento. Si el disparador se invoca **`before`** de un evento, podremos omitir la operación para la fila actual o incluso cambiar la fila que se está actualizando o insertando. En caso de que el disparador se invoque **`after`** del evento, todos los cambios estarán disponibles para el disparador.

Los disparadores son útiles en caso de que varias aplicaciones accedan a la base de datos y se desee ejecutar automáticamente cierta funcionalidad, cada vez que se modifican los datos de la tabla, independientemente de la aplicación que provoca dicha modificación. Por ejemplo, si desea mantener el historial de datos sin requerir que la aplicación tenga lógica para verificar cada evento como **`INSERT`** o **`UPDATE`.**

También se pueden usar disparadores para mantener reglas de integridad de datos complejas que no se pueden implementar en ningún otro lugar, excepto en el nivel de la base de datos. Por ejemplo, cuando se agrega una nueva fila a una tabla de **clientes**, es probable que también se deben crear otras filas en las tablas de **bancos** y **créditos**.

El principal inconveniente de usar un disparador es que debemos conocer de su existencia y comprender su lógica, para conocer sus efectos cuando cambian los datos.

Aunque PostgreSQL implementa el estándar SQL, los disparadores en PostgreSQL tienen algunas características específicas:

- PostgreSQL activa el disparador para el evento **`TRUNCATE`**.
- PostgreSQL permite definir un disparador a nivel de sentencia en las vistas.
- PostgreSQL requiere que defina una función definida por el usuario como la acción del disparador, mientras que el estándar SQL le permite usar cualquier comando SQL.

Crear un disparador.

Para crear un nuevo activador en PostgreSQL, siga estos pasos:

1. Primero, cree una función de activación usando la sentencia CREATE FUNCTION.
2. En segundo lugar, vincule la función de activación a una tabla utilizando la sentencia **`CREATE TRIGGER`**.

Crar la función de activación.

Una función de activación es similar a una función normal definida por el usuario . Sin embargo, una función de activación **no toma ningún argumento** y tiene un valor de retorno con el tipo **trigger**.

A continuación se ilustra la sintaxis de la creación de la función de activación:

```
CREATE FUNCTION nombre_función()
RETURNS TRIGGER
LANGUAGE PLPGSQL
AS $$
BEGIN
    -- lógica del disparador
END;
$$;
```

Una función de disparador recibe datos sobre su entorno de llamada a través de una estructura especial llamada **`TriggerData`**, que contiene un conjunto de variables locales.

Por ejemplo, **`OLD`** y **`NEW`** representan los estados de la fila en la tabla antes(**`OLD`**) o después(**`NEW`**) del evento desencadenante.

PostgreSQL también proporciona otras variables locales precedidas por **`TG_`** , **`TG_WHEN`** y **`TG_TABLE_NAME`**.

Una vez que define una función de activación, puede vincularla a uno o más eventos de activación, como **`INSERT`**, **`UPDATE`** y **`DELETE`**.

CREATE TRIGGER

La CREATE TRIGGER instrucción crea un nuevo disparador. A continuación se ilustra la sintaxis básica de la CREATE TRIGGERinstrucción:

```
CREATE TRIGGER NOMBRE_DISPARADOR
{BEFORE | AFTER} { evento }
ON nombre_tabla
[FOR [EACH] { ROW | STATEMENT }]
EXECUTE PROCEDURE función_activación
```

En esta sintaxis:

1. Primero, especifique el nombre del disparador después de las palabras clave **`CREATE TRIGGER`**.
2. En segundo lugar, especifique el tiempo que hace que se dispare el disparador. Puede ser **`BEFORE`** o **`AFTER`** de que se produzca un evento.
3. En tercer lugar, especifique el evento que invoca el disparador. El evento puede ser **`INSERT`**, **`DELETE`**, **`UPDATE`** o **`TRUNCATE`**.
4. Cuarto, especifique el nombre de la tabla asociada con el disparador, después de la palabra clave **`ON`**.
5. En quinto lugar, especifique el tipo de disparadores que pueden ser:
   - Desencadenador de nivel de fila especificado por la cláusula **`FOR EACH ROW`**.
   - Desencadenador de nivel de instrucción que se especifica en la cláusula **`FOR EACH STATEMENT`**.

