PL/pgSQL - SQL Procedural Language

28 de marzo de 2003

Resumen

***Esta es la primera revisión de la traducción del capítulo 19 de la obra "PostgreSQL 7.4devel Documentation", creada por el* The PostgreSQL Global Development Group.**

Traducción: Adolfo Pachón (Proyecto S.O.B.L.)

Índice General

1 Introducción

Los objetivos de diseño para PL/pgSQL fueron crear un lenguaje procedural cargable que

- pudiera ser usado para crear funciones ytriggers,
- añadiera estructuras de control al lenguaje SQL,
- pudiera realizar computaciones complejas,
- heredase todos los tipos definidos por el usuario, funciones y operadores,
- pudiera ser definido para ser validado por el servidor,
- fuese sencillo de utilizar.

El manejador de llamadas PL/pgSQL interpreta la el código fuente en texto plano de la función y produce un árbol de instrucciones binarias internas la primera vez que ésta es llamada (dentro de cualquier proceso principal). El árbol de instrucciones traduce toda la estructura del estamento PL/pgSQL, pero las expresiones individuales de SQL y las consultas SQL usadas en la función no son traducidas inmediatamente.

Como cada expresión y consulta SQL es usada primero en la función, el intérprete PL/pgSQL crea un plan de ejecución preparado (usando las funciones del gestor SPI *SPI_prepare y SPI_saveplan*). Las siguientes visitas a dicha expresión o consulta reutilizar en plan preparado. Así, una función con código condicional que contiene muchos estamentos para los cuales el plan de ejecución podría ser requerido sólo preparará y almacenará aquellos planes que realmente sean usados durante el tiempo de vida de la conexión a la base de datos. Esto puede consecuentemente reducir la cantidad de tiempo requerida para interpretar y generar planes de consultas para los estamentos en una función de lenguaje procedural. Una desventaja es que los errores en una expresión o consulta específica puede no ser detectada hasta que no se llegue a esa parte de la función en la ejecución.

Una vez que PL/pgSQL ha realizado un plan de consulta para una determinada consulta en una función, éste reusará dicho plan durante toda la vida de la conexión a la base de datos. Esto significa normalmente una ganancia en el rendimiento, pero puede causar algunos problemas si usted altera dinámicamente su esquema de base de datos. Por ejemplo:

```
CREATE FUNCTION populate() RETURNS INTEGER AS '
DECLARE
  -- Declarations
BEGIN
  PERFORM my_function();
END;
' LANGUAGE 'plpgsql';
```

Si usted ejecuta la función anterior, ésta referenciará el OID para *my_function()* en el plan de consulta producido para el estamento PERFORM. Más tarde, si useted elimina y vuelve a crear la función *my_function()*, entonces *populate()* no será capaz

de encontrar la función *my_function()* nunca más. Usted debería crear de nuevo *populate()*, o iniciar una nueva sesión de bases de datos para que se realice una nueva compilación.

Debido a que PL/pgSQL almacena los planes de ejecución de esta forma, las consultas que aparezcan directamente en una función PL/pgSQL se deben referir a las mismas tablas y campos en cada ejecución; es decir, no puede usar un parámetro como el nombre de una tabla o campo en una consulta. Para evitar esta restricción, puede construir consultas dinámicas usando el estamente de PL/pgSQL EXECUTE -al coste de construir un nuevo plan de consultas en cada ejecución-.

> **NOTA**: El estamento EXECUTE de PL/pgSQL no está relacionado con el estamento EXECUTE soportado por el motor PostgreSQL. El estamento del motor no puede ser usado dentro de funciones PL/pgSQL (y no es necesario hacerlo).

Excepto para el caso de conversión de entrada/salida y funciones de cálculo para tipos definidos por el usuario, cualquier cosa que pueda ser definida en funciones del lenguaje C también pueden serlo con PL/pgSQL. Es posible crear complejas funciones de computación condicionales y más tarde usarlas para definir operadores u usarlas en índices funcionales.

1.1 Ventajas de usar PL/pgSQL

- Mayor rendimiento (vea Sección 1.1.1)
- Soporte SQL (vea Sección 1.1.2)
- Portabilidad (vea Sección 1.1.3)

1.1.1 Mayor Rendimiento

SQL es el lenguaje que PostgreSQL (y la mayoría del resto de bases de datos relacionales) usa como lenguaje de consultas. Es portable y fácil de aprender. Pero cada estamento SQL debe ser ejecutado individualmente por el servidor de bases de datos.

Esto significa que su aplicación cliente debe enviar cada consulta al servidor de bases de datos, esperar a que se procese, recibir el resultado, realizar alguna computación, y luego enviar otras consultas al servidor. Todo esto incurre en una comunicación entre procesos y también puede sobrecargar la red si su cliente se encuentra en una máquina distinta al servidor de bases de datos.

Con PL/pgSQL puede agrupar un grupo de computaciones y una serie de consultas dentro del servidor de bases de datos, teniendo así la potencia de un lenguaje procedural y la sencilllez de uso del SQL, pero ahorrando una gran cantidad de tiempo porque no tiene la sobrecarga de una comunicación cliente/servidor. Esto puede redundar en un considerable aumento del rendimiento.

1.1.2 Soporte SQL

PL/pgSQL añade a la potencia de un lenguaje procedural la flexibilidad y sencillez del SQL. Con PL/pgSQL puede usar todos los tipos de datos, columnas, operadores y funciones de SQL.

1.1.3 Portabilidad

Debido a que las funciones PL/pgSQL corren dentro de PostgreSQL, estas funciones funcnionarán en cualquier plataforma donde PostgreSQL corra. Así podrá reusar el código y reducir costes de desarrollo.

1.2 Desarrollando en PL/pgSQL

Desarrollar en PL/pgSQL es agradable y rápido, especialmente si ya ha desarrollado con otros lenguajes procedurales de bases de datos, tales como el PL/SQL de Oracle. Dos buenas formas de desarrollar en PL/pgSQL son:

- Usar un editor de texto y recargar el archivo con *psql*
- Usando la herramienta de usuario de PostgreSQL: *PgAccess*

Una buena forma de desarrollar en PL/pgSQL es simplemente usar el editor de textos de su elección y, en otra ventana, usar *psql* (monitor interactivo de PostgreSQL) para cargar las funciones. Si lo está haciendo de ésta forma, es una buena idea escribir la función usando CREATE OR REPLACE FUNCTION. De esta forma puede recargar el archivo para actualizar la definición de la función. Por ejemplo:

```
CREATE OR REPLACE FUNCTION testfunc(INTEGER) RETURNS INTEGER AS '
...;
end; '
LANGUAGE 'plpgsql';
```

Mientras corre *psql*, puede cargar o recargar la definición de dicha función con

```
\i filename.sql
```

y luego inmediatamente utilizar comandos SQL para testear la función.

Otra buena forma de desarrollar en PL/pgSQL es usando la herramienta administrativa gráfica de PostgreSQL: *PgAccess*. Esta hace algunas cosas más cómodas para usted, tales como escapar comillas simples, y facilitar la recreación y depuración de funciones.

2 Estructura de PL/pgSQL

PL/pgSQL es un lenguaje estructurado de bloques. El texto completo de una definición de función debe ser un bloque. Un bloque se define como:

```
[ <<etiqueta>> ]
[ DECLARE
  declaraciones ]
BEGIN
  estamentos
END;
```

Cualquier estamento en la sección de estamentos de un bloque puede ser un subbloque. Los subbloques pueden ser utilizados para agrupaciones lógicas o para lcoalizar variables para un pequeño grupo de estamentos.

Las variables declaradas en la sección de declaraciones que preceden a un bloque son inicializadas a sus valores por defecto cada vez que el bloque es introducido, no sólo una vez por cada llamada a la función. Por ejemplo:

```
CREATE FUNCTION algunafuncion() RETURNS INTEGER AS '
DECLARE
  cantidad INTEGER := 30;
BEGIN
  RAISE NOTICE ''La cantidad aquí es  %'',cantidad; -- La cantidad aquí es  30
  cantidad := 50;
  --
  -- Crea un subbloque
  --
  DECLARE
    cantidad INTEGER := 80;
  BEGIN
    RAISE NOTICE ''La cantidad aquí es  %'',cantidad; -- La cantidad aquí es 80
  END;
  RAISE NOTICE ''La cantidad aquí es %'',cantidad; -- La cantidad aquí es 50
  RETURN cantidad;
END;
' LANGUAGE 'plpgsql';
```

Es importante no confundir el uso de BEGIN/END para la agrupación de estamentos en PL/pgSQL con los comandos de base de datos para el control de transacciones. Los comandos BEGIN/END de PL/pgSQL son sólo para agrupar; ellos no inician o finalizan una transacción. Los procedimientos de funciones y triggers son siempre ejecutados dentro de una transacción establecida por una consulta externa -no pueden iniciar o validar transacciones, ya que PostgreSQL no tiene transacciones anidadas.

2.1 Detalles Léxicos

Cada estamento y declaración dentro de un bloque está terminado por un punto y coma.

Todas las palabras clave e identificadores pueden ser escritos en caracteres mayúsculas/minúsculas. Los identificadores serán implícitamente convertidos a minúsculas, a menos que estén entrecomillados (usando para ello comillas dobles).

Hay dos tipos de comentarios en PL/pgSQL. Un guión doble -inicia un comentario que se extiende hasta el final de la línea-. Un /* marca el inicio de un bloque de comentarios que se extiende hasta la siguente ocurrencia de */. Los bloques de comentarios no pueden anidarse, pero los comentarios con doble guionado pueden encerrarse en un bloque de comentarios, y un doble guión puede ocultar los delimitadores de bloques de comentarios /* y */.

