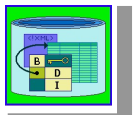


SQL 3 (Segunda Parte)

Introducción

A medida que los sistemas de bases de datos fueron invadiendo el mercado, los desarrolladores de DBMS se vieron exigidos a ofrecer mayores facilidades, muchas de ellas fuera de los estándares.

En las clases 13 y 14 analizaremos algunos cambios propuestos en SQL3 que resultan ser de gran utilidad a la hora de encarar un proyecto a gran escala.
En la clase 15 analizaremos qué posibilidades ya ofrecía SQL2 para el manejo de restricciones avanzadas y qué agrega SQL3 al respecto.

1. Módulos Persistentes Almacenados (Persistent Stored Modules, PSM o ex- Stored Procedure)

Es código compilado y almacenado en la base de datos que podrá ser ejecutado si se tiene permiso de ejecución.

Típicamente, **hay que invocarlo explícitamente** desde otro PSM (tema de esta clase), desde el bloque de algún ***trigger*** (tema de la clase 15) o desde una aplicación cliente (tema de la clase 16).

Los DBMS que soportan la escritura de ***PSM*** potencian un lenguaje ***declarativo como SQL*** con extensiones de ***lenguaje procedural*** para manejar el flujo de control.

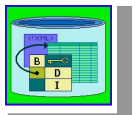
La idea de esto fue muy anterior a SQL3, y cada DBMS comercial estableció su propio lenguaje, basándose en lenguajes procedurales de su agrado (FORTRAN, C, etc). Esta falta de estandarización causó que los programadores desalentaran su uso.

Finalmente, SQL3 estableció su sintaxis y lentamente los DBMS comerciales tratarán de ser compatibles con la especificación vigente.

Claramente las sentencias dentro de un PSM son procedurales, por lo tanto, NO está permitido el SELECT tradicional, sino que se utiliza una variante del mismo: **SELECT INTO**.

SQL3 diferencia conceptualmente dos tipos de módulos: **funciones** y **procedimientos**

	CÓMO SE INVOCA	QUÉ DEVUELVE EN SU NOMBRE	CÓMO SON SUS PARÁMETROS
Función	*func_name*(params) Participa típicamente de las cláusulas SELECT donde se pueden usar funciones *built-in* del motor (Ej: LENGTH, etc).	Un valor único escalar obligatorio.	Son sólo parámetros de entrada (IN)



Procedimiento	call *proc_name*([params]) No puede participar de cláusulas SQL ya que ni siquiera devuelve un valor en su nombre.	Nada (como si fuera void).	Parámetros de entrada (IN), de salida (OUT) o de entrada/salida (INOUT). Obviamente los parámetros actuales en el caso de OUT o INOUT deben ser ***l-values***. Los parámetros de salida o entrada/salida podrán devolver un escalar o un *resultset* dinámico (resultado de selects).

Dado que las implementaciones de los ***PSM*** todavía son muy dependientes del cada DBMS, estudiaremos cómo es la especificación ANSI y la compararemos con la estrategia PostgreSQL.

Muy Importante

Cuando se usa un ***PSM*** tener precaución de que el nombre de los parámetros o variables locales NO COINCIDA con el nombre de algún atributo de tabla, porque en general esto últimos tienen prioridad.

Siempre agregarle algún prefijo o sufijo a las variables locales o parámetros formales.

1.1 Creación/Destrucción de un PSM Función

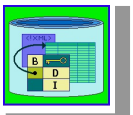
1.1.1 ANSI

Sintaxis ANSI para la Creación de un PSM Función

```
CREATE FUNCTION  nombreFuncion ( [ parametros_formales ] )
RETURNS tipo_devuelto
   LANGUAGE SQL
   [ DETERMINISTIC | NOT DETERMINISTIC]
   [ NO SQL | CONTAINS SQL | READS SQL | MODIFIES SQL DATA ]
   [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
   Sentencia SQL
```

O bien

```
CREATE FUNCTION  nombreFuncion ( [ parametros_formales ] )
RETURNS tipo_devuelto
   LANGUAGE SQL
   [ DETERMINISTIC | NOT DETERMINISTIC]
   [ NO SQL | CONTAINS SQL | READS SQL | MODIFIES SQL DATA ]
   [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
BEGIN
   Sentencias SQL;
END
```



Sintaxis ANSI para la Destrucción de un PSM Función

```
DROP FUNCTION  nombreFuncion [  (tipo, tipo, ... )  ];
```

donde la especificación de los tipos exactos de los parámetros sólo hace falta indicarla en el caso de que se haya utilizado el mismo nombre de la función para distintos grupos de argumentos.

- La lista de parámetros formales tiene la forma de: [IN] *param_name* tipo
  Como lo único que puede indicarse es IN, puede omitirse.

- Hemos mostrado la cláusula LANGUAGE SQL como obligatoria, pero en realidad se puede implementar la función con lenguajes externos como C, Java, etc.

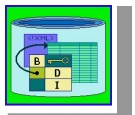
- La cláusula NOT DETERMINISTIC (default) se usa cuando el resultado obtenido dependerá de algún valor no determinable hasta el momento de la ejecución. Ejemplo: usa la función random, usa la fecha actual, usar variables de entorno, accede a archivos externos en vez de tablas de la base de datos. Se utiliza la cláusula DETERMINISTIC cuando para los mismos parámetros de entrada y valores en la base de datos se obtiene el mismo resultado en sucesivas ejecuciones. Si se declarara como DETERMINISTIC el DBMS podría decidir no reejecutar la consulta si desde la última ejecución no se produjo ni modificación de parámetros, ni de la base de datos.

- Se utiliza la cláusula NO SQL si no se invoca nada del motor (Ej: invoca una biblioteca externa para enviar mails), se usa CONTAINS SQL si contiene alguna función SQL pero no accede a la base de datos (Ej: invoca la función LENGTH), se utiliza READS SQL si contiene cláusulas SQL que sólo leen la base de datos (Ej: SELECT) o bien se usa MODIFIES SQL DATA si se invocan cláusulas SQL que modifican la base de datos (Ej: UPDATE). El *default* es CONTAINS SQL.

- Se utiliza la cláusula RETURNS NULL ON NULL INPUT cuando se desea que NO se invoque a la función si alguno de los parámetros actuales fuera NULL, prefiriendo directamente devolver NULL como resultado de la misma. Si en cambio se desea transferir el control a la función aunque algún parámetro actual fuera NULL, se utiliza CALLED ON NULL INPUT, la cual constituye la cláusula *default*.

Ejemplo 1

```
CREATE FUNCTION  ratio ( IN titulo VARCHAR(100), IN largo INT )
RETURNS REAL
   LANGUAGE SQL
   DETERMINISTIC
   CONTAINS SQL
   RETURNS NULL ON NULL INPUT
RETURN  CHAR_LENGTH(titulo)/largo
```



1.1.2 PostgreSQL

PostgreSQL no tiene PROCEDURE como el ANSI.

