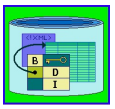


SQL 3 (Tercera Parte)

Introducción

La necesidad de utilizar un CURSOR evidencia la impedancia existente entre los lenguajes declarativos (SQL) y los lenguajes procedurales (PSM o algún lenguaje como C o Java que embeban sentencias SQL). Los lenguajes procedurales sólo saben acceder de a un registro o tupla por vez, por eso se requiere de un CURSOR que permita recorrer el *result set* de a un registro por vez.

Notar que recién ahora que introducimos un tratamiento procedural en base de datos, aparece la necesidad de usar el concepto de CURSOR.

1.5.6 Cursor

Un cursor es un concepto fundamental en base de datos. Cuando ejecutamos un SELECT obtenemos un conjunto de tuplas que verifican la condición solicitada, o sea un *result set*. Si trabajamos desde un lenguaje procedural, sólo podremos acceder de a una tupla por vez, por eso se requiere de un CURSOR, que es como un iterador del *result set* de a una por vez. Así, se "navega" en la colección resultante de elementos por medio del cursor.

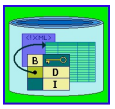
Los cursores pueden ser ***implícitos*** o ***explícitos***. Los ***implícitos*** sólo pueden usarse cuando se sabe de antemano que la consulta devolverá una sola tupla. Los ***explícitos*** deben usarse cuando se sabe que la consulta puede devolver más de una tupla. Requiere de cuatro pasos: primero debe declararse (en la zona de declaración de datos), luego abrirse explícitamente, iterarse por medio de la sentencia FETCH con lo cual se refiere a la próxima tupla del *resultset* (típicamente dentro de un ciclo) y finalmente debe cerrarse explícitamente para liberar recursos (lockeos).

En ANSI, si con **cursor implícito o explícito** no se obtuvieran tuplas (resultset vacío) se devuelve *condición,* con lo cual se la puede atrapar o no. En el segundo caso, sigue con la próxima sentencia. Pero si estamos dentro de un ciclo (el caso típico de cursor explícito) y no cortamos el ciclo cuando se acaba el cursor, quedaríamos en un ciclo infinito. ¡Cuidado!

En PostgreSQL, **un cursor implícito** que devuelva cero tuplas (result set vacío) no devuelve *condición (ni excepción)*, sino que coloca en true a la variable **NOT FOUND,** asociada al cursor y continúa ejecutando la siguiente sentencia. O sea, es diferente al ANSI. Para saber si realmente no se obtuvieron tuplas conviene consultar este valor, porque caso contrario, el valor de los atributos recuperados estarán en null y no podremos diferenciar (sólo chequeando por NULL) cuál de estos dos casos ocurrió realmente: el result set fue vacío, o bien había tuplas pero los valores del atributo consultado estaba en null.

En PostgreSQL un cursor explícito que devuelva cero tuplas (result set vacío) o que ya se terminó de navegar, no devuelve tampoco *condición (ni excepción)*, sino que coloca en true a la variable **NOT FOUND,** asociada al cursor. Si bien es diferente al ANSI, la idea es la misma: hay que salir del ciclo que contiene a la navegación del cursor. Para "cortar la ejecución del ciclo", dado que no se recibe *condición/excepción* (o sea, no sirve declarar manejador para este caso) hay que chequear esa variable NOT FOUND.

Usar un SELECT en formato PROCEDURAL (embeberlo en un PSM) implica "recuperar los valores seleccionados" en variables del PSM. Por eso se utiliza el formato SELECT INTO o FETCH INTO, según se trate de cursor implícito o explícito, respectivamente.



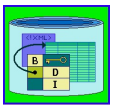
Cursor Implícito	
ANSI	**PostgreSQL**

DECLARE maximo FLOAT;
 	
BEGIN
SELECT max(sueldo) INTO maximo
FROM empleado;
END;
 	
O bien
 	
DECLARE maximo FLOAT;
 	
BEGIN
SET maximo= (SELECT max(sueldo)
FROM empleado);
END;
	👎
	DECLARE maximo FLOAT;
	
	BEGIN
	SELECT max(sueldo) INTO maximo
	FROM empleado;
	END;
	
	O bien
	
	DECLARE maximo FLOAT;
	
	BEGIN
	SELECT INTO MAXIMO max(sueldo)
	FROM empleado;
	END;

Cursor Explícito	
	PostgreSQL

DECLARE
myCursor CURSOR FOR
SELECT sueldo FROM empleado;
 	
-- manejo de la salida del ciclo en el cursor
DECLARE salir INT default 0;
DECLARE CONTINUE HANDLER FOR
NOT FOUND SET salir= 1;
 	
BEGIN
…
OPEN myCursor;
 	
A: LOOP
FETCH myCursor INTO listaVariables;
 	
IF salir = 1 THEN
LEAVE A;
END IF;
 	
END LOOP;
 	
CLOSE myCursor;
END;
	👎 (levemente diferente)
	DECLARE
	myCursor CURSOR FOR
	SELECT sueldo FROM empleado;
	
	BEGIN
	
	…
	OPEN myCursor;
	
	LOOP
	FETCH myCursor INTO listavVariables;
	
	EXIT WHEN NOT FOUND;
	
	END LOOP;
	