3 Declaraciones

Todas las variables, filas y registros usados en un bloque deben estar declaradas en la sección de declaraciones del bloque (la única excepción es que la variable de bucle para una iteración de bloque FOR sobre un rango de valores enteros es automáticamente declarada como variable entera).

Las variables PL/pgSQL pueden ser de cualquier tipo de datos SQL, tales como INTEGER, VARCHAR y CHAR.

Aquí tiene algunos ejemplos de declaración de variables:

```
user_id INTEGER;
cantidad NUMERIC(5);
url VARCHAR;
myrow nombretabla%ROWTYPE;
myfield nombretabla.nombrecampo%TYPE;
unafila RECORD;
```

La sintaxis general de una declaración de variables es:

```
nombre [ CONSTANT ] tipo [ NOT NULL ] [ { DEFAULT | := } expresión ];
```

La cláusula DEFAULT, si existe, especifica el valor inicial asignado a la variable cuando el bloque es introducido. Sinla cláusula DEFAULT no es facilitada entonces la variable es inicializada al valor SQL NULL.

La opción CONSTANT previene la asignació de otros valores a la variable, así que su valor permanece durante la duración del bloque. Si se especifica NOT NULL, una asignación de un valor NULL resulta en un error en tiempo de ejecución. Todas las variables declaradas como NOT NULL deben tener un valor por defecto no nulo especificado.

El valor por defecto es evaluado cada vez que el bloque es introducido. Así, por ejemplo, la asignación de *now* a una variable de tipo *timestamp* provoca que la variable tenga la fecha y hora de la llamada a la actual función, y no la fecha y hora de su compilación.

Ejemplos:

```
cantidad INTEGER DEFAULT 32;
url varchar := ''http://www.sobl.org'';
user_id CONSTANT INTEGER := 10;
```

3.1 Alias para los Parámetros de la Función

```
nomrbe ALIAS FOR $n;
```

Los parámetros pasados a las funciones son nominados con los identificadores $1, $2, etc. Opcionalmente, se pueden declara alias para los nombres de los parámetros, con el objeto de incrementar la legibilidad del código. Tanto el alias como el identificador numérico pueden ser utilizados para referirse al valor del parámetro. Algunos ejemplos:

```
CREATE FUNCTION tasa_ventas(REAL) RETURNS REAL AS '
DECLARE
  subtotal ALIAS FOR $1;
BEGIN
  return subtotal * 0.06;
END;
' LANGUAGE 'plpgsql';
CREATE FUNCTION instr(VARCHAR,INTEGER) RETURNS INTEGER AS '
DECLARE
  v_string ALIAS FOR $1;
  index ALIAS FOR $2;
BEGIN
  -- Algunas computaciones aquí
END;
' LANGUAGE 'plpgsql';
CREATE FUNCTION usa_muchos_campos(tablename) RETURNS TEXT AS '
DECLARE
  in_t ALIAS FOR $1;
BEGIN
  RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
' LANGUAGE 'plpgsql';
```

3.2 Tipos Fila (Row)

```
nombre nombretabla%ROWTYPE;
```

Una variable de un tipo compuesto es denominada *variable de fila* (*row-type variable*). Una variable de este tipo puede almacenar una fila completa del resultado de una consulta SELECT o FOR, mientras que la columna de dicha consulta coincida con el tipo declarado para la variable. Los campos individuales del valor de la fila son accedidos usando la típica notación de puntos, por ejemplo *variablefila.campo.*

En la actualidad, una variable de tipo fila sólo puede ser declarada usando la notación %ROWTYPE; aunque uno podría esperar que un nombre público de de nombre de tabla funcionase como tipo de declaración, ésta no sería aceptada dentro de funciones PL/pgSQL.

Los parámetros para una función pueden ser tipos compuestos (filas completas de tablas). En ese caso, el correspondiente identificador $n será una variable tipo fila, y los campos podrán ser accedidos, por ejemplo *$1.nombrecampo*.

Sólo los atributos de una tabla definidos por el usuario son accesibles en una variable tipo fila, y no OID u otros atributos de sistema (porque la fila podría venir de una vista). Los campos del tipo fila heredan el tamaño del campo de la tabla así como la precisión para tipos de datos, tales como *char(n)*.

```
CREATE FUNCTION usa_dos_tablas(tablename) RETURNS TEXT AS '
DECLARE
  in_t ALIAS FOR $1;
  use_t tabla2nombre%ROWTYPE;
BEGIN
  SELECT * INTO use_t FROM tabla2nombre WHERE ... ;
  RETURN in_t.f1 || use_t.f3 || in_t.f5 || use_t.f7;
END;
' LANGUAGE 'plpgsql';
```

3.3 Records

```
nombre RECORD;
```

Las variables de tipo *registro* (*record*) son similares a las tipo fila, pero no tienen una estructura predefinida. Ellas la toman de la actual estructura de la fila que tienen asignada durante un comando SELECT o FOR. La subestructura de una variable tipo registro puede variar cada vez que se le asigne un valor. Una consecuencia de esto es que hasta que a una variable tipo registro se le asigne valor por vez primera, ésta no tendrá subestructura, y cualquier intento de acceder a un campo en ella provocará un error en tiempo de ejecución.

Advierta que RECORD no es un verdadero tipo de datos, sólo un almacenador.

3.4 Atributos

Usando los atributos %TYPE y %ROWTYPE, puede declarar variables con el mismo tipo de datos o estructura que otro elemento de la base de datos (p.ej: un campo de tabla).

```
variable%TYPE
```

%TYPE proporciona el tipo de datos de una variable o de una columna de base de datos. Puede usar esto para declarar variables que almacenen valores de base de datos. Por ejemplo, digamos que tiene una columna llamada *user_id* en su tabla *usuarios*. Para declarar una variable con el mismo tipo de datos que usuarios.user_id usted escribiría:

```
user_id usuarios.user_id%TYPE;
```

Usando %TYPE no necesita conocer el tipo de datos de la estructura a la que está referenciando, y lo más importante, si el tipo de datos del elemento referenciado cambia en el futuro (p.ej.: usted cambia su definición de tabla para *user_id* de INTEGER a REAL), no necesitará cambiar su definición de función.

```
tabla%ROWTYPE
```

%ROWTYPE proporciona el tipo de datos compuesto correspondiente a toda la fila de la tabla especificada. La tabla debe ser una tabla existente o un nombre de vista de la base de datos.

```
DECLARE
  users_rec usuarios%ROWTYPE;
  user_id usuarios.user_id%TYPE;
BEGIN
  user_id := users_rec.user_id;
  ...
CREATE FUNCTION does_view_exist(INTEGER) RETURNS bool AS '
DECLARE
  key ALIAS FOR $1;
  table_data cs_materialized_views%ROWTYPE;
BEGIN
  SELECT INTO table_data * FROM cs_materialized_views
         WHERE sort_key=key;
  IF NOT FOUND THEN
     RETURN false;
  END IF;
  RETURN true;
END;
' LANGUAGE 'plpgsql';
```

3.5 RENAME

```
RENAME oldname TO newname;
```

Usando la declaración RENAME puede cambiar el nombre de una variable, registro o fila. Esto es inicialmente útil si NEW o OLD debieran ser referenciados por otro nombre dentro de un procedimiento trigger. Vea también ALIAS.

Ejemplos:

```
RENAME id TO user_id;
RENAME this_var TO that_var;
```

> **NOTA**: RENAME parece que no funciona en PostgreSQL 7.3. Arreglar esto es poco prioritario, ya que ALIAS cubre la mayoría de los usos prácticos de RENAME.

4 Expresiones

Todas las expresiones utilizadas en los estamentos PL/pgSQL son procesados usando el ejecutor SQL regular del servidor. Las expresiones que parecen contener constantes pueden de hecho requerir evaluación en tiempo de ejecución (p.ej. *"now"* para el tipo *timestamp*), así que es imposible para el intérprete de PL/pgSQL identificar valores reales de constantes aparte del valor clave NULL. Todas las expresiones son evaluadas internamente al ejecutar la consulta

```
SELECT expresión
```

usando el gestor SPI. En la expresión, las ocurrencias de identificadores de variables PL/pgSQL son reemplazadas por parámetros, y los actuales valores de las variables son pasados al ejecutor en el array de parámetros. Esto permite al plan de consultas para el SELECT que sea preparado sólo una vez, y luego reutilizado para subsequentes evaluaciones.

La evaluación realizada por el intérprete principal de PostgreSQL tiene algunos efectos de cara a la interpretación de valores de constantes. En detalle aquí está la diferencia entre lo que hacen estas dos funciones:

```
CREATE FUNCTION logfunc1 (TEXT) RETURNS TIMESTAMP AS '
DECLARE
  logtxt ALIAS FOR $1;
BEGIN
  INSERT INTO logtable VALUES (logtxt, ''now'');
  RETURN ''now'';
END;
' LANGUAGE 'plpgsql';
```

y

```
CREATE FUNCTION logfunc2 (TEXT) RETURNS TIMESTAMP AS '
DECLARE
  logtxt ALIAS FOR $1;
  curtime timestamp;
BEGIN
  curtime := ''now'';
  INSERT INTO logtable VALUES (logtxt, curtime);
  RETURN curtime;
END;
' LANGUAGE 'plpgsql';
```

En el caso de *logfunc1()*, el intérprete principal de PostgreSQL sabe cuándo preparar el plan para el INSERT, y la cadena *'now'* debería ser interpretada como un *timestamp* debido a que el campo destino de *logtable* es de ese tipo. Así, creará una constante a partir de él y su valor constante será usado luego en todas las invocaciones de *logfunc1()* durante el tiempo de vida del motor. No es necesario decir que esto no era lo que deseaba el programador.

