Tabla de contenidos

Capítulo 1. El Lenguaje Procedimental SQL - PL/pgSQL

Nota: Este documento fue traducido y adaptado por Roberto Andrade Fonseca (randradefonseca@gmail.com), tomando como base el capítulo llamado 'PL/pgSQL - SQL Procedural Language' de la documentación de PostgreSQL, versión 8.04, en octubre de 2005. *Esta es una versión reducida del capítulo mencionado*.

PL/pgSQL es un lenguaje procedimental cargable para el sistema de base de datos PostgreSQL. Los objetivos propuestos para PL/pgSQL consisten en crear un lenguaje procedimental cargable que

- pueda ser usado para crear funciones y procedimientos disparadores,
- adicione estructuras de control al lenguaje SQL,
- sea capaz de realizar cálculos complejos,
- herede todos los tipos, las funciones y los operadores definidos por el usuario,
- pueda ser definido como confiable (trusted) por el servidor,
- sea fácil de usar.

Excepto por las conversiones de entrada/salida y las funciones de cálculo para los tipos definidos por el usuario, todo lo que puede definirse por medio de funciones en el lenguaje C puede definirse también con PL/pgSQL. Por ejemplo, es posible crear funciones computacionales condicionales complejas que pueden ser usadas posteriormente para definir operadores o usarlas en expresiones asociadas a los índices.

Visión General

El manejador de llamadas de PL/pgSQL analiza sintácticamente el texto del código fuente de la función y produce un árbol de instrucciones binario interno la primera vez que se llama una función (dentro de cada sesión). El árbol de instrucciones traduce completamente la estructura de los comandos PL/pgSQL, pero las expresiones individuales SQL y los comandos SQL usados en las funciones no son traducidas de inmediato.

Cada vez que es usado por primera vez un comando SQL en la función el intérprete PL/pgSQL crea un plan preparado de ejecución (usando las funciones SPI_prepare y SPI_saveplan del administrador SPI). Las visitas subsecuentes a esa expresión o comando reutilizan el plan preparado. Así, una función con código condicional que contiene varias sentencias para las cuales se requiere del plan de ejecución, solamente preparará y guardará aquellos planes que realmente sean usados durante el ciclo de vida de la conexión de la base de datos. Esto puede reducir de manera importante el tiempo total requerido para revisar y generar los planes de ejecución para las sentencias en una función de PL/pgSQL. Una desventaja es que aquellos errores de un comando o expresión específica pueden no ser detectados hasta que esa parte de la función sea alcanzada en la ejecución.

Una vez que PL/pgSQL haya generado un plan de ejecución para un comando particular en una función, éste será reutilizado durante el ciclo de vida de la conexión de la base de datos. Generalmente esto es una ventaja para el desempeño, pero puede causar algunos problemas si se modifica dinámicamente el esquema de la base de datos. Por ejemplo:

```
CREATE FUNCTION populate() RETURNS integer AS $$
DECLARE
```

```
    -- declaraciones
BEGIN
    PERFORM my_function();
END;
$$ LANGUAGE plpgsql;
```

Si ejecuta la función anterior, ésta hará referencia al OID de `my_function()` en el plan de ejecución generado para el comando **PERFORM**. Más tarde, si elimina (drop) y recrea `my_function()`, entonces `populate()` no podrá encontrar de ninguna manera `my_function()`. Tendría que recrear `populate()`, o al menos iniciar una nueva sesión de la base de datos para que se recompile nuestra función. Otra manera de evitar este problema es usar **CREATE OR REPLACE FUNCTION** cuando se actualice la definición de `my_function` (cuando una función es "reemplazada", su OID no sufre cambios).

Debido a que `PL/pgSQL` guarda los planes de ejecución de esta manera, los comandos SQL que aparecen directamente en una función `PL/pgSQL` deben referirse a las mismas tablas y columnas en cada ejecución; es decir, usted no puede usar un parámetro como el nombre de una tabla o una columna en un comando SQL. Para darle la vuelta a esta restricción, puede construir comandos dinámicos usando la sentencia **EXECUTE** de `PL/pgSQL` — pagando el precio de generar un nuevo plan de ejecución en cada ejecución.

> **Nota:** La sentencia **EXECUTE** de `PL/pgSQL` no está relacionada con el comando SQL que es soportado por el servidor PostgreSQL. La sentencia **EXECUTE** del servidor no puede usarse dentro de las funciones `PL/pgSQL` (y no es necesaria).

Ventajas del Uso de `PL/pgSQL`

SQL es el lenguaje que PostgreSQL y la mayoría de la bases de datos relacionales usan como lenguaje de consulta. Es portable y fácil de aprender. Pero cada sentencia SQL debe ser ejecutada individualmente por el servidor de la base de datos.