	CLOSE myCursor;
	END;

Si el cursor no es usado sólo para navegar, sino también para modificar o borrar la tupla subyacente a medida que se lo navega, hay que declararlo **FOR UPDATE:**



Cursor Modificable (sólo explícito)	
ANSI	**PostgreSQL**

DECLARE
  myCursor CURSOR FOR
  SELECT sueldo FROM empleado
  FOR UPDATE [OF listaColumnas];
 	
  -- manejo de la salida del ciclo en el cursor
DECLARE salir INT default 0;
DECLARE CONTINUE HANDLER FOR
  NOT FOUND SET salir= 1;
 	
BEGIN
...
OPEN myCursor;
 	
A: LOOP
  FETCH myCursor INTO listaVariables;
 	
  IF salir = 1 THEN
  LEAVE A;
  END IF;
…
[DELETE FROM empleado
** WHERE CURRENT OF myCursor;]**
[UPDATE empleado SET …..
** WHERE CURRENT OF myCursor;]**
 	
END LOOP;
 	
CLOSE myCursor;
END;
	DECLARE
	  myCursor CURSOR FOR
	  SELECT sueldo FROM empleado
	  FOR UDPATE [OF listaColumnas];
	
	BEGIN
	…
	  OPEN myCursor;
	
	  LOOP
	  FETCH myCursor INTO listavVariables;
	
	  EXIT WHEN NOT FOUND;
	
	[DELETE FROM empleado
	** WHERE CURRENT OF myCursor;]**
	[UPDATE empleado SET …..
	** WHERE CURRENT OF myCursor;]**
	
	END LOOP;
	
	CLOSE myCursor;
	END;

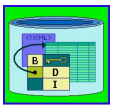
1.5.6.1 Cursor en PostgreSQL

Existe cierto tratamiento especial que vale la pena resaltar.

Muy Importante

Un cursor tiene atributos asociados que pueden usarse para averiguar información asociada al mismo:

- **FOUND**: devuelve TRUE si la última sentencia SQL, afectó una o más tuplas, y FALSE en caso contrario. Sirve para saber qué paso con el result set.
- **NOT FOUND:** al revés que el anterior. ´



- **ROW_COUNT:** devuelve la cantidad de tuplas afectadas por la última sentencia SQL. Ejemplo: la ejecución de un delete o update pueden afectar a varias tuplas.
  Para averiguar este valor hay que asignarlo en una variable:
  GET DIAGNOSTICS my_var = ROW_COUNT;

Un **cursor implícito** sólo puede ser usado para asociar a un *resultset* de tamaño 0 o 1. PostgreSQL si encuentra el uso de un cursor implícito y al ejecutar obtiene más de una tupla, sólo devuelve el valor del primer valor que encuentra en la tabla (no da error, a diferencia de otros DBMS). Pero cuidado, que ese valor es el de cualquier tupla (no asumir orden de tuplas nunca!!!).

Como dijimos previamente, cuando un **cursor explícito** navega el resultset y alcanza el fin del mismo (como cuando se intenta leer más allá de los límites de un archivo porque el mismo terminó) no lanza **condición** (ni excepción). Más aún, si la navegación del cursor estuviera contenida dentro de un ciclo, debe evaluarse explícitamente el atributo NOT FOUND para provocar el corte de la ejecución del ciclo, caso contrario se entrará en un ciclo infinito.

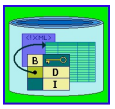
Ejemplo 1

Supongamos que se tiene el siguiente esquema:

EMPLEADO		
Legajo	Nombre	Sueldo
10	Ana	500
20	Pablo	
30	Julia	4000
40	Jorge	7000

¿Qué se obtiene al ejecutar el siguiente PSM, cuando el *resultset* resulta vacío? ¿y cuándo no resulta vacío?

```
CREATE OR REPLACE FUNCTION MAL(aLegajo empleado.legajo%TYPE) RETURNS VOID
AS $$
DECLARE
  valor  empleado.sueldo%TYPE;
BEGIN

-- cursor implicito porque legajo es PK
  SELECT sueldo INTO valor FROM empleado where legajo = aLegajo;
  Raise notice 'ENCONTRADO=%', VALOR;

  -- esta zona solo se alcanza si hay problemas
EXCEPTION
WHEN OTHERS THEN   -- cualquier exception
  IF NOT FOUND THEN
    Raise notice 'NO ENCONTRADO';
  END IF;
END;
$$ LANGUAGE plpgsql;
```



- Cuando no se encuentra en él la tupla pedida

```
DO $$
BEGIN
   PERFORM MAL(100);
END;
$$;
```

No se obtiene lo esperado, porque cuando el CURSOR IMPLICITO devuelve result set vacío no retorna excepción ni condición. La zona de excepción NO SERÁ ALCANZADA. Así, el valor del sueldo estará en NULL. El mensaje no es el correcto.

```
NOTICE:  ENCONTRADO=<NULL>
Query returned successfully with no result in 17 ms.
```

- Cuando se encuentra la tupla pedida

```
DO $$
BEGIN
   PERFORM MAL(10);
END;
$$;
```

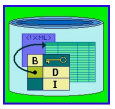
se obtiene lo esperado:

```
NOTICE:  ENCONTRADO=500
Query returned successfully with no result in 12 ms.
```