En el caso de *logfunc2()*, el intérprete principal de PostgreSQL no sabe a qué tipo debería pertenecer *'now'* y por tanto devuelve un valor de tipo *text* conteniendo la cadena *'now'*. Durante la siguiente asignación a la variable local *curtime*, el intérprete de PL/pgSQL casa esta cadena con el tipo *timestamp* llamando a las funciones *text_out()* y *timestamp_in()* para la conversión. Así, el timestamp computado es actualizado en cada ejecución, tal como esperaba el programador.

La naturaleza mutable de las variables tipo registro presenta un problema en su conexión. Cuando los campos de una variable registro son usados en expresiones o

estamentos, los tipos de datos de los campos no debe cambiar entre llamadas de una y la misma expresión, ya que la expresión será planeada usando el tipo de datos que estaba presente cuando la expresión fue analizada por vez primera. Recuerde esto cuando escriba procedimientos trigger que manejen eventos para más de una tabla (EXECUTE puede ser usado para resolver este problema cuando sea necesario).

5 Estamentos Básicos

En ésta sección y las siguientes subsecciones, describiremos todos los tipos de estamentos explícitamente reconocidos por PL/pgSQL. Cualquiera no reconocido como uno de estos tipos de estamentos se presume ser una consulta SQL, y es enviada al motor de la base de datos para ser ejecutado (tras la sustitución de cualesquiera variables PL/pgSQL usadas en el estamento). Así, por ejemplo, los comandos SQL INSERT, UPDATE, y DELETE pueden ser considerados para ser estamentos de PL/pgSQL. Pero ellos no están explícitamente listados aquí.

5.1 Asignación

Una asignación de un valor a una variable o campo de fila/registro se escribe:

```
identificador := expresión;
```

Como se explicó anteriormente, la expresión en el estamento es evaluada como si de un envío de comando SQL SELECT al motor se tratara. La expresión debe contener un único valor.

Si el tipo de datos resultante de la expresión no coincide con el tipo de datos de la variable, o la variable tiene un determinado tamaño/precisión (tal como *char(20)*), el valor resultante será implícitamente convertido por el intérprete PL/pgSQL usando la resultante función de tipos *output-function* y el tipo de variable *input-function*. Advierta que esto podría potencialmente resultar en errores en tiempo de ejecución generados por la función de entrada, si el formato de la cadena del valor resultante no es aceptable para la función de entrada.

Ejemplos:

```
user_id := 20;
tax := subtotal * 0.06;
```

5.2 SELECT INTO

El resultado de un comando SELECT que contiene múltiples columnas (pero sólo una fila) puede ser asignado a una variable tipo registro, tipo fila, o variables escalares o de lista. Esto se hace así:

```
SELECT INTO destino expressions FROM ...;
```

donde *destino* puede ser una variable registro, fila, o una lista separada por comas de variables simples y campos registro/fila. Advierta que esto es algo diferente a la interpretación normal que hace PostgreSQL del SELECT INTO, la cual es que el destino

(target) INTO es una tabla recién creada (si quiere crear una tabla a partir de un resultado de SELECT dentro de una función PL/pgSQL, use la sintaxis CREATE TABLE ... AS SELECT).

Si una fila o lista de variables es usada como destino, los valores seleccionados deben coincidir exactamente con la estructura de los destinos, u ocurrirá un error en tiempo de ejecución. Cuando una variable tipo registro es el destino, esta automáticamente se configurará al tipo fila de las columnas resultantes de la consulta.

Excepto por la cláusula INTO, el estamento SELECT es igual que el normal de la consulta SQL SELECT y puede usar todo el potencial de SELECT.

Si la consulta SELECT retorna cero filas, valores nulos son asignados a los destinos. Si la consulta SELECT retorna múltiples filas, la primera es asignada a los destinos y el resto es descartado (Advierta que la "primera fila" dependerá del uso de ORDER BY.)

Actualmente, la cláusula INTO puede aparecer en cualquier lugar en la cláusula SELECT, pero se recomienda ubicarla inmediatamante después de la palabra clave SELECT. Futuras versiones de PL/pgSQL pueden ser menos restrictivas sobre esto.

Puede usar FOUND immediatamente después de un estamento SELECT INTO para determinar si la asignación tuvo éxito (es decir, al menos una fila fue retornada por el estamento SELECT). Por ejemplo:

```
SELECT INTO myrec * FROM EMP WHERE empname = myname;
IF NOT FOUND THEN
  RAISE EXCEPTION ''employee % not found'', myname;
END IF;
```

Alternativamente, puede usar el condicional IS NULL (o ISNULL) para testear si un resultado de RECORD/ROW es nulo. Advierta que no hay forma de saber si adicionales filas han sido descartadas.

```
DECLARE
  users_rec RECORD;
  full_name varchar;
BEGIN
  SELECT INTO users_rec * FROM users WHERE user_id=3;
  IF users_rec.homepage IS NULL THEN
    -- user entered no homepage, return "http://"
    RETURN ''http://'';
  END IF;
END;
```

5.3 Executando una expresión o consulta sin resultado

Algunas veces uno desea evaluar una expresión o consulta pero descartando el resultado (normalmente porque uno está llamando a una función que tiene efectos útiles, pero un resultado inútil). Para hacer esto en PL/pgSQL, use el estamento PERFORM:

```
PERFORM query;
```

Esto ejecuta una consulta SELECT y descarta el resultado. Las variables PL/pgSQL son sustituidas en la consulta de la forma usual. Además, la variable especial FOUND se establece a *true* si la consulta produce al menos una fila, o *false* si no produce ninguna.

NOTA: Uno podría esperar que SELECT sin cláusuala INTO podría acompañar al resultado, pero actualmente la única forma aceptada para hacerlo es con PERFORM.

Un ejemplo:

```
PERFORM create_mv(''cs_session_page_requests_mv'', my_query);
```

5.4 Ejecutando Consultas Dinámicas

Frecuentemente querrá generar conultas dinámicas dentro de sus funciones PL/pgSQL, es decir, consultas que implican a diferentes tablas o distintos tipos de datos cada vez que son ejecutados. Los intentos normales de crear planes de consultas de PL/pgSQL no funcionarán en estos escenarios. Para manejar el problema, se proporciona el estamento EXECUTE:

```
EXECUTE query-string;
```

donde *query-string* es una expresión que contiene una cadena (de tipo *text*) conteniendo la consulta a ser ejecutada. Esta cadena es leída literalmente por el motor SQL.

Advierta en particular que no se hacen sustituciones de variables PL/pgSQL en la cadena consulta. Los valores de las variables deben ser insertados en la cadena en el momento de su construcción.

Cuando se trabaja con consultas dinámicas tendrá que realizar el escape de comillas simples en PL/pgSQL. Vea la tabla de la sección 11 para una explicación detallada. Le ahorrará esfuerzos.

Al contrario de otras consultas en PL/pgSQL, una consulta ejecutada por un estamento EXECUTE no es preparada ni almacenada sólo una vez durante la vida del servidor. Al contrario, la consulta es preparada cada vez que se ejecuta el estamento. La consulta-cadena puede ser dinámicamente creada dentro del procedimiento para realizar acciones sobre tablas y campos.

Los resultados de consultas SELECT son descartados por EXECUTE, y SELECT INTO no está actualmente soportado dentro de EXECUTE. Así, la única forma de extraer un resultado de un SELECT creado dinámicamente es usar el formato FOR-IN-EXECUTE descrito más adelante.

Un ejemplo:

```
EXECUTE ''UPDATE tbl SET ''
        || quote_ident(fieldname)
        || '' = ''
        || quote_literal(newvalue)
        || '' WHERE ...'';
```

Este ejemplo muestra el uso de las funciones *quote_ident(TEXT)* y *quote_literal(TEXT)*. Las variables conteniendo identificadores de campos y tablas deberían ser pasadas a la función *quote_ident()*. Las variables conteniendo elementos literales de la consulta dinámica deberían ser pasados a la función *quote_literal()*. Ambas toman los pasos apropiados para retornar el texto introducido encerrado entre comillas simples o dobles y con cualesquiera caracteres especiales embebidos, debidamente escapados.

Aquí tiene un ejemplo más largo de una consulta dinámica y EXECUTE:

```
CREATE FUNCTION cs_update_referrer_type_proc() RETURNS INTEGER AS '
DECLARE
  referrer_keys RECORD; -- Declare a generic record to be used in a FOR
  a_output varchar(4000);
BEGIN
  a_output := ''CREATE FUNCTION cs_find_referrer_type(varchar,varchar,varchar)
                RETURNS VARCHAR AS ''''
                  DECLARE
                    v_host ALIAS FOR $1;
                    v_domain ALIAS FOR $2;
                    v_url ALIAS FOR $3; BEGIN '';
  --
  -- Advierta cómo escaneamos a través de los resultados para una consulta en un bucle FOR
  -- usando el constructor  FOR <registro>.
  --
FOR referrer_keys IN SELECT * FROM cs_referrer_keys ORDER BY try_order LOOP
    a_output := a_output || '' IF v_'' || referrer_keys.kind || '' LIKE '''''''''
             || referrer_keys.key_string || ''''''''' THEN RETURN ''''''
             || referrer_keys.referrer_type || ''''''; END IF;'';
END LOOP;
a_output := a_output || '' RETURN NULL; END; '''' LANGUAGE ''''plpgsql'''';'';
-- Esto funciona porque no estamos sustituyendo variables
-- De lo contrario, fallaría. Vea PERFORM para otra forma de ejecutar funciones
EXECUTE a_output; END; ' LANGUAGE 'plpgsql';
```

5.5 Obteniendo Estado de Resultado

Hay varias formas de determinar el efecto de un comando. El primer método es usar GET DIAGNOSTICS, el cual tiene el siguiente formato:

```
GET DIAGNOSTICS variable = item [ , ... ] ;
```

Este comando permite el retorno de indicadores de estado del sistema. Cada elemento es una palabra clave identificando un estado de valor a ser asignado a la variable especificada (que debería ser del tipo de datos correcto para recibirlo). Los elementos de estado disponibles actualmente son ROW_COUNT, el número de filas procesadas por la última consulta SQL enviada al motor SQL; y RESULT_OID, el OID de la última fila insertada por la más reciente consulta SQL. Advierta que RESULT_OID sólo es útil tras una consulta INSERT.

```
GET DIAGNOSTICS var_integer = ROW_COUNT;
```

Hay una variable especial llamada FOUND, de tipo *boolean*. FOUND se inicializa a *false* con cada función PL/pgSQL. Es valorada por cada uno de los siguientes tipos de estamentos:

- Un estamento SELECT INTO establece FOUND a *true* si éste retorna una fila, *false* si no se retorna ninguna.
- Un estamento PERFORM establece FOUND a *true* si produce una fila, *false* en caso contrario.
- Los estamentos UPDATE, INSERT, y DELETE establecen FOUND a *true* si al menos una fila es afectada, *false* en caso contrario.
- Un estamento FETCH establece FOUND a *true* si retorna una fila, *false* en caso contrario.