   Se activa un activador de nivel de fila para cada fila, mientras que se activa un activador de nivel de instrucción para cada transacción.

   Suponga que una tabla tiene 100 filas y dos activadores que se activarán cuando ocurra un evento **`DELETE`**.

   - Si la sentencia **`DELETE`** elimina 100 filas, el activador de nivel de fila se activará 100 veces, una vez por cada fila eliminada.
   - Por otro lado, un disparador de nivel de sentencia se activará una sola vez, independientemente de cuántas filas se eliminen.
6. Finalmente, especifique el nombre de la función de activación después de las palabras clave **`EXECUTE PROCEDURE`**.

Ejemplo de disparador.

Supongamos que cuando cambia el nombre de un agente, deseamos registrar los cambios en una tabla separada llamada audita_agente:

```
CREATE TABLE audita_agente (
    id INT GENERATED ALWAYS AS IDENTITY,
    identificador INT NOT NULL,
    nombre VARCHAR(40) NOT NULL,
    cambio_en TIMESTAMP(6) NOT NULL
);
```

1. Primero, crearemos una nueva función llamada **`log_cambio_nombre`**:

```
CREATE OR REPLACE FUNCTION log_cambio_nombre()
RETURNS TRIGGER
LANGUAGE PLPGSQL
AS
$$
BEGIN
    IF NEW.nombre <> OLD.nombre THEN
    INSERT INTO audita_agente(identificador,nombre,cambio_en)
    VALUES(OLD.identificador,OLD.nombre,now());
    END IF;

    RETURN NEW;
END;
$$;
```

La función inserta el **nombre** anterior en la tabla **audita_agente**, incluido el **identificador** del agente, y el momento del cambio.

OLD representa la fila antes de la actualización, mientras que **NEW** representa la nueva fila que se actualizará. **OLD.nombre** devuelve el nombre antes de la actualización y **NEW.nombre** devuelve el nuevo nombre.

2. En segundo lugar, vincule la función de activación a la tabla **agentes**. El nombre del activador es **log_cambio_nombre**. Antes de que se actualice el valor de la columna **nombre**, la función de activación se invoca automáticamente para registrar los cambios.

```
CREATE TRIGGER cambio_nombre
BEFORE UPDATE
ON agentes
FOR EACH ROW
EXECUTE PROCEDURE log_cambio_nombre();
```

Supongamos que *Pedro Fernández Arias* cambia su nombre por el de *Pilar Fernández Arias*.

3. Tercero, actualice el nombre de *Pedro Fernández Arias* al nuevo:

```
UPDATE agentes
SET nombre = 'Pilar Fernández Arias'
WHERE identificador = 311;
```

4. Cuarto, comprobemos si se ha actualizado el nombre de *Pedro Fernández Arias*:

```
SELECT * FROM agentes WHERE identificador = 311;
```

```
 identificador |        nombre         | usuario | clave | habilidad | categoria
---------------+-----------------------+---------+-------+-----------+-----------
           311 | Pilar Fernández Arias | pfa     | ag311 |         5 |         0
(1 fila)
```

Como puede ver en la salida, el nombre se ha actualizado.

5. Quinto, verifique el contenido de la tabla **`audita_agente`**:

```
SELECT * FROM audita_agente;
```

```
 id | identificador |        nombre          |         cambio_en
----+---------------+------------------------+----------------------------
  1 |           311 | Pedro Fernández Arias | 2022-12-11 20:23:07.114626
(1 fila)
```

El disparador registró el cambio en la tabla **`audita_agente`**.

Borrar un disparador.