Ejemplo 2

¿Mejora si realizamos el siguiente chequeo? (notar que sacamos la zona de excepción porque no se arroja excepción NOT FOUND).

```
CREATE OR REPLACE FUNCTION Sorpresa(aLegajo empleado.legajo%TYPE) RETURNS
VOID AS $$
DECLARE
   valor  empleado.sueldo%TYPE;
BEGIN

   -- cursor implicito porque legajo es PK
   SELECT sueldo INTO valor FROM empleado where legajo = aLegajo;
   IF valor IS NULL THEN
      Raise notice 'NO ENCONTRADO';
   ELSE
      Raise notice 'ENCONTRADO=%', VALOR;
   END IF;
END;
$$ LANGUAGE plpgsql;
```



- Cuando no se encuentra en él la tupla pedida

```
DO $$
BEGIN
   PERFORM Sorpresa(100);
END;
$$;
```

Se obtiene lo esperado:

```
NOTICE:  NO ENCONTRADO
Query returned successfully with no result in 17 ms.
```

- Cuando se encuentra la tupla pedida con sueldo distinto de NULL

```
DO $$
BEGIN
   PERFORM Sorpresa(10);
END;
$$;
```

Se obtiene lo esperado:

```
NOTICE:  ENCONTRADO=500
Query returned successfully with no result in 12 ms.
```

- Cuando se encuentra la tupla pedida pero el sueldo es NULL

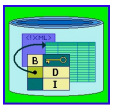
```
DO $$
BEGIN
   PERFORM Sorpresa(20);
END;
$$;
```

El mensaje obtenido es incorrecto. El problema es que si el result set es vacío devuelve los atributos en null. Pero eso también ocurre si los atributos eran null. Por lo tanto, no se puede manejar ese chequeo para diferenciar ambos casos.

```
NOTICE:  NO ENCONTRADO
Query returned successfully with no result in 17 ms.
```

Ejemplo 3

Esta es la correcta forma de chequear si el result set fue o no vacío:



```
CREATE OR REPLACE FUNCTION Bien(aLegajo empleado.legajo%TYPE) RETURNS
VOID AS $$
DECLARE
  valor  empleado.sueldo%TYPE;
BEGIN

  -- cursor implicito porque legajo es PK
  SELECT sueldo INTO valor FROM empleado where legajo = aLegajo;
  IF NOT FOUND THEN
    Raise notice 'NO ENCONTRADO';
  ELSE
    Raise notice 'ENCONTRADO=%', VALOR;
  END IF;

END;
$$ LANGUAGE plpgsql;
```

- Cuando no se encuentra en él la tupla pedida

```
DO $$
BEGIN
  PERFORM Bien(100);
END;
$$;
```

se obtiene lo esperado, porque se chequeó el valor NOT FOUND para diferenciar result set vacío o valor encontrado.

```
NOTICE:  NO ENCONTRADO
Query returned successfully with no result in 14 ms.
```

- Cuando se encuentra la tupla pedida con sueldo diferente a NULL

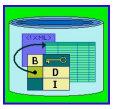
```
DO $$
BEGIN
  PERFORM Bien(10);
END;
$$;
```

se obtiene lo esperado:

```
NOTICE:  ENCONTRADO=500
Query returned successfully with no result in 12 ms.
```

- Cuando se encuentra la tupla pedida con sueldo NULL

```
DO $$
BEGIN
  PERFORM Bien(20);
END;
$$;
```



se obtiene lo esperado:

```
NOTICE:  ENCONTRADO=<NULL>
Query returned successfully with no result in 12 ms.
```

Ejemplo 4

Si se aplica una función de agregación es imposible no obtener tuplas en la respuesta (SUM, COUNT, etc). Es absurdo preguntar por NOT FOUND porque eso siempre es FALSE: si no hay tuplas en la tabla se devuelve una tupla con NULL o 0, según la función utilizada. Tampoco sirve chequear ROW_COUNT: siempre se devuelve una tupla!.

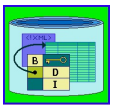
Supongamos que se quiere lanzar una excepción de usuario que indique explícitamente "SIN TUPLAS O SUELDOS NULL" para cuando no hay tuplas que satisfacen cierta condición o todos los sueldos son NULL para los que la satisfacen. Para los demás casos devolver el valor máximo de sueldo de empleado con legajo mayor que el solicitado. Acá sí hay que chequear por NULL.

Es decir, queremos transformar a una excepción donde no la había.

```
CREATE  OR  REPLACE  FUNCTION  MAX_DESDE(aLegajo  empleado.legajo%TYPE)
RETURNS empleado.sueldo%TYPE AS $$
DECLARE
  valor  empleado.sueldo%TYPE;

BEGIN

  -- cursor implicito porque la funcion de agregacion
  -- devuelve una sola tupla
  SELECT MAX(sueldo) INTO valor FROM empleado WHERE legajo > aLegajo;

  IF valor IS NULL THEN
    Raise exception 'SIN TUPLAS O SUELDOS NULL' USING ERRCODE = 'PP111';
  END IF;

  RETURN valor;
END;

$$ LANGUAGE plpgsql;
```