Esto significa que su aplicación cliente debe enviar cada consulta al servidor de la base de datos, esperar a que sea procesada, recibir los resultados, hacer algunos cálculos, y enviar después otras consultas al servidor. Todo esto implica la comunicación entre procesos y puede también implicar una sobrecarga a la red si su cliente se encuentra en una máquina diferente a la del servidor de la base de datos.

Con `PL/pgSQL` usted puede agrupar un bloque de cálculos y una serie de consultas *dentro* del servidor de la base de datos, obteniendo de esta manera el poder de un lenguaje procedimental y la facilidad de uso de SQL, pero ahorrando una gran cantidad de tiempo debido a que no tiene la sobrecarga de la comunicación completa cliente/servidor. Esto puede aumentar el desempeño de una manera considerable.

También, con `PL/pgSQL` usted puede usar todos los tipos de datos, operadores y funciones de SQL.

Argumentos y Tipos de Datos de los Resultados Soportados

Las funciones escritas en `PL/pgSQL` pueden aceptar como argumentos cualquier tipo de datos escalar o matriz soportados por el servidor, y pueden regresar un resultado de cualquiera de esos tipos. Tambien pueden aceptar o regresar cualquier tipo compuesto (tipo renglón, row type) especificado por su nombre. También es posible declarar una función de `PL/pgSQL` que regrese un registro (record), lo cual significa que el resultado es del tipo renglón, cuyas columnas son definidas por la especificación de la llamada a la consulta.

Las funciones `PL/pgSQL` también pueden declararse para que acepten y regresen los tipos polimórficos anyelement y anyarray. Los tipo de datos actuales manejados por

una función polimórfica pueden variar de llamada en llamada. En la sección de nombre *Aliases para los Parámetros de las Funciones* se muestra un ejemplo.

Las funciones PL/pgSQL también pueden declararse para que regresen un "conjunto (set)", o tabla, de cualquier tipo de datos del cual puedan regresar una sola instancia. Tal función genera su salida ejecutando RETURN NEXT para cada elemento deseado del conjunto de resultados.

Finalmente, una función PL/pgSQL puede ser declarada para que regrese void si acaso no se utiliza el valor de retorno.

PL/pgSQL no tiene actualmente el soporte completo para tipos dominio: trata un dominio de la misma manera que el tipo escalar subyacente. Esto significa que las restricciones asociadas con el dominio no serán forzadas. Esto no es un problema para los argumentos de las funciones, pero es un peligro si usted declara una función PL/pgSQL que devuelva un tipo dominio.

Sugerencias para Desarrollar en PL/pgSQL

Una buena manera de programar en PL/pgSQL es utilizar su editor de textos favorito para crear sus funciones y, en otra ventana, usar psql para cargar y probar esas funciones. Si lo hace de esta manera, es una buena idea escribir la función usando **CREATE OR REPLACE FUNCTION**. De esta manera usted puede recargar el archivo para actualizar la definición de la función. Por ejemplo:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
        ....
$$ LANGUAGE plpgsql;
```

Desde psql usted puede cargar o recargar la definición de la función con

```
\i nombrearchivo.sql
```

e inmediatamente teclear los comandos SQL necesarios para probar la función.

Otra manera adecuada de programar en PL/pgSQL es por medio de una herramienta GUI de acceso a la base de datos, que le facilite el desarrollo de un lenguaje procedimental. Un ejemplo es PgAccess, aunque existen otras. Estas herramientas suelen ofrecer prestaciones convenientes para escapar las comillas sencillas y facilitar la recreación y depuración de las funciones.

Manejo de las Comillas

El código de una función PL/pgSQL se especifica en **CREATE FUNCTION** como una cadena literal. Si usted escribe la cadena literal de la manera común, con comillas sencillas como delimitadores, entonces cada comilla sencilla dentro del cuerpo de la función debe ser duplicada; de igual manera cualquier diagonal inversa debe ser duplicada. La duplicación de las comillas es, por lo menos, tedioso, y en los casos complejos el código se vuelve incomprensible, debido a que se pueden necesitar media docena o más de comillas sencillas contiguas. Es recomendable que mejor escriba el cuerpo de la función como una cadena literal "dollar-quoted". Con el estilo dollar-quoting usted nunca necesitará duplicar las comillas, en cambio deberá cuidar el uso de un delimitador dollar-quoting diferente para cada nivel de anidamiento que necesite. Por ejemplo, usted debe escribir el comando **CREATE FUNCTION** como

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
        ....
$PROC$ LANGUAGE plpgsql;
```

Dentro de éste, usted puede usar comillas para las cadenas literales simples en los comandos SQL y $$ para delimitar fragmentos de comandos SQL que esté ensamblando para crear cadenas más largas. Si necesita incluir en su texto $$, puede usar Q y así consecutivamente.