- Un estamento FOR establece FOUND a *true* si éste itera una o más veces, o *false* en caso contrario. Esto se aplica a las tres variantes del estamento FOR (bucles FOR enteros, bucles de registro, y bucles de registros dinámicos). FOUND sólo se establece cuando el bucle FOR termina: dentro de la ejecución del bucle, FOUND no es modificado por el estamento FOR, aunque puede ser cambiado por la ejecución de otros estamentos dentro del cuerpo del bucle.

FOUND es una variable local; cualesquiera cambios afectarán sólo a la actual función PL/pgSQL.

6 Estructuras de Control

Las estructuras de control son probablemente la parte más útil (e importante) de PL/pgSQL. Con las estructuras de control de PL/pgSQL, puede manipular datos de PostgreSQL de forma flexible y potente.

6.1 Retornando desde una Función

```
RETURN expression;
```

RETURN con una expresión es usado para retornar desde una función PL/pgSQL que no retorna nada. La función termina y el valor de la expresión es retornado al peticionario.

Para retornar un valor compuesto (una fila), usted debe escribir una variable registro o fila como expresión. Cuando retorne una tipo escalar, cualquier expresión pude ser usada. El resultado de la expresión será automáticamente convertido al tipo de retorno de la función (si ha declarado la función para que retorne *void* (nulo), entonces la expresión puede ser omitida, y se ignorará en cualquier caso).

El valor de retorno de una función no puede ser dejado sin definir. Si el control llega al final del bloque de mayor nivel de la función sin detectar un estamento RETURN, ocurrirá un error en tiempo de ejecución.

Cuando una función PL/pgSQL es declarada para que retorne un SETOF de algún tipo, el procedimiento a seguir es algo diferente. En ese caso, los elementos individuales a retornar son especificados en comandos RETURN NEXT, y luego un comando final RETURN sin argumentos es usado para indicar que la función ha terminado su ejecución. RETURN NEXT puede ser usado tanto con tipos de datos escalares como compuestos; en el último caso, una “tabla” entera de resultados será retornada. Las funciones que usen RETURN NEXT deberían ser llamadas de la siguiente forma:

```
SELECT * FROM some_func();
```

Esto es, la función es usada como una tabla de origen en una cláusula FROM.

```
RETURN NEXT expression;
```

RETURN NEXT does not actually return from the function; it simply saves away the value of the expression (or record or row variable, as appropriate for the data type

being returned). Execution then continues with the next statement in the PL/pgSQL function. As successive RETURN NEXT commands are executed, the result set is built up. A final RETURN, which need have no argument, causes control to exit the function.

> **Note**: The current implementation of RETURN NEXT for PL/pgSQL stores the entire result set before returning from the function, as discussed above. That means that if a PL/pgSQL function produces a very large result set, performance may be poor: data will be written to disk to avoid memory exhaustion, but the function itself will not return until the entire result set has been generated. A future version of PL/pgSQL may allow users to allow users to define set-returning functions that do not have this limitation. Currently, the point at which data begins being written to disk is controlled by the SORT_MEM configuration variable. Administrators who have sufficient memory to store larger result sets in memory should consider increasing this parameter.

6.2 Condicionales

IF statements let you execute commands based on certain conditions. PL/pgSQL has four forms of IF:

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSE IF and
- IF ... THEN ... ELSIF ... THEN ... ELSE

6.2.1 IF-THEN

```
IF boolean-expression THEN
  estamentos
END IF;
```

Los estamentos IF-THEN son el formato más simple del IF. Los estamentos entre THEN y END IF serán ejecutados si la condición se cumple. En caso contrario, serán ignorados.

```
IF v_user_id <> 0 THEN
  UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

6.2.2 IF-THEN-ELSE

```
IF boolean-expression THEN
  estamentos
ELSE
  estamentos
END IF;
```

Los estamentos IF-THEN-ELSE añadidos al IF-THEN le permiten especificar un juego alternativo de estamentos que deberían ser ejecutados si la condición se evalúa a FALSE.

```
IF parentid IS NULL or parentid = '''' THEN
  return fullname;
ELSE
  return hp_true_filename(parentid) || ''/'' || fullname;
END IF;
IF v_count > 0 THEN
  INSERT INTO users_count(count) VALUES(v_count);
  RETURN ''t'';
ELSE
  RETURN ''f'';
END IF;
```

6.2.3 IF-THEN-ELSE IF

Los estamentos IF pueden anidarse, tal como en el siguiente ejemplo:

```
IF demo_row.sex = ''m'' THEN
  pretty_sex := ''man'';
ELSE
  IF demo_row.sex = ''f'' THEN
    pretty_sex := ''woman'';
  END IF;
END IF;
```

Cuando use este formato, estará anidando un estamento IF dentro de la parte ELSE de un estamento IF superior. Necesitará un estamenteo END IF por cada IF anidado, y uno para el IF-ELSE padre. Esto funciona, pero es tedioso cuando existen muchas alternativas a ser chequeadas.

6.2.4 IF-THEN-ELSIF-ELSE

```
IF boolean-expression THEN
  estamentos
[ ELSIF boolean-expression THEN
  estamentos
[ ELSIF boolean-expression THEN
  estamentos ...]]
[ ELSE
  estamentos ]
END IF;
```

IF-THEN-ELSIF-ELSE proporciona un método más conveniente de chequear muchas alternativas en un estamento. Formalmante es equivalente a comandos IF-THEN-ELSE-IF-THEN anidados, pero sólo se necesita un END IF.

Aquí tiene un ejemplo:

```
IF number = 0 THEN
  result := ''zero'';
ELSIF number > 0 THEN
  result := ''positive'';
ELSIF number < 0 THEN
  result := ''negative'';
ELSE
  -- hmm, the only other possibility is that number IS NULL
  result := ''NULL'';
END IF;
```

La sección ELSE final es opcional.

6.3 Bucles Simples

Con los estamentos LOOP, EXIT, WHILE y FOR, usted puede programar su función PL/pgSQL para repetir una serie de comandos.

6.3.1 LOOP

```
[<<etiquetal>>]
LOOP
  estamentos
END LOOP;
```

LOOP define un bucle incondicional que es repetido indefinidamente hasta que sea termiando por un estamento EXIT o RETURN. La etiqueta opcional puede ser usada por estamentos EXIT en bucles anidados para especificar qué nivel de anidación debería ser terminado.

6.3.2 EXIT

```
EXIT [ etiqueta ] [ WHEN expresión ];
```

Si no se proporciona etiqueta, el bucle más al interior es terminado y el estamento más cercano a END LOOP es ejecutado a continuación. Si se proporciona etiqueta, esta debe ser la etiqueta del bucle o bloque más actual o de nivel externo. Entonces el bucle o bloque nombrado es terminado, y el control continúa con el estamento que sigue al correspondiente END del bucle/bloque.

Si WHEN está presente, sólo ocurrirá la salida del bucle si la condición especificada es cierta, de lo contrario pasa al estamento tras EXIT.

Ejemplos:

```
LOOP
  -- algunas computaciones
  IF count > 0 THEN
     EXIT; -- exit loop
  END IF;
END LOOP;
LOOP
  -- algunas computaciones
  EXIT WHEN count > 0;
END LOOP;
BEGIN
  -- algunas computaciones
  IF stocks > 100000 THEN
     EXIT; -- ilegal. No puede usar  EXIT fuera de un  LOOP
  END IF;
END;
```

6.3.3 WHILE

```
[<<etiqueta>>]
WHILE expresión LOOP
  estamentos
END LOOP;
```

El estamento WHILE repite una secuencia de estamentos mientras que la condición se evalúe a verdadero. La condición es chequeada jsuto antes de cada entrada al cuerpo del bucle.

Por ejemplo:

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
  -- algunas computaciones aquí
END LOOP;
WHILE NOT boolean_expression LOOP
  -- algunas computaciones aquí
END LOOP;
```

6.3.4 FOR (integer for-loop)

```
[<<etiqueta>>]
FOR nombre IN [ REVERSE ] expresión .. expresión LOOP
  estamentos
END LOOP;
```

Este formato de FOR crea un bucle que itera sobre un rango de valores enteros. La variable *nombre* es automáticamente definida como de tipo *integer* y existe sólo dentro del bucle. Las dos expresiones dando el menor y mayor valores del rango son evaluadas unavez se entra en el bucle. El intervalo de iteración es de 1 normalmente, pero es -1 cuando se especifica REVERSE.