- Cuando no se encuentra la tupla pedida

```
DO $$
DECLARE
  rta DOUBLE PRECISION;

BEGIN
  rta:= MAX_DESDE(1000);
  raise notice 'Max sueldo para legajo > 1000 es %', rta;
EXCEPTION
WHEN SQLSTATE 'PP111' THEN
   raise notice 'Uy! % %', SQLSTATE, SQLERRM;

END;
$$;
```



El PSM lanza la excepción (termina anormalmente) y el cliente lo captura:

```
NOTICE:  Uy! PP111 SIN TUPLAS O SUELDOS NULL
Query returned successfully with no result in 16 ms.
```

- Cuando se encuentra la tupla pedida

```
DO $$
DECLARE
 rta DOUBLE PRECISION;

BEGIN
   rta:= MAX_DESDE(10);
   raise notice 'Max sueldo para legajo > 10 es %', rta;
EXCEPTION
WHEN SQLSTATE 'PP111' THEN
    raise notice 'Uy! % %', SQLSTATE, SQLERRM;

END;
$$;
```

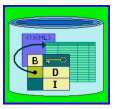
El PSM termina exitosamente y el cliente imprime el valor obtenido:

```
NOTICE:  Max sueldo para legajo > 10 es 7000
Query returned successfully with no result in 14 ms.
```

Ejemplo 5

Este código es incorrecto porque pretende usar un cursor implícito en una consulta que puede devolver más de una tupla (acá no estamos usando función de agregación).

```
CREATE OR REPLACE FUNCTION DESDE_MAL(aLegajo empleado. legajo%TYPE)
RETURNS empleado.sueldo%TYPE AS $$
DECLARE
valor  empleado.sueldo%TYPE;

BEGIN

   -- cursor implicito porque la funcion de agregacion
   -- devuelve una sola tupla
   SELECT sueldo INTO valor FROM empleado WHERE legajo > aLegajo;

   IF NOT FOUND THEN
     Raise exception 'SIN TUPLAS EN EMPLEADO' USING ERRCODE = 'PP111';
   END IF;

   RETURN valor;
END;

$$ LANGUAGE plpgsql;
```



- Cuando no se encuentra la tupla pedida, todo sale bien

```
DO $$
DECLARE
   rta DOUBLE PRECISION;

BEGIN
   rta:= DESDE_MAL(1000);
   raise notice 'Max sueldo para legajo > 10 es %', rta;
EXCEPTION
WHEN SQLSTATE 'PP111' THEN
    raise notice 'Uy! % %', SQLSTATE, SQLERRM;

END;
$$;
```

se obtiene lo esperado:

```
NOTICE:  Uy! PP111 SIN TUPLAS EN EMPLEADO
Query returned successfully with no result in 64 ms.
```

- Cuando se encuentran las tuplas pedidas

```
DO $$
DECLARE
   rta DOUBLE PRECISION;

BEGIN
   rta:= DESDE_MAL(10);
   raise notice 'Max sueldo para legajo > 10 es %', rta;
EXCEPTION
WHEN SQLSTATE 'PP111' THEN
    raise notice 'Uy! % %', SQLSTATE, SQLERRM;

END;
$$;
```

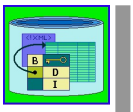
En casi todos los DBMS se obtiene una excepción al estilo:

```
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
```

Y aunque en PostgreSQL no se obtiene excepción, devuelve "cualquier tupla", lo cual tampoco sirve porque **es incorrecto**. En el ejemplo en vez de obtener todas las tuplas de legajo > 10 obtendríamos una sola tupla (cualquier tupla!):

```
NOTICE:  Max sueldo para legajo > 10 es null
Query returned successfully with no result in 14 ms.
```

Lo cual no es el valor esperado!



Ejemplo 6

La única forma es navegar por el conjunto de valores usando un **CURSOR EXPLICITO**, ya que el resultado puede devolver más de un valor.

Esta implementación **es incorrecta** porque cuando se usa un CURSOR EXPLICITO hay que salir del ciclo explícitamente, caso contrario se queda en un ciclo infinito.

```
CREATE OR REPLACE FUNCTION CICLO_INFINITO(aLegajo empleado.legajo%TYPE)
RETURNS VOID AS $$
DECLARE
  valor  empleado.sueldo%TYPE;
  myCursor CURSOR FOR
         SELECT sueldo FROM empleado WHERE legajo > aLegajo;

BEGIN

  OPEN myCursor;

  LOOP
    FETCH myCursor INTO valor;
    raise NOTICE 'sueldo %', valor;
  END LOOP;

  CLOSE myCursor;
END;

$$ LANGUAGE plpgsql;
```

Ya sea con *resultset* vacío o no vacío, el ciclo infinito se genera.