La siguiente tabla le muestra lo que debe hacer cuando escriba comillas sin delimitar su cadena con $$. Puede serle útil cuando traduzca sus funciones hechas con código en el que no se usaba $$ y para hacer más claro el código.

1 comilla

Para iniciar y finalizar el cuerpo de la función, por ejemplo:

```
CREATE FUNCTION foo() RETURNS integer AS '
        ....
' LANGUAGE plpgsql;
```

En cualquier lugar dentro del cuerpo de la función delimitada por una comilla, las comillas *deben* aparecer en pares.

2 comillas

Para cadenas literales dentro del cuerpo de la función, por ejemplo:

```
a_output := ”Blah”;
SELECT * FROM users WHERE f_name=”foobar”;
```

Usando la técnica de $$, usted debería escribir

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

que es exactamente lo que el analizador de PL/pgSQL vería en ambos casos.

4 comillas

Cuando requiera una comilla sencilla en una cadena constante dentro del cuerpo de la función, por ejemplo;

```
a_output := a_output || ” AND name LIKE ””foobar”” AND xyz”
```

El valor realmente concatenado a a_output sería: `AND name LIKE 'foobar' AND xyz`.

Usando la técnica de $$, usted escribiría

```
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

teniendo cuidado de que cualquiera de los delimitadores estilo dólar alrededor de esta cadena no sea ciertamente $$.

6 comillas

Cuando una comillas sencilla en una cadena dentro del cuerpo de la función sea adyacente al final de la cadena constante, por ejemplo:

```
a_output := a_output || ” AND name LIKE ””foobar”””
```

El valor concatenado a a_output sería entonces: `AND name LIKE 'foobar'`.

Con el estilo dólar se transformaría en:

```
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10 comillas

Cuando requiera dos comillas sencillas en una cadena constante (que equivalen a 8 comillas) y ésta es adyacente al final de la cadena constante (2 más). Es probable que solamente necesite esto si está escribiendo una función que genera otras funciones. Por ejemplo:

```
a_output := a_output || ” if v_” ||
    referrer_keys.kind || ” like ”””””
    || referrer_keys.key_string || ”””””
    then return ”””  || referrer_keys.referrer_type
    || ”””; end if;”;
```

El valor de `a_output` sería entonces:

```
if v_... like ”...” then return ”...”; end if;
```

Con el estilo dólar quedaría como

```
  a_output := a_output || $$ if v_$$ || referrer_keys.kind ||
  $$ like ’$$ || referrer_keys.key_string || $$’
    then return ’$$  || referrer_keys.referrer_type
    || $$’; end if;$$;
```

en donde suponemos que solamente requerimos colocar una comilla dentro de `a_output` debido a que será re-encomillado antes de usarse.

Una variante es el escapar las comillas en el cuerpo de la función con una diagonal invertida en lugar de duplicarlas. Con este método escribiría cosas como `\'\'` en lugar de `””`. Algunas personas consideran esta técnica más sencilla, pero otras no.

La Estructura de PL/pgSQL

`PL/pgSQL` es un lenguaje estructurado a base de bloques. El texto completo de la definición de una función debe ser un *bloque*. Un bloque está definido como:

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

Cada declaración y cada sentencia dentro de un bloque deben terminar con punto y coma. Un bloque que aparece dentro de otro bloque debe contar con un punto y coma después de `END`, como se muestra abajo; sin embargo, el `END` final que cierra el cuerpo de una función no requiere el punto y coma.

Todas la palabras reservadas y los identificadores pueden escribirse en mayúsculas, minúsculas o una mezcla de ellas. Los identificadores son convertidos implicitamente a minúsculas, a menos que estén encerradas en comillas dobles.

Existen dos tipos de comentarios en `PL/pgSQL`. Un doble guión (`--`) marca el inicio de un comentario que se extiende hasta el final de la línea. Los símbolos `/*` marcan el inicio de un bloque de un comentario que se extiende hasta la aparición de `/*`. Los bloques de comentarios no pueden anidarse, pero los comentarios con doble guión pueden ocultar los delimitadores de un bloque de un comentario `/*` y `*/`.

Cualquier sentencia en la sección de sentencias de un bloque puede ser un *subbloque*. Los subbloques pueden usarse para agrupamientos lógicos o para hacer locales las variables de un grupo de sentencias.