Algunos ejemplos:

```
FOR i IN 1..10 LOOP
  -- algunas expresiones aquí
  RAISE NOTICE ''i is %'',i;
END LOOP;
FOR i IN REVERSE 10..1 LOOP
  -- algunas expresiones aquí
END LOOP;
```

6.4 Bucles a través de los Resultados de una Búsqueda

Usando un tipo diferente de bucle FOR, usted puede iterar a través de los resultados de una consulta y manipular sus datos. La sintaxis es:

```
[<<etiqueta>>]
FOR registro | fila IN select_query LOOP
  estamentos
END LOOP;
```

A la variable *registro* o *fila* le son sucesivamente asignadas todas las filas resultantes de la consulta SELECT y el cuerpo del bucle es ejecutado para cada fila. Aquí tiene un ejemplo:

```
CREATE FUNCTION cs_refresh_mviews () RETURNS INTEGER AS '
DECLARE
  mviews RECORD;
BEGIN
  PERFORM cs_log(''Refreshing materialized views...'');
  FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP
     -- Ahora   "mviews" tiene un registro de la vista  cs_materialized_views
     PERFORM cs_log(''Refreshing materialized view '' || quote_ident(mviews.mv_name) || ''...'');
     EXECUTE ''TRUNCATE TABLE '' || quote_ident(mviews.mv_name);
     EXECUTE ''INSERT INTO '' || quote_ident(mviews.mv_name) || '' '' || mviews.mv_query;
  END LOOP;
  PERFORM cs_log(''Done refreshing materialized views.'');
  RETURN 1;
end;
' LANGUAGE 'plpgsql';
```

Sinel bucle es terminado por un estamento EXIT, el valor de la última fila asignada es todavía accesible tras la salida del bucle.

El estamento FOR-IN-EXECUTE es otra forma de iterar sobre registros:

```
[<<etiqueta>>]
FOR registro | fila IN EXECUTE expresión_texto LOOP
    estamentos
END LOOP;
```

Esto es como el anterior formato, excepto porque el estamento origen SELECT es especificado como una expresión de texto, la cual es evaluada y replanificada en cada entrada al bucle FOR. Esto permite al programador seleccionar la velocidad de una consulta preplanificada o la flexibilidad de una consulta dinámica, tal como con un estamento plano EXECUTE.

> **NOTA**: El intérprete PL/pgSQL actualmente ditingue los dos tipos de buclesFOR (enteros o retornadores de registros) comprobando si la variable destino mencionada justo antes del FOR ha sido declarada como variable tipo registro/fila. Si no es así, se presume que es un bucle FOR con valor entero. Esto puede causar algunos mensajes de error no intuitivos cuando el verdadero problema es, digamos, que uno se ha olvidado del nombre de la variable FOR.

7 Cursores

Además de ejecutar una consulta completa al mismo tiempo, es posible configurar un cursor que encapsule la consulta, y luego leer del resultado de la consulta unas cuantas filas al tiempo. Una razón de hacer esto es para evitar sobrecargas de memoria cuando el resultado contiene un gran número de filas (Sin embargo, los usuarios de PL/pgSQL no necesitarán normalmente preocuparse por esto, ya que los bucles FOR automáticamente usan un cursor internamente para evitar problemas de memoria). Un uso más interesante es retornar una referencia a un cursor que él ha creado, permitiento al peticionario leer las filas. Esto proporciona una forma eficiento de retornar largos juegos de filas desde funciones.

7.1 Declarando Variables Cursor

Todo el acceso a cursores en PL/pgSQL va a través de variables *cursor*, las cuales son siempre del tipo de datos especial *refcursor*. Una forma de crear una variable tipo cursor es declararla como de tipo *refcursor*. Otra forma es usar la sintaxis de declaración de cursor, la cual en general es:

```
nombre CURSOR [ ( argumentos ) ] FOR select_query ;
```

(NOTA: FOR puede ser reemplazado por IS para compatibilidad con Oracle)

Los argumentos, si los hay, son pares de tipos de datos *name* separados por comas que definen nombres a ser reemplazados por valores de parámetros en la consulta dada. Los actuales valores a sustituir para estos nombres serán especificados más tarde, cuando el cursor es abierto.

Algunos ejemplos:

```
DECLARE
  curs1 refcursor;
  curs2 CURSOR FOR SELECT * from tenk1;
  curs3 CURSOR (key int) IS SELECT * from tenk1 where unique1 = key;
```

Estas tres variables tienen el tipo de datos *refcursor*, pero la primera puede ser usada con cualquier consulta, mientras que la segunda tiene una consulta completamente especificada para trabajar con ella, y la última tiene una consulta parametrizada (*key* será reemplazado por un valor de parámetro entero cuando el cursor es abierto). La variable *curs1* será obviada, ya que no está asignada a ninguna consulta en particular.

7.2 Abriendo Cursores

Antes de que un cursor pueda ser utilizado para retornar filas, éste debe ser abierto (esta es la acción equivalente al comando SQL DECLARE CURSOR). PL/pgSQL tiene cuatro formas para el estamento OPEN, dos de las cuales usan variables *cursor* no asignadas y las otras dos usan variables *cursor* asignadas.

7.2.1 OPEN FOR SELECT

```
OPEN unbound-cursor FOR SELECT ...;
```

La variable *cursor* es abierta y se le pasa la consulta especificada para ejecutar. El cursor no puede ser abierto aún, y debe haber sido declarado como cursor no asignado (esto es, una simple variable *refcursor*). La consulta SELECT es tratada de la misma forma que otros estamentos SELECT en PL/pgSQL: los nombres de variables PL/pgSQL son sustituidos, y el plan de consulta es almacenado para su posible reutilización.

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

7.2.2 OPEN FOR EXECUTE

```
OPEN unbound-cursor FOR EXECUTE query-string;
```

La variable cursor es abierta y se le pasa la consulta especificada para ser ejecutada. El cursor no puede ser abierto aún, y debe haber sido decalrado como cursor no asignado (esto es, una simpla variable *refcursor*). La consulta es especificada como una expresión de texto de la misma forma que en el comando EXECUTE. Como es usual, esto le da flexibilidad para que la consulta pueda variar en cada ejecución.

```
OPEN curs1 FOR EXECUTE ''SELECT * FROM '' || quote_ident($1);
```

7.2.3 Opening a bound cursor

```
OPEN bound-cursor [ ( argument_values ) ];
```

Esta forma de OPEN es usada para abrir una variable cursor cuya consulta le fué asignada cuando ésta fue declarada. El cursor no puede ser abierto todavía. Una lista de los actuales argumentos de los valores de las expresiones debe aparecer si y sólo si el cursor fue declarado para tomar argumentos. Estos valores serán sustituidos en la consulta. El plan de consulta para un cursor asignado siempre es considerado como almacenable -no hay equivalencia para EXECUTE en éste caso-.

```
OPEN curs2;
OPEN curs3(42);
```

7.3 Usando Cursores

Una vez un cursor ha sido abierto, éste puede ser manipulado con los estamentos descritos aquí.

Estas manipulaciones no necesitan ocurrir en la misma función que abrió el cursor. Puede retornar un valor *refcursor* de una función y permitir al peticionario operar con el cursor (internamente, un valor *refcursor* es simplemente la cadena nombre de una Portal conteniendo la consulta activa para el cursor. Este nombre puede ser pasado, asignado a otras variables *refcursor*, sin afectar al Portal) .

Todos los Portales son implícitamente cerrados al final de la transacción. Por tanto un valor *refcursor* es útil para referenciar un cursor abierto sólo hasta el final de la transacción.

7.3.1 FETCH

```
FETCH cursor INTO target;
```

FETCH retorna la siguiente fila desde el cursor a un destino, el cual puede ser una variable fila, registro o una lista de vairbales simples, separadas por comas, tal como en un SELECT INTO. Al igual que en un SELECT INTO, la variable especial FOUND puede ser chequeada para ver si se obutvo o no una fila.

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo,bar,baz;
```

7.3.2 CLOSE

```
CLOSE cursor;
```

CLOSE cierra el Portal de un cursor abierto. Esto puede ser suado para liberar recursos antes del final de una transacción, o para liberar la variable cursor para abrirla de nuevop.

```
CLOSE curs1;
```

7.3.3 Retornando Cursores

Las funciones PL/pgSQL pueden retornar cursores al peticionario. Esto es usado para retornar múltiples filas o columnas desde la función. La función abre el cursor y retorna el nombre del cursor al peticionario. El peticionario entonces puede obtener (con FETCH) filas desde el cursor. El cursor puede ser cerrado por el remitente, o será cerrado automáticamente cuando termine la transacción.

El nombre del cursor retornado por la función puede ser especificado por el peticionario o generado automáticamente. El siguiente ejemplo muestra cómo un nombre de cursor puede ser proporcionado por el peticionario:

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');
CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
  OPEN $1 FOR SELECT col FROM test;
  RETURN $1;
END;
' LANGUAGE 'plpgsql';
BEGIN;
  SELECT reffunc('funccursor');
  FETCH ALL IN funccursor;
  COMMIT;
```

El siguiente ejemplo usa generación automática de nombre de cursor:

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
  ref refcursor;
BEGIN
  OPEN ref FOR SELECT col FROM test;
  RETURN ref;
END;
' LANGUAGE 'plpgsql';
BEGIN;
  SELECT reffunc2();
  ----------------------
   <unnamed cursor 1>
(1 row)
FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

8 Errores y Mensajes

Use el estamento RAISE para retornar mensajes y lanzar errores.

```
RAISE level 'format' [, variable [...]];
```

Los posibles niveles son DEBUG (escribe el mensaje en el registro del servidor), LOG (escribe el mensaje en el servidor con una prioridad alta), INFO, NOTICE y WARNING (escribe el mensaje en el registro del servidor y lo envía al cliente, con prioridad alta en ambos envíos), y EXCEPTION (lanza un error y aborta la actual transacción). Si los mensajes de error de una determinada prioridad son reportados al cliente, escritos en el registro del servidor, o ambos es es controlado por las variables de configuración LOG_MIN_MESSAGES y CLIENT_MIN_MESSAGES. Vea la Guía del Adminitrador de PostgreSQL para más información.