Sólo maneja FUNCTION. A diferencia del ANSI, una FUNCTION puede declarar parámetros IN, OUT e INOUT.

Importante

Los pasajes de parámetros en PostgreSQL son **SOLO POR VALOR**.

Caso 1)
Si no hay parámetros OUT o INOUT la función retornará el tipo deseado (Ej: RETURNS FLOAT), o bien RETURNS VOID si se quiere que se comporte como un procedimiento.

Caso 2)
¿Qué pasa si tiene parámetros INOUT o OUT?
¿Qué hace entonces cuando encuentra por lo menos un parámetro de tipo INOUT o OUT?

- Automáticamente genera un "**RETURNS tipo**" o bien "**RETURN record**".

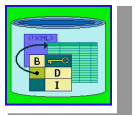
El primer caso, si sólo encuentra un parámetro formal de tipo INOUT o OUT.
El segundo caso, si encuentra más de uno y arma entonces el tipo compuesto.

- Aconsejamos OMITIR colocar explícitamente la cláusula "RETURNS …" cuando se declaran parámetros de salida (INOUT; OUT) ya que el compilador lo hará automáticamente por nosotros, y de no coincidir, dará error de compilación.

- No permite en ningún lugar del cuerpo realizar "return algo", sino que espera que se asignen directamente valores a los parámetros formales (de tipo INOUT o OUT). Luego podremos colocar "return" a secas para devolver el control cuando se quiera.

- Los valores que se asignaron a los parámetros FORMALES de tipo INOUT o OUT se pierden al retornar. O sea, el retorno se hace en el nombre de la función.

Así, NO hay problema en que las funciones sean invocadas desde sentencias SQL con sus parámetros OUT o INOUT (ya que técnicamente hablando no vuelven modificados).



Sintaxis PostgreSQL para la Creación de un PSM Función

Caso 1: Todos los parámetros son IN

```
CREATE [OR REPLACE] FUNCTION  nombreFuncion ( [ parametros_formales ] )
   RETURNS tipo_devuelto
   AS $$

   DECLARE  ...;

   BEGIN

      Sentencias SQL;

   END
   $$ LANGUAGE plpgSQL
   [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]

-- donde tipo_devuelto podría ser VOID
```

Caso 2: Existen parámetros OUT o INOUT (mejor omitir el returns …)

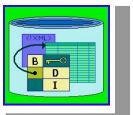
```
CREATE [OR REPLACE] FUNCTION  nombreFuncion ( [ parametros_formales ] )
   AS $$
   DECLARE ...;
   BEGIN

      Sentencias SQL;

   END
   $$ LANGUAGE plpgSQL
  [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
```

Sintaxis PostgreSQL para la Destrucción de un PSM Función

Ídem al ANSI

Las diferencias con el ANSI consisten en:

- En la lista de parámetros formales se permite el uso de IN, OUT e IN OUT. El default es IN.
- No existe la cláusula NO SQL, CONTAINS SQL, READ SQL, MODIFIES SQL DATA, DETERMINISTIC, etc. Las que ofrece **PostgreSQL son muy distintas**, por eso las omitimos.



- Aparece el uso obligatorio de la palabra

```
AS $$


$$ LANGUAGE plpgSQL
```

- Agrega una zona, previa al bloque donde se pueden declarar variables locales y constantes
- Los topes para los parámetros formales o retorno de tipo CHAR o VARCHAR son ignorados. Toman la extensión del parámetro actual.

Ejemplo 2

Esta función escalar calcula la tangente de un número. Indicamos que no queremos transferir el control cuando el parámetro es null porque queremos que directamente se devuelva null en ese caso.

```
CREATE or replace FUNCTION TAN(X double precision)
RETURNS double precision
AS $$
BEGIN
  RETURN sin(x) / cos(x);
END;
$$ LANGUAGE plpgSQL
RETURNS NULL ON NULL INPUT;
```

Devolver un valor en el nombre de la función de tipo FLOAT es análogo a declarar un parámetro OUT de tipo FLOAT y asignarle valor antes de retornar:

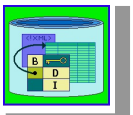
```
CREATE OR REPLACE FUNCTION TAN( IN x double precision, OUT rta double
precision)
AS $$
BEGIN
  rta:= sin(x) / cos(x);
  RETURN;
END;
$$ LANGUAGE plpgSQL
RETURNS NULL ON NULL INPUT;
```

Ya que el compilador lo cambia así:

```
CREATE OR REPLACE FUNCTION tan(IN x double precision, OUT rta double
precision) RETURNS double precision
AS $$
…
```

También se podría obtener el mismo resultado declarando el parámetro de tipo INOUT.

```
CREATE OR REPLACE FUNCTION TAN( INOUT x double precision)
AS $$
BEGIN
  x:= sin(x) / cos(x);
  RETURN;
END;
$$ LANGUAGE plpgSQL
RETURNS NULL ON NULL INPUT;
```



Y el compilador lo cambia así:

```
CREATE OR REPLACE FUNCTION tan(INOUT x double precision)
  RETURNS double precision
AS $$
…
```

Ejemplo 3

Esta función escalar genera un string reversa de uno dado. Indicamos que no queremos transferir el control cuando el parámetro es null porque queremos que directamente se devuelva null en ese caso.

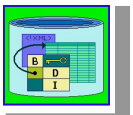
```
CREATE OR REPLACE FUNCTION REVERSE2(IN cartel VARCHAR) RETURNS VARCHAR
AS $$
DECLARE
  rta VARCHAR(100) DEFAULT '';
  len INT;

BEGIN
  len:= LENGTH( cartel);
  WHILE ( len > 0 ) LOOP
    rta:= rta || substr( cartel, len, 1);
    len := len - 1;
  END LOOP;

  RETURN rta;
END;
$$ LANGUAGE plpgSQL
RETURNS NULL ON NULL INPUT;
```

El compilador lo cambia así:

```
CREATE OR REPLACE FUNCTION reverse2(cartel character varying)
  RETURNS character varying
AS $$
...
```

Devolver un valor en el nombre de la función de tipo VARCHAR es análogo a declarar un parámetro OUT de ese tipo y asignarle valor antes de retornar:

```
CREATE OR REPLACE FUNCTION REVERSE2(IN cartel VARCHAR, OUT rta VARCHAR )
AS $$
DECLARE
  len INT;

BEGIN
  len:= LENGTH( cartel);
  rta:= '';

  WHILE ( len > 0 ) LOOP
    rta:= rta || substr( cartel, len, 1);
    len := len - 1;
  END LOOP;

END;
$$ LANGUAGE plpgSQL
RETURNS NULL ON NULL INPUT;
```