Las variables declaradas en la sección de declaraciones que precede a un bloque son inicializadas a su valor por omisión cada vez que se entra al bloque, no solamente una vez por llamada a la función. Por ejemplo:

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
DECLARE
    quantity integer := 30;
BEGIN
  RAISE NOTICE 'Aquí, cantidad tiene el valor de %', quantity;
  -- Aquí, cantidad tiene el valor de 30
    quantity := 50;
    --
    -- Creamos un subbloque
    --
    DECLARE
        quantity integer := 80;
    BEGIN
  RAISE NOTICE 'Aquí, cantidad tiene el valor de %', quantity;
    -- Aquí, cantidad tiene el valor de 80
    END;

  RAISE NOTICE 'Aquí, cantidad tiene el valor de %', quantity;
  -- Aquí, cantidad tiene el valor de 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

Es importante no confundir el uso de **BEGIN**/**END** para agrupar sentencias en PL/pgSQL con los comandos de la base de datos para el control de las transacciones. Los comandos **BEGIN**/**END** de PL/pgSQL se usan solamente para agrupar; no inician ni terminan una transacción. Las funciones y los procedimientos disparadores (trigger) siempre son ejecutados dentro de una transacción establecida por una consulta externa — no pueden iniciar o hacer que se ejecute un commit de esa transacción, puesto que no existe un contexto para su ejecución. Sin embargo, un bloque que contenga una claúsula de EXCEPTION sí genera una subtransacción que puede echarse atrás (rolled back) sin afectar la transacción externa. Para más detalles puede ver la sección de nombre *Atrapar los Errores*.

Declaraciones

Todas las variables usadas en un bloque deben ser declaradas en la sección de declaraciones de ese bloque. (La única excepción es que la variable índice de un ciclo FOR que itera sobre un rango de valores enteros es declarada automáticamente como una variable entera).

Las variables de PL/pgSQL pueden tener cualquier tipo de dato de SQL, tales como integer, varchar y char.

Estos son algunos ejemplos de la declaración de variables:

```
user_id  integer;
cantidad  numeric(5);
url  varchar;
myrenglon  nombretabla%ROWTYPE;
mycampo    nombretabla.nombrecolumna%TYPE;
unrenglon  RECORD;
```

La sintaxis general de la declaración de una variable es:

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

La cláusula `DEFAULT`, cuando existe, especifica el valor inicial asignado a la variable cuando se ingresa al bloque. Si la claúsula `DEFAULT` no existe entonces la variable se inicializa al valor nulo de SQL. Si se especifica `NOT NULL`, la asignación de un valor nulo dará por resultado un error en tiempo de ejecución. Todas las variable declaradas como `NOT NULL` deben tener especificado un valor no nulo.

El valor por omisión se evalúa cada vez que se entra al bloque. Así, por ejemplo, el asignar `now()` a la variable de tipo timestamp causará que esa variable tenga la hora de la llamada a la función actual, no la hora en que la función fue precompilada.

Ejemplos:

```
cantidad  integer DEFAULT 32;
url varchar := 'http://misitio.com';
user_id CONSTANT integer := 10;
```

Aliases para los Parámetros de las Funciones

Los parámetros pasados a las funciones se nombran con los identificadores `$1`, `$2`, etc. Opcionalmente, se pueden declarar aliases para `$n` nombres de parámetros para una mayor claridad. Tanto el alias como su identificador numérico pueden ser usados para referirse al valor del parámetro.

Existen dos maneras de crear un alias. La manera preferida es asignarle un nombre al parámetro en el comando **CREATE FUNCTION**, por ejemplo:

```
CREATE FUNCTION impuesto_ventas(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

La otra manera, que era la única disponible para versiones previas a PostgreSQL 8.0, es el declarar explícitamente un alias, usando la sintaxis de declaración

```
nombre ALIAS FOR $n;
```

El mismo ejemplo en este estilo se vería así

```
CREATE FUNCTION impuesto_ventas(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Algunos ejemplos más:

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- Algún procesamiento aquí
END;
$$ LANGUAGE plpgsql;


CREATE FUNCTION concatenar_campos_seleccionados(in_t nombretabla)
```

```
                        RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