Dentro del formato de la cadena, % es reemplazado por el siguiente argumento opcional de la representación externa. Escriba %% para emitir un literal %. Advierta que los argumentos opcionales deben ser variables simples, no expresiones, y el formato debe ser un simple literal.

Ejemplos:

```
RAISE NOTICE ''Calling cs_create_job(%)'',v_job_id;
```

En éste ejemplo, el valor de *v_job_id* reemplazará el % en la cadena.

```
RAISE EXCEPTION ''Inexistent ID --> %'',user_id;
```

Esto abortará la transacción con el mensaje de error dado.

8.1 Excepciones

PostgreSQL no tiene un modelo de manejo de excepciones demasiado elegante. Cuando el intérprete, planificador/optimizador o ejecutor decide que el estamento no puede ser procesado, toda la transacción es abortada y el sistema salta al bucle principal para obtener la siguiente consultad desde la aplicación cliente.

Es posible entrar en el mecanismo de error para averiguar qué es lo que está pasando. Pero actualmente es imposible decir qué es lo que realmente causó la cancelación (error de conversión de entrada/salida, error de coma flotante, error de sintaxis). Y es posible que el motor de base de datos esté en un estado inconsistente en éste punto, así que podría retornar al ejecutor superior o la entrega de más comandos podría corromper toda la base de datos.

Así, la única cosa que hace PL/pgSQL actualmente cuando se encuentra con una cancelación durante la ejecución de una función o trigger es escribir adicionales mensajes de nivel NOTICE, indicando en qué función y cuándo (número de línea y tipo de estamento) ocurrió. El error siempre para la ejecución de la función.

9 Procedimientos Trigger

PL/pgSQL puede ser utilizado para definir triggers. Un procedimiento trigger es creado con el comando CREATE FUNCTION como una función sin argumentos y retornando un tipo *trigger*. Advierta que la función debe ser declarada sin argumentos, aunque espere recibir argumentos específicos en CREATE TRIGGER -los argumentos trigger son pasados vía TG_ARGV, como veremos más adelante-.

Cuando una función PL/pgSQL es denominada como *trigger*, varias variables especiales son creadas automáticamente en el bloque de nivel superior. Estas son:

NEW Tipo de datos RECORD; variable que almacena la nueva fila de base de datos para operaciones INSERT/UPDATE en triggers de nivel ROW. Esta variable es NULL en los triggers de nivel STATEMENT.

OLD Tipo de datos RECORD; variable que almacena la antigua fila de base de datos para operaciones UPDATE/DELETE en triggers de nivel ROW. Esta variable es NULL en triggers de nivel STATEMENT.

TG_NAME Tipo de datos *name*; variable que contiene el nombre del trigger actualmente disparado.

TG_WHEN Tipo de datos *text*; una cadena de BEFORE o AFTER dependiendo de la definición del trigger.

TG_LEVEL Tipo de datos *text*; una cadena de ROW o STATEMENT dependiendo de la definición del trigger.

TG_OP Tipo de datos *text*; una cadena de INSERT, UPDATE o DELETE indicando por cuál operación se disparó el trigger.

TG_RELID Tipo de datos *oid*; el ID de objeto de la tabla que causó la invocación del trigger.

TG_RELNAME Tipo de datos *name*; el nombre de la tabla que causó la invocación del trigger.

TG_NARGS Tipo de datos *integer*; el número de argumentos proporcionado al procedimiento trigger en el estamento CREATE TRIGGER.

TG_ARGV[] Tipo de datos *array* de *text*; los argumentos del estamento CREATE TRIGGER. El índice cuenta desde 0 y puede ser dado como expresión. Índices inválidos (< 0 or >= tg_nargs) resultan en un valor nulo.

Una función trigger debe retornar o NULL o un valor registro/fila teniendo exactamente la misma estructura de la tabla que disparó el trigger. El valor de retorno de un trigger de nivel BEFORE o AFTER STATEMENT, o un trigger de nivel AFTER ROW es ignorado; también puede retornar NULL. Sin embargo, cualquiera de estos tipos aún pueden abortar toda la operación del trigger lanzando un error.

Los triggers de nivel ROW lanzados antes (BEFORE) pueden retornar NULL para señalar al gestor del trigger que se salte el resto de la operación para esa fila (p.ej., subsecuentes triggers no son lanzados, y el INSERT/UPDATE/DELETE no ocurre para esa fila). Si un valor no NULL es retornado, entonces la operación procede con ese valor de fila. Advierta que retornar un valor de fila diferente del valor original del nuevo (NEW) altera la fila que será insertada o actualizada. Es posible reemplazar valores simples directamente en NEW y retornarlos, o construir un registro/fila completamente nuevo para retornarlo.

Ejemplo 19-1. Un Ejemplo de Procedimiento Trigger en PL/pgSQL

Este trigger asegura que en cualquier momento en que una fila sea insertada o actualizada en la tabla, el actual nombre de usuario y fecha/hora son almacenadas en la fila. Y garantiza que un nombre de empleado es dado y que el *salario* es un valor positivo.

```
CREATE TABLE emp (
       nombre_empleado text,
       salario         integer,
       ultima_fecha    timestamp,
       ultimo_usuario  text );
CREATE FUNCTION emp_stamp () RETURNS TRIGGER AS '
BEGIN
  -- Comprueba que se proporcionan nombre_empleado y salario
  IF NEW.nombre_empleado ISNULL THEN
     RAISE EXCEPTION ''El nombre del empleado no puede ser un valor NULO'';
  END IF;
  IF NEW.salario ISNULL THEN
     RAISE EXCEPTION ''% no puede tener un salario NULO'', NEW.nombre_empleado;
  END IF;
  -- ¿Quién trabaja gratis?
  IF NEW.salario < 0 THEN
     RAISE EXCEPTION ''% no puede tener un salario negativo'', NEW.nombre_empleado;
  END IF;
  -- Recuerda quién y cuándo hizo el cambio
  NEW.ultima_fecha   := ''now'';
  NEW.ultimo_usuario := current_user;
  RETURN NEW;
END;
' LANGUAGE 'plpgsql';
CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
       FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

10 Ejemplos

Aquí tiene unas cuantas funciones para demostrar lo sencillo que es escribir funciones PL/pgSQL. Para ejemplos más complejos el programador podría echar un vistazo a los tests de regresión para PL/pgSQL.

Un detalle importante en la escritura de funciones PL/pgSQL es el manejo de las comillas simples. El código fuente de la función en CREATE FUNCTION debe ser un literal de cadena. Las comillas simples dentro de literales cadena deben ser dobles o antecedidas por una barra invertida (*backslash*). Todavía seguimos buscando una alternativa elegante. Por ahora, doblar las comillas simples como en los ejemplos es lo que debería usar. Cualquier solución para futuras versiones de PostgreSQL será compatible.

Para una explicación detallada y ejemplos sobre cómo escapar comillas simples en diferentes situaciones, vea la Sección 11.1.1.

Ejemplo 19-2. Una simple función PL/pgSQL para incrementar un entero.

Esta función recibe un entero y lo incrementa en uno, retornando el valor incrementado.

```
CREATE FUNCTION add_one (integer) RETURNS INTEGER AS '
BEGIN
  RETURN $1 + 1;
END;
' LANGUAGE 'plpgsql';
```

Ejemplo 19-3. Una simple función PL/pgSQL para concatenar texto.

Esta función recibe dos parámetros de texto y retorna el resultado de su unión.

```
CREATE FUNCTION concat_text (TEXT, TEXT) RETURNS TEXT AS '
BEGIN
  RETURN $1 || $2;
END;
' LANGUAGE 'plpgsql';
```

Ejemplo 19-4. Una función PL/pgSQL para composición de texto.

En éste ejemplo, tomamos EMP (una tabla) y un entero com argumentos de nuestra función, la cual retorna un booleano. Si el campo *salary* de la tabla EMP es NULL, nosotros retornamos *f*. De lo contrario comparamos dicho campo con el entero pasado a la función y retornamos el resultado booleano de la comparación (*t* o *f*).