El compilador lo cambia así:

```
CREATE OR REPLACE FUNCTION reverse2(IN cartel character varying, OUT rta
character varying)
  RETURNS character varying
AS $$
…
```

También se podría obtener el mismo resultado declarando el parámetro de tipo INOUT.

```
CREATE OR REPLACE FUNCTION REVERSE2(INOUT cartel VARCHAR )
AS $$
DECLARE
   len INT;
   rta VARCHAR(100) default '';

BEGIN
   len:= LENGTH( cartel);

   WHILE ( len > 0 ) LOOP
      rta:= rta || substr( cartel, len, 1);
      len := len - 1;
   END LOOP;

   cartel:= rta;

END;
$$ LANGUAGE plpgSQL
RETURNS NULL ON NULL INPUT;
```

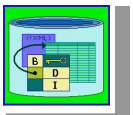
El compilador lo cambia así:

```
CREATE OR REPLACE FUNCTION reverse2(INOUT cartel character varying)
  RETURNS character varying
AS $$
…
```

Ejemplo 4

Esta función escalar devuelve el MCD entre 2 números enteros.

```
CREATE OR REPLACE FUNCTION mcd(dividendo IN integer, divisor IN integer)
RETURNS integer
AS $$

BEGIN

   WHILE (divisor > 0) LOOP
      DECLARE auxi INT;
      BEGIN
         auxi:= dividendo % divisor;
         dividendo:= divisor;
         divisor:= auxi;
      END;
   END LOOP;
   RETURN dividendo;
END;
$$ LANGUAGE plpgsql;
```



Y también podría definirse con parámetro INOUT o OUT, como discutimos

Ejemplo 5

La siguiente función simplifica una fracción, dada por su numerador y denominador, invocando la función MCD.

La idea es que recibe 2 parámetros que representa el numerador y denominador de la fracción y devuelve una fracción simplificada. Dado que PostgreSQL tiene pasaje por valor, no importa si se declaran INOUT o OUT, siempre construye un nuevo tipo para devolver. A diferencia de los anteriores ejemplos, ahora tenemos 2 parámetros de salida con lo cual construirá un RECORD:

```
CREATE OR REPLACE FUNCTION simplifiedfraction( INOUT numerator integer,
    INOUT denominator integer)
AS  $$

DECLARE
  auxi INT;

BEGIN
  auxi:= MCD(numerator, denominator);

  numerator:= numerator / auxi;
  denominator := denominator / auxi;
END;
$$   LANGUAGE plpgsql;
```

El compilador generará el siguiente código, el cual devuelve un RECORD para encapsular los 2 valores que debemos retornar:

```
CREATE OR REPLACE FUNCTION simplifiedfraction(
    INOUT numerator integer,
    INOUT denominator integer)
  RETURNS record
AS $$
…
```

1.2.1. Invocación de un PSM Función

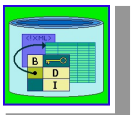
1.2.1 *ANSI*

Sintaxis ANSI para la invocación de un PSM Función

nombreFuncion **([** ***parámetros_formales*** **])**

Ejemplo 6

```
SELECT ratio( nombre, 10) FROM empleado
```



1.2.2 *PostgreSQL*

Sintaxis PostgreSQL

1) para la invocación de un PSM Función que devuelve un valor ya sea en su nombre o en parámetros formales OUT o INOUT

Ídem al ANSI

2) para la invocación de un PSM Función de tipo VOID que no devuelve nada ni siquiera en parámetros!!!

PERFORM nombreFuncionVoid(parámetros actuales)

Ejemplo 7

Invocamos la función escalar que calcula el string *reverse2*. Lo aplicamos a la tabla empleado

```
SELECT reverse2( nombre )
FROM empleado;
```

Ejemplo 8

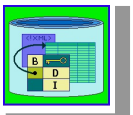
Invocamos la función escalar *reverse2* para detectar palíndromos

```
SELECT *
FROM empleado
WHERE reverse2( nombre ) = nombre;
```

Ejemplo 9

Invocamos la función escalar *SimplifiedFraction* para simplificar una fracción

```
SELECT simplifiedfraction(20, 4)
FROM empleado;

Si la tabla empleado tuviera una sola tupla obtendríamos el resultado:

Simplifiedfraction record
(5, 1)
```



1.3 Creación/Destrucción de un PSM Procedimiento

1.3.1 ANSI

Sintaxis ANSI para la Creación de un PSM procedimiento

```
CREATE PROCEDURE  nombreProcedimiento ( [ parametros_formales ] )
   LANGUAGE SQL
   [ DETERMINISTIC | NOT DETERMINISTIC]
   [ NO SQL | CONTAINS SQL | READS SQL | MODIFIES SQL DATA ]
   [ DYNAMIC RESULT SETS ]
   Sentencia SQL
```

O bien

```
CREATE PROCEDURE  nombreProcedimiento ( [ parametros_formales ] )
   LANGUAGE SQL
   [ DETERMINISTIC | NOT DETERMINISTIC]
   [ NO SQL | CONTAINS SQL | READS SQL | MODIFIES SQL DATA ]
   [ DYNAMIC RESULT SETS ]
BEGIN
   Sentencias SQL;
END
```

Sintaxis ANSI para la Destrucción de un PSM procedimiento

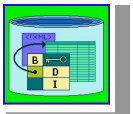
```
DROP PROCEDURE  nombreProcedimiento [  (tipo, tipo, ... )  ];
```

donde la especificación de los tipos exactos de los parámetros sólo hace falta indicar en el caso de que se haya utilizado el mismo nombre de procedimiento para distintos grupos de argumentos.

- La lista de parámetros formales tiene la forma de: [IN | OUT | INOUT] *param_name* tipo. El *default* es IN.
- Hemos mostrado la cláusula LANGUAGE SQL como obligatoria, pero en realidad se puede implementar la función con lenguajes externos como C, Java, etc. La implementación con rutinas externas la veremos más adelante.
- Las cláusulas DETERMINISTIC/NOT DETERMINISTIC y NO SQL/CONTAINS SQL/READS SQL/MODIFIES SQL DATA tienen el mismo significado que para funciones.
- La cláusula DYNAMIC RESULT SETS se usa sólo cuando se devuelve un *resultset dinámico.* Se explicará en la sección 2.

Ejemplo 10

```
CREATE PROCEDURE ratio (IN titulo VARCHAR(100),
                        IN largo INT, OUT rta REAL )
   LANGUAGE SQL
   DETERMINISTIC
   CONTAINS SQL
SET  rta= CHAR_LENGTH(titulo)/largo
```



1.3.2 PostgreSQL

Como dijimos previamente, PostgreSQL no tiene PROCEDURE. Sólo se representa como una FUNCTION que devuelve VOID y no tiene parámetros OUT o INOUT.