Para eliminar un disparador de una tabla, utilice la sentencia **`DROP TRIGGER`** con la siguiente sintaxis:

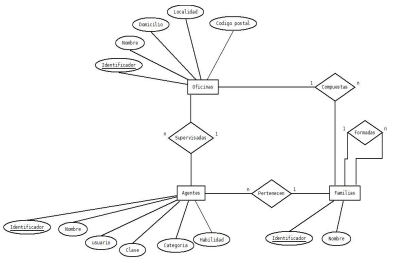
```
DROP TRIGGER [IF EXISTS] nombre_disparador
ON nombre_tabla [ CASCADE | RESTRICT ];
```

Anexo I.- Caso de estudio.

Una empresa de telefonía tiene sus centros de llamadas distribuidos por la geografía española en diferentes oficinas. Estas oficinas están jerarquizadas en familias de agentes telefónicos. Cada familia, por tanto, podrá contener agentes u otras familias. Los agentes telefónicos, según su categoría, además se encargarán de supervisar el trabajo de todos los agentes de una oficina o de coordinar el trabajo de los agentes de una familia dada. El único agente que pertenecerá directamente a una oficina y que no formará parte de ninguna familia será el supervisor de dicha oficina, cuya categoría es la 2. Los coordinadores de las familias deben pertenecer a dicha familia y su categoría será 1 (no todas las familias tienen por qué tener un coordinador y dependerá del tamaño de la oficina, ya que de ese trabajo también se puede encargar el supervisor de la oficina). Los demás agentes deberán pertenecer a una familia, su categoría será 0 y serán los que principalmente se ocupen de atender las llamadas.

- De los agentes queremos conocer su nombre, su clave y contraseña para entrar al sistema, su categoría y su habilidad que será un número entre 0 y 9 indicando su habilidad para atender llamadas.
- Para las familias sólo nos interesa conocer su nombre.
- Finalmente, para las oficinas queremos saber su nombre, domicilio, localidad y código postal de la misma.

Un posible modelo entidad-relación para el problema expuesto podría ser el siguiente:



Ministerio de Educación (Uso educativo nc)

El **Modelo Relacional** resultante sería:

OFICINAS (identificador, nombre, domicilio, localidad, codigo_postal)

FAMILIAS (identificador, nombre, familia (fk), oficina (fk))

AGENTES (identificador, nombre, usuario, clave, habilidad, categoría, familia (fk), oficina (fk))

De este modelo de datos surgen tres tablas, que puedes crear en Oracle con el script del siguiente enlace

Script CreaCasoEstudio.zip (zip - 1,62 KB)

Para ejecutar el script, es conveniente crear la base de datos. A continuación se muestran los comandos necesarios para crear la base de datos y ejecutar el script desde el terminal:

```
su postgres -- (contraseña postgres)
psql
CREATE DATABASE telefonica OWNER alumno;
exit -- salimos de postgresql
exit -- salimos de la cuenta del usuario postgres
psql telefonica
```

copiar y pegar el script

```
[alumno@vm1:~]$ su postgres
Contraseña:
postgres@vm1:/home/alumno$ psql
could not change directory to "/home/alumno": Permiso denegado
psql (14.5 (Debian 14.5-1.pgdg110+1))
Type "help" for help.

postgres=# CREATE DATABASE telefonica OWNER alumno;
CREATE DATABASE
postgres=# exit
postgres@vm1:/home/alumno$ exit
exit
[alumno@vm1:~]$ psql telefonica
psql (14.5 (Debian 14.5-1.pgdg110+1))
Digite «help» para obtener ayuda.

telefonica=> create table oficinas (
     identificador     numeric(6) not null primary key,
     nombre          varchar(40) not null unique,
     domicilio     varchar(40),
     localidad     varchar(20),
     codigo_postal      varchar(5)
);

create table familias (
     identificador     numeric(6) not null primary key,
     nombre          varchar(40) not null unique,
     familia          numeric(6) references familias,
     oficina          numeric(6) references oficinas
);
```