```
CREATE FUNCTION c_overpaid (EMP, INTEGER) RETURNS BOOLEAN AS '
DECLARE
  emprec ALIAS FOR $1;
  sallim ALIAS FOR $2;
BEGIN
  IF emprec.salary ISNULL THEN
     RETURN ''f'';
  END IF;
  RETURN emprec.salary > sallim;
END;
' LANGUAGE 'plpgsql';
```

11 Portando desde Oracle PL/SQL

Autor: Roberto Mello (<rmello@fslc.usu.edu>)

Esta sección explica las diferencias entre el PL/SQL de Oracle y el PL/pgSQL de PostgreSQL con el objetivo de ayudar a los programadores a portar aplicaciones de Oracle a PostgreSQL. La mayoría del código expuesto aquí es del módulo *ArsDigita Clickstream* que yo (el autor) porté a PostgreSQL cuando desarrollé una tienda interna con OpenForce Inc. en el verano de 2000.

PL/pgSQL es similar a PL/SQL en muchos aspectos. Es un lenguaje de bloques estructurados, imperativo (todas las variables tienen que ser declaradas). PL/SQL tiene muchas más características que su homónimo de PostgreSQL, pero PL/pgSQL permite una gran funcionalidad y es mejorado constantemente.

11.1 Principales Diferencias

Algunas cosas que debería recordar cuando porte de Oracle a PostgreSQL:

- No hay parámetros por defecto en PostgreSQL.
- Puede volver a cargar funciones en PostgreSQL. Esto es frecuentemente usado para saltarse la deficiencia de los parámetros por defecto.
- Las asignaciones, bucles y condicionales son similares.
- No es necesario para los cursores en PostgreSQL, sólo ponga la consulta en el estamento FOR (vea ejemplo más adelante).
- En PostgreSQL necesita escapar las comillas simples. Vea la Sección 11.1.1.

11.1.1 Escapando Comillas Simples

En PostgreSQL necesita escapar las comillas simples dentro de su definición de función. Esto es importante recordarlo para el caso de crear funciones que generan otras funciones, como en el Ejemplo 19-6.

La Tabla 19-1 le ayudará (amará a esta pequeña tabla).

Tabla 19-1. Esquema de Escapado de Comillas Simples.

No. de Comillas	Uso	Ejemplo	Resultado
1	Para iniciar/terminar cuerpos de función	CREATE FUNCTION foo() RETURNS INTEGER AS '...' LANGUAGE 'plpgsql';	tal cual
2	En asignaciones, estamentos SELECT, para delimitar cadenas, etc.	a_output := "Blah"; SELECT * FROM users WHERE f_name="foobar";	SELECT * FROM users WHERE f_name='foobar';
4	Cuando necesite dos comillas simples en su cadena de resultado sin terminar dicha cadena.	a_output := a_output || " AND name LIKE ""foobar"" AND ..."	AND name LIKE 'foobar' AND ...
6	Cuando quiera comillas dobles en su cadena de resultad y terminar dicha cadena.	a_output := a_output || " AND name LIKE ""foobar"""	AND name LIKE 'foobar'
10	Cuando quiera dos comillas simples en la cadena de resultado (lo cual cuenta para 8 comillas) y terminar dicha cadena (2 más). Probablemente sólo necesitará esto si usa una función para generar otras funciones (como en el Ejemplo 19-6).	a_output := a_output || " if v_" || referrer_keys.kind || " like """""""" || referrer_keys.key_string || """""""" then return """" || referrer_keys.referrer_type || """"; end if;";	if v_<...> like "<...>" then return "<...>"; end if;

11.2 Portando Funciones

Ejemplo 19-5. Una Función Simple.

Aquí tiene una función de Oracle:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name IN varchar, v_version IN varchar)
RETURN varchar IS
BEGIN
  IF v_version IS NULL THEN
     RETURN v_name;
  END IF;
  RETURN v_name || '/' || v_version;
END;
/
SHOW ERRORS;
```

Vayamos a través de la función para ver las diferencias con PL/pgSQL:

- PostgreSQL ni tiene parámetros nominados. Usted tiene que especificarlos como ALIAS dentro de su función.
- Oracle puede tener parámetros IN, OUT, y INOUT pasados a funciones. El INOUT, por ejemplo, significa que el parámetro recibirá un valor y retornará otro. PostgreSQL sólo tiene parámetros "IN" y las funciones sólo pueden retornar un valor simple.

- La palabra clave RETURN en la función *prototype* (no la función *body*) retornar RETURNS en PostgreSQL.
- Las funciones PostgreSQL son creadas usando comillas simples y delimitadores, así que tiene que escapar comillas simples dentro de sus funciones (lo cual puede ser engorroso en algunas ocasiones; vea la Sección 11.1.1).
- El comando */show errors* no existe en PostgreSQL.

Así que veamos cómo ésta función quedaría al portarla a PostgreSQL:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(VARCHAR, VARCHAR)
RETURNS VARCHAR AS '
DECLARE
  v_name ALIAS FOR $1;
  v_version ALIAS FOR $2;
BEGIN
  IF v_version IS NULL THEN
     return v_name;
  END IF;
  RETURN v_name || ''/'' || v_version;
END;
' LANGUAGE 'plpgsql';
```

Ejemplo 19-6. Una Función que crea otra Función.

El siguiente procedimiento graba filas desde un estamento SELECT y construye una gran función con los resultados en estamentos IF, para una mayor eficiencia. Advierta particularmente las diferencias en cursores, bucles FOR, y la necesidad de escapar comillas simples en PostgreSQL.

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
  CURSOR referrer_keys IS
    SELECT * FROM cs_referrer_keys
    ORDER BY try_order;
a_output VARCHAR(4000);
BEGIN
  a_output := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR, v_domain IN VARCHAR,
v_url IN VARCHAR) RETURN VARCHAR IS BEGIN';
  FOR referrer_key IN referrer_keys LOOP
      a_output := a_output || ' IF v_' || referrer_key.kind || ' LIKE ''' ||
                  referrer_key.key_string || ''' THEN RETURN ''' || referrer_key.referrer_type ||
                  '''; END IF;';
  END LOOP;
  a_output := a_output || ' RETURN NULL; END;';
  EXECUTE IMMEDIATE a_output; END;
/
show errors
```

Aquí tiene cómo quedaría la función en PostgreSQL:

```
CREATE FUNCTION cs_update_referrer_type_proc() RETURNS INTEGER AS '
DECLARE
  referrer_keys RECORD; -- Declare a generic record to be used in a FOR
  a_output varchar(4000);
BEGIN
  a_output := ''CREATE FUNCTION cs_find_referrer_type(VARCHAR,VARCHAR,VARCHAR)
                RETURNS VARCHAR AS ''''
                DECLARE
                  v_host ALIAS FOR $1;
                  v_domain ALIAS FOR $2;
                  v_url ALIAS FOR $3;
                BEGIN '';
  --
  -- Advierta cómo escaneamos a través de los resultados de una consulta en un bucle  FOR
  -- usando el constructor  FOR <registro>.
  --
  FOR referrer_keys IN SELECT * FROM cs_referrer_keys ORDER BY try_order LOOP
```

```
      a_output := a_output || '' IF v_'' || referrer_keys.kind || '' LIKE ''''''''''
                  || referrer_keys.key_string || '''''''''' THEN RETURN ''''''
                  || referrer_keys.referrer_type || ''''''; END IF;'';
  END LOOP;
  a_output := a_output || '' RETURN NULL; END; '''' LANGUAGE ''''plpgsql'''';'';
  -- Esto funciona porque no estamos sustituyendo variables.
  -- De lo contrario fallaría. Mire  PERFORM para otra forma de ejecutar funciones.
  EXECUTE a_output;
END;
' LANGUAGE 'plpgsql';
```

Ejemplo 19-7. Un Procedimiento con un montón de manipulación de cadenas y parámetros OUT.

El siguiente procedimiento de Oracle PL/SQL es usado para interpretar una URL y retornar varios elementos (máquina, ruta y consulta). Es un procedimiento porque en las funciones PL/pgSQL sólo puede ser retornado un valor (vea la Sección 11.3). En PostgreSQL, una forma de salvar esto es dividir el procedimiento en tres funciones diferentes: una para retornar el nombre de máquina, otra para la ruta y otra para la consulta.

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
  v_url IN VARCHAR,
  v_host OUT VARCHAR,  -- This will be passed back
  v_path OUT VARCHAR,  -- This one too
  v_query OUT VARCHAR) -- And this one
is
  a_pos1 INTEGER;
  a_pos2 INTEGER;
begin
  v_host := NULL;
  v_path := NULL;
  v_query := NULL;
  a_pos1 := instr(v_url, '//'); -- PostgreSQL not tiene una función  instr
  IF a_pos1 = 0 THEN
     RETURN;
  END IF;
  a_pos2 := instr(v_url, '/', a_pos1 + 2);
  IF a_pos2 = 0 THEN
     v_host := substr(v_url, a_pos1 + 2);
     v_path := '/'; RETURN;
  END IF;
  v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
  a_pos1 := instr(v_url, '?', a_pos2 + 1);
  IF a_pos1 = 0 THEN
     v_path := substr(v_url, a_pos2);
     RETURN;
  END IF;
  v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
  v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

Aquí tiene cómo éste procedimiento podría ser trasladado a PostgreSQL:

```
CREATE OR REPLACE FUNCTION cs_parse_url_host(VARCHAR) RETURNS VARCHAR AS '
DECLARE
  v_url ALIAS FOR $1;
  v_host VARCHAR;
  v_path VARCHAR;
  a_pos1 INTEGER;
  a_pos2 INTEGER;
  a_pos3 INTEGER;
BEGIN
  v_host := NULL;
  a_pos1 := instr(v_url,''//'');
  IF a_pos1 = 0 THEN
```

```
    RETURN ''''; -- Return a blank
  END IF;
  a_pos2 := instr(v_url,''/'',a_pos1 + 2);
  IF a_pos2 = 0 THEN
     v_host := substr(v_url, a_pos1 + 2);
     v_path := ''/'';
     RETURN v_host;
  END IF;
  v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2 );
  RETURN v_host;
END;
' LANGUAGE 'plpgsql';
```

NOTA: PostgreSQL no tiene una función *instr*, así que tendrá que solventarlo usando una combinación de otras funciones. Yo me creé mis propias funciones *instr* que hacen exactamente lo mismo que las de Oracle. Vea la Sección 11.6 para el código.

11.3 Procedimientos

Los procedimientos de Oracle le dan al programador más flexibilidad porque nada necesita ser explícitamente retornado, pero esto mismo se puede hacer a través de los parámetros INOUT o OUT.