1.3 Invocación de un PSM Procedimiento

1.4.1 ANSI

Sintaxis ANSI para la invocación de un PSM Procedimiento

```
CALL nombreProc ( [ parámetros_formales ] )
```

Ejemplo 11

```
CALL  RATIO('HOLA', 2)
```

1.4.2 PostgreSQL

No tiene. ***(pero si se usa una función que devuelve void con parámetros sólo IN se lo invoca con PERFORM. Técnicamente hablando no es más que una función especial).***

1.5 Sentencias para el Cuerpo de un PSM (función o procedimiento)

En este lenguaje neutral (mezcla de varios) aparece: la declaración de parámetros previamente discutida (IN, OUT, INOUT), forma de retornar valores (si se trata de una función), bloques con su declaración de variables locales, constantes y cursores, sentencias de asignación, sentencias de decisión, sentencias de repetición, especificación de manejador de excepciones.

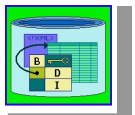
Las líneas pueden comentarse, siendo ignoradas. El comentario a nivel línea es - -
El comentario que abarca varias líneas es /* */

Analizaremos básicamente cada una de las sentencias que típicamente usamos al codificar los PSM.

1.5.1 Definición de Bloque (sentencia compuesta)

Un PSM está formado por un bloque. Un bloque comienza y termina con el par BEGIN/END.

Un bloque puede estar formado por varios bloques anidados, para definir menor alcance de variables (*scope*). Así, cada bloque resuelve un problema: estrategia conocida de DIVIDE & CONQUER.



Como se observa en el cuadro, el ANSI incluye en el bloque la declaración de constantes, variables, cursores, excepciones y obviamente las sentencias procedurales esperadas. La zona de declaración siempre debe estar al inicio del bloque.

PostgreSQL, en cambio, define una zona previa al bloque y otra posterior al mismo (justo antes de finalizarlo). En la zona previa coloca la declaración de constantes, variables y cursores. En la zona posterior coloca la zona de excepciones. En el medio coloca el bloque propiamente dicho con las sentencias procedurales esperadas. Por eso, al conjunto de estas tres zonas se le suele llamar *pseudo-bloques*.

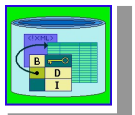
Bloque	
ANSI	**PostgreSQL**

BEGIN
[declaracion_variables_constantes]
[declaracion _cursores]
[declaracion_de_condicion_o_excepciones]
sentencias Procedurales;
END;
	
	Está formada por tres partes: declarativa, ejecutable, manejadora de excepciones
	
	[**DECLARE**
	-- declaraciones
	]
	BEGIN
	-- sentencias
	[**EXCEPTION**
	-- manejadores de excepciones
	]
	END;

1.5.2 Declaración de Variables y Constantes

Tener en cuenta que ANSI declaran esta zona a comienzo del bloque. PostgrelSQL, por el contrario, lo declara previo al bloque.

Variables y Constantes	
ANSI	**PostgreSQL**
Variables: DECLARE Vble1 tipo1 [DEFAULT valor1]; Vble2 tipo2 [DEFAULT valor2]; ... VbleN tipoN [DEFAULT valorN];	ANSI ✓ **Aclaración:** también permite asignarlo en forma no ANSI DECLARE Vble1 tipo1 [:= valor1]; O también DECLARE Vble1 tipo1 [= valor1];



Constantes: -	**Permite declarar constantes (no omitir inicializarlas en el momento de declararlas porque después no es posible y contendrán NULL)** DECLARE Cte1 CONSTANT tipo DEFAULT valor1; **O bien** DECLARE Cte1 CONSTANT tipo := valor1; **O bien** DECLARE Cte1 CONSTANT tipo = valor1;

Muy Importante

PostgreSQL para variables locales define los atributos **%TYPE** y **%ROWTYPE** que se pueden aplicar sobre una columna de una tabla o una tupla de una tabla respectivamente. El primero provee el tipo de datos de la columna sobre la cual aplica. El segundo provee el tipo de todo un registro de la tabla (cada columna dentro de dicho registro se accede igual que en los *structs* del lenguaje C).

Resulta muy conveniente declarar aquellas variables de un PSM que deseamos que se corresponda con los tipos de alguna columna o registro de cierta tabla, por medio de estos atributos. De esta manera, si se cambian los tipos de las columnas en las tablas, automáticamente cambian las declaraciones de las variables PSM en tiempo de ejecución. Con esta estrategia se reduce el costo de mantenimiento de los programas.

Las variables se definen:

```
nombre tabla.columna%TYPE
```
o bien

```
nombre tabla%ROWTYPE
```

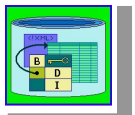
Para parámetros formal o valor retornado cambia levemente:

- Al encontrar tabla.columna%TYPE lo cambia "estáticamente" por el tipo correspondiente.

- En vez de declarar tabla%ROWTYPE hay que colocar directamente el nombre de la tabla.

Ej:

```
CREATE FUNCTION proof( rec  empleado) …

en vez de

CREATE FUNCTION proof( rec  empleado%ROWTYPE)
```



1.5.3 Asignación de valores

Asignación	
ANSI	**PostgreSQL**

`SET Vble = valor;`	👎
	`Vble := valor;`
	O bien
	`Vble = valor;`

El valor con que empiezan las variables que no han sido inicializadas es NULL.

Tenerlo en cuenta porque puede ocasionar efectos indeseables.

Ejemplo 12

```
DECLARE
 cont  INTEGER;

BEGIN
   cont:= cont + 1;
                    -- cont sigue en NULL
...
END;
```

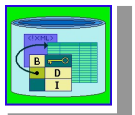
1.5.4 Sentencias Procedurales

Sentencias de Decisión	
ANSI	**PostgreSQL**

`CASE expresion`
`WHEN cte1 THEN sentencia1;`
`WHEN cte2 THEN sentencia2;`
`[ELSE sentenciaN; ]`
`END CASE;`
	ANSI ✓

`CASE`
`WHEN exp1 THEN sentencia1;`
`WHEN exp2 THEN sentencia2;`
`[ELSE sentenciaN; ]`
`END CASE;`
	ANSI ✓

`IF condicion1 THEN sentencias1;`
`[ELSEIF condicion2 THEN sentencias2;]`
`[ELSE sentenciasN; ]`
`END IF;`
	👎
	`IF condicion1 THEN sentencias1;`
	`[ELSIF condicion2 THEN sentencias2;]`
	`[ELSE sentenciasN; ]`
	`END IF;`



Sentencias de Repetición	
ANSI	**PostgreSQL**
LOOP sentencias; **END LOOP;**	ANSI ✓

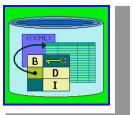
WHILE condicion **DO**
  sentencias;
END WHILE;
	👎
	WHILE condicion **LOOP**
	  sentencias;
	END LOOP;

REPEAT sentencias; **UNTIL** condicion **END REPEAT;**	👎 -
-	👎 **FOR** vble **IN [REVERSE]** limiteInf .. limiteSup [by step] **LOOP** sentencias; **END LOOP;**
Para salir de un bloque o ciclo: **LEAVE rotulo;**	👎 **(ídem con CONTINUE)** **EXIT;** O bien **EXIT WHEN condición;** O bien **EXIT rotulo WHEN condicion;**

Si quisiéramos realizar un bloque anónimo para probar que la función **SimplifiedFraction**, que definimos previamente, funciona correctamente sería útil imprimir el valor devuelto en la salida estándar.