Si se invocara con:

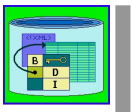
```
DO $$
DECLARE
  rta DOUBLE PRECISION;
BEGIN
  PERFORM CICLO_INFINITO(10);
END;
$$;
```

se obtendría

```
NOTICE:  sueldo 4000
NOTICE:  sueldo 7000
NOTICE:  sueldo <NULL>
NOTICE:  sueldo <NULL>
…
```

Ejemplo 7

La única forma correcta de navegar un conjunto de valores es usando un **CURSOR EXPLÏCITO** y controlando el momento en que hay que dejarlo (con EXIT).



```
CREATE OR REPLACE FUNCTION CICLO_FINITO(aLegajo empleado.legajo%TYPE)
RETURNS VOID AS $$
DECLARE
      valor  empleado.sueldo%TYPE;
      myCursor CURSOR FOR
          SELECT sueldo FROM empleado WHERE legajo > aLegajo;

BEGIN
  OPEN myCursor;

  LOOP

      FETCH myCursor INTO valor;
      EXIT WHEN NOT FOUND;

      raise NOTICE 'sueldo %', valor;

  END LOOP;

  CLOSE myCursor;
END;
$$ LANGUAGE plpgsql;
```

- Cuando no se encuentra la tupla pedida (*resultset* vacío)

```
DO $$
DECLARE
  rta DOUBLE PRECISION;
BEGIN
  PERFORM CICLO_FINITO(1000);
END;
$$;
```

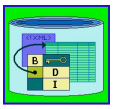
no se obtiene salida. No hay excepciones lanzadas.

- Cuando se encuentra alguna tupla, el *resultset* no es vacío, pero luego de la navegación se encontrará el fin del conjunto. Sin embargo, no habrá ciclo infinito porque se manejó explícitamente:

```
DO $$
DECLARE
  rta DOUBLE PRECISION;
BEGIN
  PERFORM CICLO_FINITO(10);
END;
$$;
```

se obtiene:

```
NOTICE:  sueldo 4000
NOTICE:  sueldo <NULL>
NOTICE:  sueldo 7000
Query returned successfully with no result in 84 ms.
```



Aclaración

Si bien otra forma de salir del loop sería chequear por algún atributo que alcanza el valor null, inmediatamente luego del fetch:

FETCH myCursor INTO valor;
EXIT WHEN valor IS NULL;

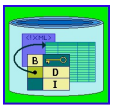
No se aconseja porque no todos los motores devuelven NULL para indicar fin del resultset. Mucho más portable es usar "NOT FOUND"

Ejemplo 8

Si quisiéramos listar cada legajo y sueldo de empleado acompañado por un cartel que indique si el sueldo es bajo (< 6000) o normal (>= 6000), y ordenado descendentemente por sueldo, donde aquel que tiene sueldo NULL no deber aparecer en el resultado, podríamos hacerlo así:

```
CREATE OR REPLACE FUNCTION CLASIFICADOR() RETURNS VOID AS $$
DECLARE
  aLegajo  empleado.legajo%TYPE;
  aSueldo  empleado.sueldo%TYPE;
  aCartel  VARCHAR(10);

  myCursor  CURSOR FOR
    SELECT legajo, sueldo, CASE WHEN sueldo < 600 THEN 'Bajo'
                                ELSE 'Normal'
                           END AS cartel
    FROM empleado
         WHERE sueldo IS NOT NULL
       ORDER BY sueldo DESC;

BEGIN

  OPEN myCursor;

  raise notice 'Legajo - Sueldo - Cartel';

  LOOP

    FETCH myCursor INTO aLegajo, aSueldo, aCartel;
    EXIT WHEN NOT FOUND;

    raise NOTICE '% % %', aLegajo, aSueldo, aCartel;

  END LOOP;

  CLOSE myCursor;
END;

$$ LANGUAGE plpgsql;
```

El ejecutarlo:



```
DO $$
BEGIN
   PERFORM CLASIFICADOR();
END;
$$;
```

Se obtiene:

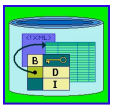
```
NOTICE:  Legajo - Sueldo - Cartel
NOTICE:  40 7000 Normal
NOTICE:  30 4000 Normal
NOTICE:  10 500 Bajo
Query returned successfully with no result in 13 ms.
```

Ejemplo 9

Como se observa, si hay que recuperar varios atributos de una tupla, resulta más sencillo si se pueden declarar en un solo paso. Sin embargo, en este caso usar el tipo "empleado%ROWTYPE" no sirve, porque el resultado del cursor no se corresponde con las columnas de la tabla. **El ROWTYPE en PostgreSQL sólo se puede asociar a tablas y vistas.**

Para declarar "un tipo compuesto (como un struct)" que nos permita levantar todos los valores de un select deseado (binding dinámico) y luego tomar cada una de sus componentes se debe declarar una variable de tipo RECORD (a secas).

```
CREATE OR REPLACE FUNCTION CLASIFICADOR() RETURNS VOID AS $$
DECLARE
   aRec  RECORD;

   myCursor  CURSOR FOR
      SELECT legajo, sueldo, CASE WHEN sueldo < 600 THEN 'Bajo'
                                  ELSE 'Normal'
                             END AS cartel
      FROM empleado
      WHERE sueldo IS NOT NULL
      ORDER BY sueldo DESC;

BEGIN
   OPEN myCursor;

   raise notice 'Legajo - Sueldo - Cartel';
   LOOP

      FETCH myCursor INTO aRec;
      EXIT WHEN NOT FOUND;

      raise NOTICE '% % %', aRec.legajo, aRec.sueldo, aRec.cartel;

   END LOOP;

   CLOSE myCursor;
END;
$$ LANGUAGE plpgsql;
```



Obteniéndose la misma salida anterior.