Un ejemplo:

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
  a_running_job_count INTEGER;
  PRAGMA AUTONOMOUS_TRANSACTION;(1)
BEGIN
  LOCK TABLE cs_jobs IN EXCLUSIVE MODE;(2)
  SELECT count(*) INTO a_running_job_count
  FROM cs_jobs
  WHERE end_stamp IS NULL;
  IF a_running_job_count > 0 THEN
     COMMIT; -- free lock(3)
     raise_application_error(-20000, 'Unable to create a new job: a job is currently running.');
  END IF;
  DELETE FROM cs_active_job;
  INSERT INTO cs_active_job(job_id) VALUES (v_job_id);
  BEGIN
    INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, sysdate);
    EXCEPTION WHEN dup_val_on_index THEN NULL; -- don't worry if it already exists(4)
  END;
  COMMIT;
END;
/
show errors
```

Los procedimientos como éste pueden ser fácilmente convertidos a funciones PostgreSQL retornando un INTEGER. Este procedimiento en particular es interesante porque nos puede enseñar algunas cosas:

(1) No hay ningún estamento pragmático en PostgreSQL.

(2) Si usted realiza un LOCK TABLE en PL/pgSQL, el bloqueo (lock) no será liberado hasta que la transacción no termine.

(3) Tampoco puede tener transacciones en procedimientos PL/pgSQL. Toda la función (y otras funciones llamadas desde ésta) es ejecutada en una transacción, y PostgreSQL retira los resultados si algo va mal. Por lo tanto sólo está permitido un estamento BEGIN.

(4) La excepción a esto es cuando tuviera que ser reemplazado por un estamento IF.

Así que veamos una de las formas de portar éste procedimiento a PL/pgSQL:

```
CREATE OR REPLACE FUNCTION cs_create_job(INTEGER) RETURNS INTEGER AS '
DECLARE
  v_job_id ALIAS FOR $1;
  a_running_job_count INTEGER;
  a_num INTEGER;
  -- PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  LOCK TABLE cs_jobs IN EXCLUSIVE MODE;
  SELECT count(*) INTO a_running_job_count
  FROM cs_jobs
  WHERE end_stamp IS NULL;
  IF a_running_job_count > 0 THEN
     -- COMMIT; -- free lock
     RAISE EXCEPTION ''Unable to create a new job: a job is currently running.'';
  END IF;
  DELETE FROM cs_active_job;
  INSERT INTO cs_active_job(job_id) VALUES (v_job_id);
  SELECT count(*) into a_num
  FROM cs_jobs
  WHERE job_id=v_job_id;
  IF NOT FOUND THEN
     -- If nothing was returned in the last query
     -- This job is not in the table so lets insert it.
     INSERT INTO cs_jobs(job_id, start_stamp) VALUES (v_job_id, sysdate());
     RETURN 1;
  ELSE
     RAISE NOTICE ''Job already running.'';(1)
  END IF;
  RETURN 0;
END;
' LANGUAGE 'plpgsql';
```

(1) Advierta que puede enviar noticias (o errores) in PL/pgSQL.

11.4 Paquetes

NOTA: No he hecho mucho con los paquetes, así que si detecta errores hágamelo saber.

Los paquetes son una forma que Oracle le proporciona para encapsular estamentos y funciones PL/SQL en una entidad, tal como las clases de Java, donde usted define métodos y objetos. Usted puede acceder a estos objetos/métodos con un punto "." (dot). Aquí tiene un ejemplo de paquete Oracle de ACS 4 (el *ArsDigital Community System*):

```
CREATE OR REPLACE PACKAGE BODY acs
AS
  FUNCTION add_user (
    user_id       IN users.user_id%TYPE DEFAULT NULL,
    object_type   IN acs_objects.object_type%TYPE DEFAULT 'user',
    creation_date IN acs_objects.creation_date%TYPE DEFAULT sysdate,
    creation_user IN acs_objects.creation_user%TYPE DEFAULT NULL,
    creation_ip   IN acs_objects.creation_ip%TYPE DEFAULT NULL,
    ...
  ) RETURN users.user_id%TYPE
  IS
    v_user_id     users.user_id%TYPE;
    v_rel_id      membership_rels.rel_id%TYPE;
  BEGIN
    v_user_id := acs_user.new (user_id, object_type, creation_date,
                 creation_user, creation_ip, email, ...
    RETURN v_user_id;
  END;
END acs;
/
show errors
```

Nosotros portaremos esto a PostgreSQL creando los diferentes objetos del paquete Oracle como funciones con una convención de nombres standard. Tenemos que poner atención en algunos detalles, tales como la falta de parámetros por defecto en las funciones PostgreSQL. El paquete anterior sería algo como esto:

```
CREATE FUNCTION acs__add_user(INTEGER,INTEGER,VARCHAR,TIMESTAMP,INTEGER,INTEGER,...)
RETURNS INTEGER AS '
DECLARE
  user_id ALIAS FOR $1;
  object_type ALIAS FOR $2;
  creation_date ALIAS FOR $3;
  creation_user ALIAS FOR $4;
  creation_ip ALIAS FOR $5;
  ...
  v_user_id users.user_id%TYPE;
  v_rel_id membership_rels.rel_id%TYPE;
BEGIN
  v_user_id := acs_user__new(user_id,object_type,creation_date,creation_user,creation_ip, ...);
  ...
  RETURN v_user_id;
END;
' LANGUAGE 'plpgsql';
```

11.5 Otras Cosas por Ver

11.5.1 EXECUTE

La versión PostgreSQL de EXECUTE funciona de maravilla, pero tiene que recordar usar literales entrecomillados (TEXT) y cadenas entrecomilldas (TEXT) tal como se describe en la Sección 5.4. Los constructores del tipo *EXECUTE "SELECT * from $1";* no funcionarán a menos que use estas funciones.

11.5.2 Optimizando las Funciones PL/pgSQL

PostgreSQL le da dos modificadores de creación de funciones para optimizar la ejecución: *iscachable* (esta función siempre retorna el mismo resultado cuando se le dan los mismos argumentos) y *isstrict* (la función retorna NULL si cualquiera de los argumentos es NULL). Consulte la referencia de CREATE FUNCTION para más detalles.

Para hacer uso de estos atributos de optimización, usted tiene que usar el modificador WITH en su estamento CREATE FUNCTION. Algo como:

```
CREATE FUNCTION foo(...) RETURNS INTEGER AS '
...
' LANGUAGE 'plpgsql' WITH (isstrict, iscachable);
```

11.6 Apéndice

11.6.1 Código para mis funciones *instr*

Esta función podría probablemente ser integrada.

```
--
-- función instr que imita a la de  Oracle
-- Sintaxis: instr(string1,string2,[n],[m]) donde [] denota parámetros opcionales.
--
-- Busca string1 empezando por el carácter  n para la ocurrencia  m
-- de string2. Si n es negativa, busca hacia atrás. Si m no es proporcionada,
-- asume 1  (la búsqueda se inicia en el primer carácter).
--
-- por Roberto Mello (rmello@fslc.usu.edu)
-- modificada por  Robert Gaszewski (graszew@poland.com)
-- Licenciada bajo la  GPL v2 o siguientes.
--
```

```
CREATE FUNCTION instr(VARCHAR,VARCHAR) RETURNS INTEGER AS '
DECLARE
  pos integer;
BEGIN
  pos:= instr($1,$2,1);
RETURN pos;
END;
' LANGUAGE 'plpgsql';
CREATE FUNCTION instr(VARCHAR,VARCHAR,INTEGER) RETURNS INTEGER AS '
DECLARE
  string ALIAS FOR $1;
  string_to_search ALIAS FOR $2;
  beg_index ALIAS FOR $3;
  pos integer NOT NULL DEFAULT 0;
  temp_str VARCHAR;
  beg INTEGER;
  length INTEGER;
  ss_length INTEGER;
BEGIN
  IF beg_index > 0 THEN
     temp_str := substring(string FROM beg_index);
     pos := position(string_to_search IN temp_str);
    IF pos = 0 THEN
       RETURN 0;
    ELSE
       RETURN pos + beg_index - 1;
    END IF;
  ELSE
    ss_length := char_length(string_to_search);
    length := char_length(string);
    beg := length + beg_index - ss_length + 2;
    WHILE beg > 0 LOOP
      temp_str := substring(string FROM beg FOR ss_length);
      pos := position(string_to_search IN temp_str);
      IF pos > 0 THEN
         RETURN beg;
      END IF;
      beg := beg - 1;
    END LOOP;
    RETURN 0;
  END IF;
END;
' LANGUAGE 'plpgsql';
--
-- modificada por  Robert Gaszewski (graszew@poland.com)
-- Licenciada bajo la  GPL v2 o siguientes.
--
CREATE FUNCTION instr(VARCHAR,VARCHAR,INTEGER,INTEGER) RETURNS INTEGER AS '
DECLARE
  string ALIAS FOR $1;
  string_to_search ALIAS FOR $2;
  beg_index ALIAS FOR $3;
  occur_index ALIAS FOR $4;
  pos integer NOT NULL DEFAULT 0;
  occur_number INTEGER NOT NULL DEFAULT 0;
  temp_str VARCHAR;
  beg INTEGER;
  i INTEGER;
  length INTEGER;
  ss_length INTEGER;
BEGIN
  IF beg_index > 0 THEN
     beg := beg_index;
     temp_str := substring(string FROM beg_index);
     FOR i IN 1..occur_index LOOP
         pos := position(string_to_search IN temp_str);
         IF i = 1 THEN
            beg := beg + pos - 1;
         ELSE
            beg := beg + pos;
         END IF;
         temp_str := substring(string FROM beg + 1);
     END LOOP;
     IF pos = 0 THEN
        RETURN 0;
     ELSE
        RETURN beg;
```

```
    END IF;
  ELSE
    ss_length := char_length(string_to_search);
    length := char_length(string);
    beg := length + beg_index - ss_length + 2;
    WHILE beg > 0 LOOP
          temp_str := substring(string FROM beg FOR ss_length);
          pos := position(string_to_search IN temp_str);
          IF pos > 0 THEN
             occur_number := occur_number + 1;
             IF occur_number = occur_index THEN
                RETURN beg;
             END IF;
          END IF;
          beg := beg - 1;
    END LOOP;
    RETURN 0;
  END IF;
END;
' LANGUAGE 'plpgsql';
```