PostgreSQL tiene una función que permite imprimir en la salida estándar, al estilo printf:
raise notice ' %', vble
donde aparecen tantos % como variables se deseen imprimir (como si fuera un printf pero sin tipos) y luego la lista de valores/variables separada por comas.

```
DO $$

DECLARE g record;
   n int := 28;
   d int := 6;

BEGIN
  g:= simplifiedfraction(n, d);
  raise notice '%', g;
  raise notice '% %', n, d;
END;

$$;
```



Al ejecutar, obtendríamos:

```
NOTICE:  (14,3)
NOTICE:  28/6
Query returned successfully with no result in 14 ms.
```

El primer valor (14, 3) muestra el record que devolvió la función invocada.

El segundo valor 28/6 muestra los valores de las variables "n" y "d", que como dijimos previamente siempre se PASAN POR VALOR, por eso no se modifican.

1.5.5 Manejo de Condiciones y Excepciones

Manejador de Condición o Excepción	
ANSI	**PostgreSQL**
El explicado en este ítem	 Es tan distinto que lo explicamos en el ítem **1.5.5.1**

La ejecución de una sentencia puede resultar o no exitosa. Es fundamental que un PSM declare un manejo para los errores que puedan producirse, generando *código robusto*. En BD, muchos son los problemas que pueden generarse: división por cero, violación de PK/FK, violación de restricción de integridad (intento de que una columna que forma parte de una clave sea null), violación de la lógica de negocios, etc.

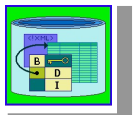
Los DBMS, luego de cada ejecución de una sentencia se comunican con el invocador por medio de una zona llamada **SQL Communication Area** (**SQLCA**) para dejar la información de cómo resultó la operación solicitada. En dicho SQLCA existen dos variables que indican el estado resultante de la ejecución: **SQLCODE y SQLSTATE**.

Ambas variables pueden ser analizadas por el programa que lanzó la sentencia SQL, para decidir qué hacer en caso de haber obtenido un error.

La variable SQLSTATE es un CHAR(5), según se la definió en ISO/ANSI SQL92. Sus primeros dos caracteres indican una primera clasificación (clase) del estado del resultado y los otros tres caracteres indican información adicional. Los primeros 2 caracteres están codificados de acuerdo a la Tabla de Equivalencia.

La variable SQLCODE es un entero.

Tabla de Equivalencia		
Estado de la operación	SQLCODE	Primeros de caracteres de SQLSTATE
OK (exitosa)	0	00
NOT FOUND	100	02
SQLWARNING	>0 && <> 100	01
SQLEXCEPTION	<0	Cualquier otra combinación



Importante

Existe un problema de compatibilidad que consiste en que los valores de SQLSTATE no son coincidentes entre los distintos DBMS, salvo los prefijos.

Los valores de SQLCODE suelen resultar todavía más incompatibles, ya que a excepción del valor 100 (NOT FOUND) los valores positivos o negativos concretos no están estandarizados. Por ejemplo, ante un PRIMARY KEY VIOLATION, DB2 lanza un valor ,Oracle otro, etc.

En ANSI SQL3 los casos que no resultan exitosos se diferencian en dos: *condiciones* y *excepciones*.

- Se consideran *condiciones* a los valores que representan casos normales, o sea **NOT FOUND** producido por una sentencia que no devuelva tuplas y **SQLWARNING (**Ej: truncamiento).

- Se consideran *excepciones* a los valores que representan casos anormales. Un ejemplo de excepción es la VIOLACION DE UNA RESTRICCION, como ser un PRIMARY KEY en una inserción o actualización.

SQL3 permite declarar un "manejador" para ambos casos, o sea, definir cómo reaccionar ante cierta *condición* o *excepción*. Esto no es otra cosa que permitir realizar *programación defensiva*.

Como el DBMS considera que el nivel de severidad no es igual en el caso de las *condiciones* y *excepciones*, el tratamiento en ambos casos es diferente. El algoritmo es el siguiente:

- Si se produce una ***condición*** y no hay declarado un manejador para ellas, el DBMS continúa ejecutando la próxima sentencia. Si hay un manejador, le transfiere el control al mismo.

- Si se produce una ***excepción*** y no hay declarado un manejador para ellas, el DBMS termina inmediatamente la ejecución de dicho PSM y retorna al invocador. Si hay un manejador, le transfiere el control al mismo.

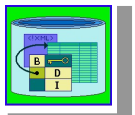
Un *manejador* consta de dos zonas: una zona que declara bajo qué condiciones se le transfiere el control al manejador *(valores generales o específicos)* y otra zona que declara a dónde transferir el control *(continue/exit/undo)* luego de que el manejador termine su acción *(sentencias)*.

Sintaxis de Declaración de un Manejador

```
DECLARE  { CONTINUE/EXIT/UNDO }
HANDLER FOR { condicionesGenerales/condicionesEspecíficas }
sentencias;
```

donde:

- ***condicionesGenerales*** **pueden ser: SQLEXCEPTION (o sea SQLSTATE con prefijo distintos de 00, 01 y 02), SQLWARNING (o sea SQLSTATE con prefijo 01), NOT FOUND (o sea SQLSTATE con prefijo 02)**



- ***condicionesEspecíficas* pueden ser: SQLSTATE *stringCte* o un alias declarado previamente de la siguiente manera:**

```
DECLARE  alias CONDITION FOR SQLSTATE 'stringCte'
```

o bien

```
DECLARE  alias CONDITION FOR SQLCODE nroCte
```

- **si hay más de una sentencia, deberá usarse un bloque BEGIN-END.**

Si la transferencia de control indica **CONTINUE** significa que se debe seguir con la próxima sentencia a la que produjo la *condición* o *excepción*, **EXIT** significa que se debe salir del bloque que contiene la sentencia SQL que produjo la *condición* o *excepción*, y **UNDO** es igual que EXIT pero además realiza un ROLLBACK de todas las sentencias SQL dentro del bloque donde se produjo la *condición* o *excepción*.