Notar que es muy importante poner un ALIAS al atributo "cartel" que no es de la tabla, ya que sino, no se lo puede referenciar desde el RECORD.

1.6 Algunos Ejercicios

1.6.1

Escribir una función PostgreSQL que reciba un timestamp YYYY-MM-DD HH:MM:SS y se quede sólo con la parte fecha en un formato dd-mm-yyyy, siendo el símbolo separador proporcionado por el usuario.

Ejemplo, si se invoca

```
select myfecha(fecha, '%') from auditoria
```

de obtendría algo así:

```
"23%04%2005"
"14%05%2016"
"15%05%2016"
…
```

Rta:

```
CREATE OR REPLACE FUNCTION myfecha (aFecha timestamp, letra char)
RETURNS CHAR(10) AS $$
BEGIN
  RETURN  substr(cast(aFecha as char(10)), 9, 2) || letra
       || substr(cast(aFecha as char(10)), 6, 2) || letra
       || substr(cast(aFecha as char(10)), 1, 4);

END;
$$  LANGUAGE plpgsql ;
```

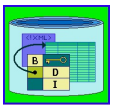
1.6.2

¿Por qué el siguiente fragmento de **código** es erróneo?

```
...
precio    PRODUCTO.precio%TYPE;
BEGIN
precio:= 12.34;
...
DELETE FROM producto WHERE precio = precio;
```

Rta:

En realidad, borra todas las tuplas de la tabla PRODUCTO y no las que poseen precio 12.34. Lo correcto sería:



```
...
   myPrecio    PRODUCTO.precio%TYPE;
BEGIN
   myPrecio:= 12.34;
   ...
   DELETE FROM producto WHERE precio = myPrecio;
```

No usar como nombre de variables los mismos nombres que en las columnas de las tablas porque estas últimas tienen prioridad. Agregarles a las variables algún prefijo o sufijo.

1.6.3.

Sea la tabla EMPLEADO con las siguientes tuplas:

EMPLEADO		
Legajo	Nombre	Sueldo
10	Ana	500
20	Pablo	null
30	Julia	4000
40	Jorge	7000

Escribir una función PSM devuelva el sueldo del empleado con legajo 30.

Rta:

Como legajo es clave en empleado podemos usar cursor IMPLICITO ya que sabemos que devuelve una sola tupla.

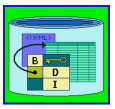
```
CREATE OR REPLACE FUNCTION Sueldo()  RETURNS FLOAT AS $$
DECLARE
  aSueldo empleado.sueldo%TYPE;
BEGIN
   SELECT sueldo INTO aSueldo FROM empleado WHERE legajo = 30;
   RETURN aSueldo;
END;
$$  LANGUAGE plpgsql ;
```

Y al ejecutar:

```
DO $$
DECLARE
   rta FLOAT;
BEGIN
   rta:= Sueldo();
raise notice '%', rta;
END;
$$;
```

se obtiene lo esperado:

```
NOTICE:  4000
Query returned successfully with no result in 54 ms.
```



1.6.4

Escribir una función PSM que devuelva el sueldo del empleado con legajo parametrizado. Si el legajo no existe lanzar excepción indicando 'Legajo inexistente'. Si el sueldo es NULL, devolverlo tal cual se lo recupera.

Rta:

Como legajo es clave en empleado podemos usar cursor IMPLICITO ya que sabemos que devuelve una sola tupla.

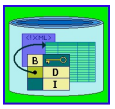
```
CREATE OR REPLACE FUNCTION Sueldo(aLegajo empleado.legajo%TYPE) RETURNS
FLOAT AS $$
DECLARE
   aSueldo empleado.sueldo%TYPE;
BEGIN
   SELECT sueldo INTO aSueldo FROM empleado WHERE legajo = aLegajo;
   IF NOT FOUND THEN
      raise exception 'SIN TUPLAS EN EMPLEADO' USING ERRCODE = 'PP111';
  ELSE
     return aSueldo;
   END IF;
END;
$$ LANGUAGE plpgsql;
```

Si lo invocamos con parámetro actual 30,

```
DO $$
DECLARE
   rta empleado.sueldo%TYPE;
BEGIN
   rta:= Sueldo(30);
   raise notice '%', rta;
EXCEPTION
   WHEN SQLSTATE 'PP111' THEN
      raise notice 'Uy! % %', SQLSTATE, SQLERRM;
END;
$$;
```

Obtenemos lo esperado:

```
NOTICE:  4000
Query returned successfully with no result in 13 ms.
```



Si lo invocamos con parámetro actual 20,

```
DO $$
DECLARE
   rta empleado.sueldo%TYPE;
BEGIN
   rta:= Sueldo(20);
   raise notice '%', rta;
EXCEPTION
   WHEN SQLSTATE 'PP111' THEN
      raise notice 'Uy! % %', SQLSTATE, SQLERRM;
END;
$$;
```

Obtenemos lo esperado:

```
NOTICE:  <NULL>
Query returned successfully with no result in 13 ms.
```

Si lo invocamos con parámetro actual inexistente:

```
DO $$
DECLARE
   rta empleado.sueldo%TYPE;
BEGIN
   rta:= Sueldo(1000);
   raise notice '%', rta;
EXCEPTION
   WHEN SQLSTATE 'PP111' THEN
      raise notice 'Uy! % %', SQLSTATE, SQLERRM;
END;
$$;
```