Además de las *condiciones* o *excepciones* contempladas por el DBMS puede ocurrir que los programadores quieran lanzar sus "propias excepciones" para indicar sobre un error ocurrido. La estrategia de generar **nuevas excepciones** desde un PSM se usa cuando se viola una regla de la lógica de negocios de la organización, como, por ejemplo, el sueldo de un empleado jefe debe superar en un 30% el promedio de los empleados que tiene a su cargo. Así, se notifica del problema específico (que no estaba previsto por el DBMS). Para ello existe la sentencia SIGNAL que fuerza el seteado de la variable del SQLSTATE con el valor solicitado. Lo que ocurre luego de forzar que se produzca un error, depende de si hay o no manejador definido para dicho valor.

Si los programadores usaran para los errores propios de la organización un valor de SQLSTATE ya definido por el DBMS, se confundirían los mensajes de error. Por ejemplo, si usara un SQLSTATE correspondiente a "PRIMARY KEY VIOLATION" para indicar un error que consiste en que el "sueldo de los empleados supera al sueldo del gerente" se reaccionaría equivocadamente. Los DBMS dejan rangos de SQLSTATE sin usar, permitiendo definir errores nuevos y sus mensajes de error correspondientes.

También puede usarse la sentencia RESIGNAL para redefinir una excepción ya lanzada, pero sólo puede ser lanzada desde un manejador de excepciones.

Sintaxis para Lanzar Excepciones

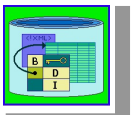
```
SIGNAL  SQLSTATE stringCte SET MESSAGE_TEXT= stringMessage;
```

o bien

```
RESIGNAL  SQLSTATE stringCte SET MESSAGE_TEXT= stringMessage;
```

Ejemplo 13

El siguiente procedimiento muestra un código que maneja 2 inserciones y en el medio una sentencia que genera truncamiento (condición). Si no existiera el manejador de la misma seguiría ejecutando la próxima sentencia (insert en auditoría). Como se declaró manejador para la misma se transfiere el control el cual indica setear el error y salir de la ejecución.



Observar que es complejo leer los PSM ANSI porque la zona de manejo de excepciones/condiciones se definen al comienzo (no están cerca de la zona donde se produce el problema, al estilo try/catch).

```
CREATE PROCEDURE NewEmpleado (LEGAJO INT, NOMBRE CHAR(10), SUELDO
DOUBLE, OUT error char(5))
BEGIN
  DECLARE SQLSTATE CHAR(5);
  DECLARE exit HANDLER
          FOR SQLWARNING
          SET error=SQLSTATE;

  INSERT INTO empleado VALUES (legajo, nombre, sueldo);

  SET error= 'demasiado largo';

  INSERT INTO auditoria VALUES (USER, current timestamp);
END
```

2 1

SQLWarning

1.5.5.1 "PostgreSQL"

El manejo de condiciones y excepciones en PostgreSQL es totalmente distinto. La idea básicamente es la misma, pero la sintaxis no.

La zona manejadora de excepciones se coloca siempre al final, o sea, es lo último antes de terminar un bloque. Cuando se produce una condición o excepción, se transfiere el control a dicha zona (si la hubiere) para ver si está contemplado atraparla.

PostgreSQL no diferencia entre *condiciones* y *excepciones*. Todas son *excepciones.* El algoritmo es el siguiente:

Algoritmo PostgreSQL para manejo de excepciones

Si dentro de un bloque se produce una excepción:

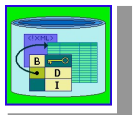
1) deshace todas las sentencias SQL que hubiera realizado él (hace rollback).

2) se fija si hay manejador de excepción para ella en dicho bloque y procede con alguna se las siguientes 3 alternativas, según el caso:

2.1) si lo hay, entonces le transfiere el control y la ejecución sigue normal a partir de allí.

2.2) si no lo hay y el bloque está anidado en otro bloque, relanza la excepción al bloque que la contiene (propaga la excepción). Dicho bloque externo, recibe la excepción "como si se hubiera producido en él", es decir sigue otra vez con el paso 1) explicado anteriormente.

2.3) si no lo hay y el bloque era el más externo, relanza la excepción, pero además termina anormalmente (no exitosamente).



Por otro lado, PostgreSQL maneja también un SQLCA, caracterizado sólo por SQLSTATE Agrega además la variable SQLERRM que contiene un mensaje más claro asociado a la excepción (no codificado). Pero tener en cuenta que en PostgreSQL los valores dichas variables asociadas a la excepción SOLO PUEDEN CONSULTARSE dentro la cláusula EXCEPTION.

Sintaxis de Declaración de un Manejador en PostgreSQL

```
BEGIN
   ....

   -- manejador de excepciones en el final
   EXCEPTION
      WHEN exception1 THEN accion1;
      WHEN expeption2 THEN accion2;
      ...
    [ WHEN OTHERS THEN accionOthers; ]   -- las no contempladas arriba
END;
```

Sintaxis para Lanzar Excepciones en PostgreSQL

- Para relanzar la misma excepción recibida;
  RAISE exceptionName;

- Para lanzar una nueva exception:
  RAISE EXCEPTION 'messageError' [USING ERRCODE = 'xxxx'];

Si se proporciona un ERRCODE no usar los ya existentes para no confundir al usuario.

Ejemplo 14

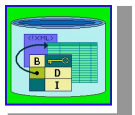
El ejecutar el siguiente código de bloque anónimo

```
DO $$

BEGIN
1  raise notice '%', 28 / 0;
   raise notice 'no pasa por aca';
END;

$$;
```

Exception

Sólo se ejecuta la primera línea y como se produce una división por cero, la excepción corta la ejecución ya que no hay manejador declarado para la misma. Es decir, la **ejecución finaliza anormalmente**.

El cliente que lo ejecuta obtiene un error
ERROR: división por cero
SQL state: 22012



Ejemplo 15

En esta variante se declara una zona de excepción, pero no atrapa el error de división por cero (22012) sino otro error que es el de formato inválido de timestamp. Por lo tanto, se obtiene el mismo efecto que el anterior. Se ejecuta la primera línea, la cual produce la excepción de división por cero y como no hay manejador para la misma se corta la ejecución. Es decir, la **ejecución finaliza anormalmente**.

```
DO $$
BEGIN
1   raise notice '%', 28 / 0;                          ──► Exception
    raise notice 'no pasa por aca';
EXCEPTION
    WHEN SQLSTATE '22007' THEN
      raise notice '% % ', SQLSTATE, SQLERRM;
END;

$$;
```

El cliente que lo ejecuta obtiene un error

ERROR: división por cero
SQL state: 22012

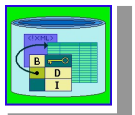
Ejemplo 16

En esta variante se atrapa la excepción producida. Se ejecuta la primera línea, la cual produce la excepción de división por cero y como hay manejador para la misma le transfiere el control. Allí, se imprime el valor de las variables de error SQLSTATE y SQLERRM. A diferencia de los casos anteriores, el bloque **finaliza exitosamente**.

```
DO $$

BEGIN
1   raise notice '%', 28 / 0;                          ──► Exception
    raise notice 'no pasa por aca';                        │
EXCEPTION                                                  │
    WHEN SQLSTATE '22012' THEN                             │
2     raise notice '% % ', SQLSTATE, SQLERRM;  ◄───────────┘
END;

$$;
```

El cliente que lo ejecuta no obtiene ningún error (el cartel de división por cero fue impreso desde el manejador).

NOTICE: 22012 división por cero
Query returned successfully with no result in 14 ms.



Los casos más complejos se producen cuando los bloques, sean anónimos o PSM, involucran sentencias SQL y se producen excepciones, por lo explicado anteriormente (el algoritmo de rollback).

A continuación, discutimos algunos casos.

Ejemplo 17

Supongamos que tenemos es siguiente esquema:

```
create table empleado
(
   legajo   INT NOT NULL PRIMARY KEY,
   nombre   varchar(10),
   sueldo   double
);

create table auditoria
(
   user     varchar(20),
   fecha    timestamp
);
```

Supongamos que inicialmente no hay tuplas insertadas.

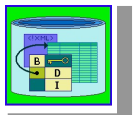
Si se tuviera el siguiente PSM que no tiene manejador de errores:

```
CREATE OR REPLACE FUNCTION  NewEmpleado (LEGAJO INT, NOMBRE CHAR(10),
SUELDO FLOAT) RETURNS VARCHAR AS $$

DECLARE
auxi VARCHAR(10);

BEGIN
1  INSERT INTO empleado VALUES (legajo, nombre, sueldo);

2  auxi:= 'demasiado largo';

   INSERT INTO auditoria VALUES (USER, current_timestamp);

   RETURN auxi;
END;

$$   LANGUAGE plpgsql ;
```

SQLWarning (para PostgreSQL Exception)

al invocarlo con el siguiente bloque anónimo

```
DO $$
DECLARE
   rta VARCHAR(10);
BEGIN
   rta:= NewEmpleado(10, 'Ana', 500);
   raise notice 'valor retornado %', rta;
END;

$$;
```



El PSM ejecuta la primera sentencia, es decir inserta en la tabla empleado. Luego, al intentar asignar en la variable *auxi* un cartel más largo que el definido, se obtiene un problema de truncamiento (lo que en ANSI sería una ***"condición"*** o ***"warning"***). Como en PostgreSQL las condiciones son excepciones, (todo en PostgreSQL es una excepción) se genera una excepción.

Cada vez que se produce una excepción se deshacen las sentencias SQL realizadas en el bloque que produjo la excepción (rollback del insert en la tabla empleado) y como es el bloque más externo y no tiene manejador para la misma, finaliza la ejecución anormalmente y relanza la excepción al invocador.

Así, el PSM termina anormalmente, el cliente recibe la excepción pero tampoco la maneja con lo que no llega a imprimir el cartel que dice 'valor retornado'. Las tablas quedan intactas (en este caso con 0 tuplas):

EMPLEADO		
Legajo	Nombre	Sueldo

AUDITORIA	
User	Fecha

El cliente que lo ejecuta obtiene un error

ERROR: el valor es demasiado largo para el tipo character varying(10)
SQL state: 22001

Ejemplo 18

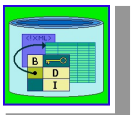
Si el bloque anónimo invocador decidiera capturar la excepción que pueda producir el PSM NewEmplado, es decir si el cliente tuviera el siguiente código:

```
DO $$
DECLARE
  rta VARCHAR(10);
BEGIN
  rta:= NewEmpleado(10, 'Ana', 500);
  raise notice 'valor retornado %', rta;
EXCEPTION
WHEN OTHERS THEN
     raise notice 'valor retornado %', rta;
    raise notice '% %', SQLSTATE, SQLERRM;
END;

$$;
```

El PSM se comporta igual que no explicado en el ejemplo anterior, pero el cliente atrapa la excepción que recibe (el cliente terminar normal), e imprime lo siguiente:

NOTICE: valor retornado <NULL>
NOTICE: 22001 el valor es demasiado largo para el tipo character varying(10)

Query returned successfully with no result in 54 ms.



Es decir, como el PSM finalizó anormalmente no llegó a devolver un valor (return) con lo que la variable está en null. Además, los valores SQLSTATE y SQLERRM asociados a la excepción son los que se imprimen. Ninguna inserción queda en las tablas.

Ejemplo 19

Continuando con el ejemplo anterior, si el PSM se modificara de la siguiente manera, usando bloques anidados:

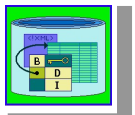
```
CREATE OR REPLACE FUNCTION  NewEmpleado(LEGAJO INT, NOMBRE CHAR(10),
SUELDO FLOAT) RETURNS VARCHAR AS $$
DECLARE
   rta VARCHAR(10);
BEGIN -- block 1
(1) INSERT INTO empleado VALUES (legajo, nombre, sueldo);
   BEGIN -- block 2
(2)    INSERT INTO auditoria VALUES (USER, current_timestamp);
(3)    rta:= 'demasiado largo';
          raise notice 'nunca llega aca';
   EXCEPTION
(4) WHEN OTHERS THEN
      rta:= 'bloque 2';
   END;

(5) return rta;
END;
$$   LANGUAGE plpgsql ;
```

SQLWarning (para PostgreSQL Exception)

En el bloque 1 se produjo la inserción en la tabla empleado (1). Luego, en el bloque 2 se realiza una inserción en la tabla auditoría (2) y se produce una excepción por truncamiento (3). Así, el bloque 2 (que es donde se produce la excepción) deshace las sentencias SQL realizadas en él (rollback) y transfiere el control a su manejador de excepciones donde se setea el valor de la variable rta (4). Como la excepción fue manejada (atrapada) la ejecución continúa normalmente. Dicho bloque 1 finaliza. El bloque 2 no recibe ningún problema y finaliza normalmente, retornando el valor al módulo invocador (5).