Obtenemos lo esperado, el cliente captura la excepción recibida y terminar normalmente:

```
NOTICE:  Uy! PP111 SIN TUPLAS EN EMPLEADO
Query returned successfully with no result in 14 ms.
```

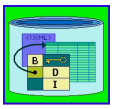
1.6.5

¿Qué se obtiene si invocamos la implementación anterior con el siguiente código?

```
select sueldo(legajo) from empleado order by legajo
```

Rta

```
500
null
4000
7000
```



1.6.6

Escribir una función PSM que devuelve la suma de los sueldos correspondientes a los legajos solicitados. No debe lanzar excepción. Si un empleado no existe o su sueldo es null debe devolver el sueldo del otro empleado. Es decir, sólo devolverá null si ambos empleados no existen, ambos empleados tienen sueldo null o combinación de esto.

Rta:

- **Opción A:**

  ¿Notar que NULL + cte es NULL por eso hay que testear antes de aplicar "+" si alguno de ellos vale NULL. Además, si la primera sentencia da NOT DATA FOUND, no devuelve excepción alguna y sigue ejecutando la próxima instrucción. O sea, estamos tranquilos que se calculan los 2 SELECTS.

Notar que el siguiente código es incorrecto porque si alguno de los legajos no existe (o el sueldo de ese legajo es NULL), la respuesta es NULL porque NULL + algo es NULL.

```
-- versión incorrecta

CREATE OR REPLACE FUNCTION combina(aLegajo1 empleado.legajo%TYPE,
aLegajo2 empleado.legajo%TYPE)
  RETURNS empleado.sueldo%TYPE AS
$$
DECLARE
   sueldo1 empleado.sueldo%TYPE;
   sueldo2 empleado.sueldo%TYPE;

BEGIN
   SELECT sueldo INTO sueldo1 FROM empleado WHERE legajo = aLegajo1;
   SELECT sueldo INTO sueldo2 FROM empleado WHERE legajo = aLegajo2;
   return sueldo1 + sueldo2;

END;

$$
LANGUAGE plpgsql ;
```

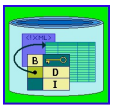
Al invocar con:

```
DO $$
BEGIN
   raise notice 'valor retornado %', combina(30, 20);
END;
$$;
```

El empleado con legajo 20 tiene sueldo null y el de legajo 30 tiene sueldo 4000. Como uno es null debe devolverse el otro, es decir 4000.
Sin embargo, el código es incorrecto y se obtendría NULL.

Análogamente el mismo problema si se invocara con:

```
DO $$
BEGIN
   raise notice 'valor retornado %', combina(30, 1000);
```



```
END;
$$;
```

Esta versión es correcta:

```
CREATE OR REPLACE FUNCTION combina(aLegajo1 empleado.legajo%TYPE,
aLegajo2 empleado.legajo%TYPE) RETURNS empleado.sueldo%TYPE AS
$$
DECLARE
   sueldo1 empleado.sueldo%TYPE;
   sueldo2 empleado.sueldo%TYPE;

BEGIN
SELECT sueldo INTO sueldo1 FROM empleado WHERE legajo = aLegajo1;
SELECT sueldo INTO sueldo2 FROM empleado WHERE legajo = aLegajo2;

IF sueldo1 IS NULL
    THEN RETURN sueldo2;
ELSEIF sueldo2 IS NULL
     THEN RETURN sueldo1;
       END IF;
RETURN sueldo1 + sueldo2;

END;

$$
LANGUAGE plpgsql ;
```

Como acá el tratamiento de empleado no existente y sueldo null es análogo, nos conviene tomar ventaja de eso.

Al invocar con:

```
DO $$
BEGIN
   raise notice 'valor retornado %', combina(30, 20);
END;
$$;
```

Obtenemos lo esperado:

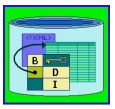
```
NOTICE:  valor retornado 4000
Query returned successfully with no result in 14 ms.
```

Si lo invocamos con:

```
DO $$
BEGIN
   raise notice 'valor retornado %', combina(30, 1000);
END;
$$;
```

Obtenemos lo esperado:

```
NOTICE:  valor retornado 4000
Query returned successfully with no result in 14 ms.
```



Si lo invocamos con:

```
DO $$
BEGIN
  raise notice 'valor retornado %', combina(100000, 300000);
END;
$$;
```

Obtenemos lo esperado:

```
NOTICE:  valor retornado NULL
Query returned successfully with no result in 14 ms.
```

Si lo invocamos con:

```
DO $$
BEGIN
  raise notice 'valor retornado %', combina(10, 30);
END;
$$;
```

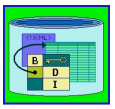
Obtenemos lo esperado:

```
NOTICE:  valor retornado 4500
Query returned successfully with no result in 14 ms.
```

- **Opción B:**

Aprovechando que la función de agregación SUM trabaja con NULL de la forma en que queremos, con este código obtenemos el mismo resultado que el anterior.

```
CREATE OR REPLACE FUNCTION combina(aLegajo1 empleado.legajo%TYPE,
aLegajo2 empleado.legajo%TYPE) RETURNS empleado.sueldo%TYPE AS
$$
DECLARE
  aSueldo empleado.sueldo%TYPE;
 BEGIN
  SELECT SUM(sueldo) INTO aSueldo
  FROM empleado
  WHERE legajo IN (aLegajo1, aLegajo2);
RETURN aSueldo;
END;
$$
LANGUAGE plpgsql ;
```



1.6.7.