Suponiendo que lo invocamos con:

```
DO $$
DECLARE
   rta VARCHAR(10);
BEGIN
   rta:= NewEmpleado(10, 'Ana', 500);
   raise notice 'valor retornado %', rta;
EXCEPTION
WHEN OTHERS THEN
     raise notice 'valor retornado %', rta;
      raise notice '% %', SQLSTATE, SQLERRM;
END;

$$;
```



El PSM termina normalmente, sólo una operación queda en la BD (la inserción en la tabla empleado) y el cliente imprime el valor obtenido por la ejecución normal del PSM

```
NOTICE:  valor retornado bloque 2
Query returned successfully with no result in 12 ms.
```

Y las tablas quedan de la siguiente manera

EMPLEADO		
Legajo	Nombre	Sueldo
10	Ana	500

AUDITORIA	
User	Fecha

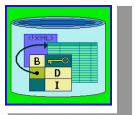
Ejemplo 20

¿Hubiera sido lo mismo si se cambia el código de la siguiente manera, donde el manejo de excepción se realiza en el bloque 1? Supongamos que las tablas están vacías, y se lo invoca con el mismo código cliente del ejemplo anterior.

```
CREATE OR REPLACE FUNCTION  NewEmpleado(LEGAJO INT, NOMBRE CHAR(10),
SUELDO FLOAT) RETURNS VARCHAR AS $$
DECLARE
   rta VARCHAR(10);
BEGIN -- block 1
(1) INSERT INTO empleado VALUES (legajo, nombre, sueldo);
   BEGIN -- block 2
(2)    INSERT INTO auditoria VALUES (USER, current_timestamp);
(3)    rta:= 'demasiado largo';
          raise notice 'nunca llega aca';
   END;
  EXCEPTION
      WHEN OTHERS THEN
(4)   rta:= 'bloque 1';
(5)   return rta;
END;
$$   LANGUAGE plpgsql ;
```

SQLWarning (para PostgreSQL Exception)

En el bloque 1 se produjo la inserción en la tabla empleado (1). Luego, en el bloque 2 se realiza una inserción en la tabla auditoría (2) y se produce una excepción por truncamiento (3). Así, el



bloque 2 (que es donde se produce la excepción) deshace las sentencias SQL realizadas en él (rollback) y como no tiene definido manejador para la misma, relanza la excepción al bloque 2.

Dicho bloque recibe la excepción "como si se hubiera producido en él", por lo tanto, deshace las sentencias SQL realizadas en él (el insert en la tabla empleado) y transfiere el control a su manjeador. Allí, setea el valor de la variable (4) y retorna normalmente al módulo invocador (5).

La ejecución del PSM fue exitosa (la excepción fue atrapada). Pero el efecto no es el mismo que en el ejemplo anterior porque en este caso NINGUNA de las inserciones persisten en la BD.

Las tablas quedan de la siguiente manera

EMPLEADO		
Legajo	Nombre	Sueldo

AUDITORIA	
User	Fecha

El cliente imprimiría

```
NOTICE:  valor retornado bloque 1
Query returned successfully with no result in 26 ms.
```

Consejo

Tratar de no relanzar las excepciones hacia afuera y atraparlas en donde se produjo la excepción para evitar la propagación. Es decir, que cada bloque sea responsable de intentar solucionar los errores que en él se produzcan.

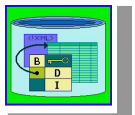
No mezclar sentencias SQL que se saben que no lanzan excepción con otras que sí en los mismos bloques, porque se deshará todas las sentencias SQL del bloque.

Ejemplo 21

Supongamos que ya tenemos insertadas las siguientes tuplas:

EMPLEADO		
Legajo	Nombre	Sueldo
10	Ana	500
20	Pablo	3000
30	Julia	4000

AUDITORIA	
User	Fecha
Salerno	2005-04-23 17:07:32
Salerno	2005-04-23 17:46:55



¿Cómo hacer para que las inserciones se intenten por separado, es decir la inserción en auditoria no esté supeditada a la otra?

Manejar cada una de las sentencias que por algún motivo puedan lanzar excepción (PK/FK violation, etc) en bloques separados, cada uno con su manejador de excepciones.

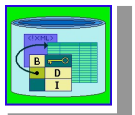
```
CREATE OR REPLACE FUNCTION  NewEmpleado(LEGAJO INT, NOMBRE CHAR(10),
SUELDO FLOAT) RETURNS VARCHAR AS $$
BEGIN -- block 1

  BEGIN  -- block 2
    INSERT INTO empleado VALUES (legajo, nombre, sueldo);
  EXCEPTION
  WHEN OTHERS THEN
    raise notice 'el problema fue en empleado. Go on';
  END;

  BEGIN -- block 3
    INSERT INTO auditoria VALUES (USER, current_timestamp);
  EXCEPTION
  WHEN OTHERS THEN
    raise notice 'el problema fue en auditoria. Go on';
       END;

return 'OK';
END;
$$   LANGUAGE plpgsql ;
```

En este caso específico, auditoría no puede lanzar excepciones (no tiene PK, etc). Puede simplificarse así:

```
CREATE OR REPLACE FUNCTION  NewEmpleado(LEGAJO INT, NOMBRE CHAR(10),
SUELDO FLOAT) RETURNS VARCHAR AS $$
BEGIN -- block 1

  BEGIN  -- block 2
    INSERT INTO empleado VALUES (legajo, nombre, sueldo);
  EXCEPTION
  WHEN OTHERS THEN
    raise notice 'el problema fue en empleado. Go on';
  END;

INSERT INTO auditoria VALUES (USER, current_timestamp);
return 'OK';

END;
$$   LANGUAGE plpgsql ;
```

Suponiendo que lo invocamos con:

```
DO $$
DECLARE
  rta VARCHAR(10);
BEGIN
  rta:= NewEmpleado(10, 'Pablo', 5500);
  raise notice 'valor retornado %', rta;
EXCEPTION
WHEN OTHERS THEN
     raise notice 'valor retornado %', rta;
    raise notice '% %', SQLSTATE, SQLERRM;
END;
$$;
```



El PSM termina exitosamente. En realidad, la inserción en la tabla empleado es la que produce la excepción (PK violation), pero la misma es atrapada. La inserción en auditoria ocurre normalmente.

El cliente recibiría

NOTICE: el problema fue en empleado. Go on
NOTICE: valor retornado OK
Query returned successfully with no result in 13 ms.

Las tablas quedan así:

EMPLEADO		
Legajo	Nombre	Sueldo
10	Ana	500
20	Pablo	3000
30	Julia	4000

AUDITORIA	
User	Fecha
Salerno	2005-04-23 17:07:32
Salerno	2005-04-23 17:46:55
Salerno	2005-04-29 22:00:00

Consejo

Tener muchos bloques anidados complica la lectura del código. Más prolijo es definir PSMs e invocarlos.

No es cuestión de escribir un PSM por cada sentencia SQL, sino detectar "cuales sentencias SQL funcionan como un todo y si falla una debe deshacerse todas juntas. Esas son las que van al mismo PSM".