Escribir una variante al anterior, donde la función PSM devuelva en un parámetro el resultado esperado.

Rta:

```
CREATE OR REPLACE FUNCTION combina(aLegajo1 empleado.legajo%TYPE,
aLegajo2 empleado.legajo%TYPE, OUT rta  empleado.sueldo%TYPE) AS
$$
BEGIN

   SELECT SUM(sueldo) INTO rta
   FROM empleado
   WHERE legajo IN (aLegajo1, aLegajo2);

END;

$$
LANGUAGE plpgsql ;
```

No lo invocamos con Perform porque esperamos un valor retornado en un PARAMETRO FORMAL. Las invocaciones son las mismas que antes.

1.6.8.

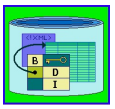
Se quiere estimar el riesgo de una compañía de aumentar los sueldos en un cierto PORCENTAJE a aquellos empleados que tengan sueldos por debajo de un UMBRAL. Para ellos se pensó realizar un PSM con parámetro UMBRAL, PORCENTAJE y que devuelva en un tercer parámetro el total del incremento que deberían contemplar en el presupuesto**.**

A los empleados que poseen sueldo null o cero, ignorarlos del cálculo.

Podríamos usar un CURSOR para navegar y analizar cada caso.

Ejemplo de invocación (significa que a aquellos empleados con sueldos inferiores a 3200 pesos incrementarían su sueldo en un 50%)

```
DO $$
DECLARE
 auxi empleado.sueldo%type;
BEGIN
   auxi:= aplica(4200, 50);
   raise notice 'el prespuesto debe incrementarse en % pesos', auxi;
END;
$$;
```



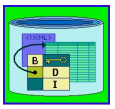
Con los datos del ejemplo se obtendría 2250 porque los sueldos que están debajo del umbral son 4000 y 500 y como se espera aumentarles un 50% el total que hay que prever en el presupuesto es de 2250.

Rta:

```
CREATE OR REPLACE FUNCTION aplica(UMBRAL empleado.sueldo%type, porcentaje
float, OUT rta  empleado.sueldo%TYPE)
  AS
$$
DECLARE
  myCursor CURSOR FOR
     SELECT sueldo
      FROM empleado;
  auxi empleado.sueldo%TYPE;
BEGIN
  OPEN myCursor;
  rta:= 0;
  LOOP
     FETCH myCursor INTO auxi;
     EXIT WHEN NOT FOUND;

     IF (auxi < umbral)  THEN
        rta:= rta + porcentaje * auxi * 0.01; -- null o 0 no cuentan
     END IF;
  END LOOP;

  CLOSE MyCursor;
END;
$$ LANGUAGE plpgsql ;
```

1.6.9

Similar al anterior, pero donde realmente se actualice el sueldo (no hace falta el tercer parámetro)



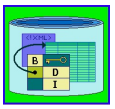
```
CREATE OR REPLACE FUNCTION aumento(UMBRAL empleado.sueldo%type,
porcentaje float) RETURNS VOID AS $$
DECLARE
  myCursor CURSOR FOR Select sueldo FROM empleado FOR UPDATE;
  auxi empleado.sueldo%TYPE;
BEGIN
  OPEN myCursor;

  LOOP
    FETCH myCursor INTO auxi;
    EXIT WHEN NOT FOUND;

    IF (auxi < umbral) THEN
      UPDATE empleado SET sueldo = auxi * (1 + porcentaje *  0.01)
      WHERE current of mycursor;
    END IF;
  END LOOP;
  close MyCursor;
END;
$$  LANGUAGE plpgsql ;
```

La invocación sería

```
DO $$
BEGIN
  PERFORM aumento(4200, 50);
END;
$$;
```

Y al consultar la tabla empleado los sueldos que satisfacen la condición quedan cambiados:

EMPLEADO		
Legajo	Nombre	Sueldo
10	Ana	750
20	Pablo	null
30	Julia	6000
40	Jorge	7000



Discusión:

Aunque estos últimos dos ejercicios suelen aparecer en lo libros … ¿Tiene sentido reemplazar un único SENCILLO SELECT o UPDATE por un PSM????

Claro que NO!!!

Se podría haber obtenido lo mismo simplemente ejecutando un SENCILLO SELECT para el primer caso:

```
SELECT sum(sueldo * 50 * 0.01)
FROM empleado
WHERE sueldo < 4200
```

y para el aumento por un SENCILLO UPDATE:

```
UPDATE empleado SET sueldo = sueldo * ( 1 + 50 * 0.01) WHERE sueldo < 4200;
```

SQL es un lenguaje MUY DECLARATIVO y potente. Sólo implementar PSMs cuando no se lo puede hacer con una única sentencia SQL, con la intención de que el agregado procedural (la secuencia, la decisión, la repetición) agregue funcionalidad.