Cuando el tipo regresado de una función PL/pgSQL se declara como de tipo polimórfico (anyelement o anyarray), se crea un parámetro especial $0. Su tipo de dato es el tipo de dato actual regresado por la función, el cual deduce de los tipos de entrada actuales. Esto permite a la función acceder a su tipo de retorno actual como se muestra en la sección de nombre *Copiando Tipos*. La variable $0 se inicializa a nulo y puede ser modificada por la función, de tal manera que pueda ser usada para almacenar el valor de retorno si eso es lo que se desea, aunque esto no es necesario. A la variable $0 también se le puede dar un alias. Por ejemplo, esta función trabaja con cualquier tipo de dato que tenga un operador +:

```
CREATE FUNCTION suma_tres_valores(v1 anyelement,v2 anyelement,
      v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    resultado ALIAS FOR $0;
BEGIN
    resultado := v1 + v2 + v3;
    RETURN resultado;
END;
$$ LANGUAGE plpgsql;
```

Copiando Tipos

variable%TYPE

%TYPE proporciona el tipo de dato de una variable o de una columna de una tabla. Puede utilizarla para declarar variables que almacenarán valores de la base de datos. Por ejemplo, supongamos que tiene una columna llamada id_usuario en su tabla usuarios. Para declarar una variable con el mismo tipo de dato que usuarios.id_usuario usted escribiría:

```
id_usuario usuarios.id_usuario%TYPE;
```

Al usar %TYPE no necesita conocer el tipo de dato de la estructura a la que está haciendo referencia, y lo más importante, si el tipo de dato del ítem referido cambia en algún momento en el futuro (por ejemplo: si cambia el tipo de id_usuario de integer a real), usted no necesita cambiar la definición en su función.

%TYPE es particularmente útil en las funciones polimórficas, puesto que los tipos de datos necesarios para las variables internas puede cambiar de una llamada a otra. Se pueden crear variables apropiadas aplicando %TYPE a los argumentos o a los comodines de los resultados de la función.

Tipos Renglón

nombre nombre_tabla%ROWTYPE;
nombre nombre_tipo_compuesto;

Una variable de un tipo compuesto se denomina una variable *renglón* (o *tipo-renglon*). Dicha variable puede almacenar un renglón completo resultado de una consulta **SELECT** o **FOR**, en tanto el conjunto de columnas de la consulta coincida con el tipo de la variable declarado. Los campos individuales del valor del renglón pueden accederse usando la notación usual de punto, por ejemplo `renglonvar.campo`.

Una variable renglón puede declararse para tener el mismo tipo que los renglones de una tabla o vista existente, usando la notación *`nombre_tabla`*`%ROWTYPE`; o puede ser declarada dando el nombre de un tipo compuesto. (Puesto que todas las tablas tiene un tipo compuesto asociado del mismo nombre, en PostgreSQL realmente no importa si escribe `%ROWTYPE` o no. Pero la forma con `%ROWTYPE` es más portable).

Los parámetros de una función pueden ser de tipo compuesto (renglones de tablas completos). En ese caso, el identificador correspondiente `$`*n* será un variable renglón, y pueden seleccionarse los campos a partir de él, por ejemplo `$1.id_usuario`.

En una variable tipo-renglón, solamente son accesibles las columnas definidas por el usuario, no el OID u otras columnas del sistema (debido a que el renglón podría ser de una vista). Los campos del tipo renglón heredan el tamaño o la precisión del campo de la tabla para datos tales como char(*n*).

Este es un ejemplo del uso de los tipos compuestos:

```
CREATE FUNCTION merge_fields(t_row tablename) RETURNS text AS $$
DECLARE
    t2_row table2name%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2name WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM tablename t WHERE ... ;
```

Tipos Registro

```
nombre RECORD;
```

Las variables registro son similares a las variables tipo-renglón, pero no tienen una estructura predefinida. Toman la estructura actual del renglón al que son asignados durante un comando **SELECT** o **FOR**. La subestructura de una variable registro puede cambiar cada vez que es asignada. Una consecuencia de ésto es que mientras una variable registro no sea asignada por primera vez, no tendrá subestructura y cualquier intento de acceder a uno de sus campos generará un error en tiempo de ejecución.

Observe que `RECORD` no es realmente un tipo de dato, sino un comodín. También debe darse cuenta que cuando una función `PL/pgSQL` se declara para regresar un tipo record, no es lo mismo que una variable registro, aunque tal variable pueda utilizar una variable registro para almacenar su resultado. En ambos casos la estructura actual del renglón es desconocida cuando se escribe la función, pero para una función que regresa un record la estructura actual se determina cuando se analiza la consulta solicitada, mientras que una variable record puede cambiar su estructura de renglón al vuelo.

RENAME

```
RENAME nombreviejo TO nombrenuevo;
```

La declaración RENAME le permite cambiar el nombre de una variable, registro o renglón. Inicialmente es útil si NEW u OLD deben ser referidas con otro nombre dentro de un procedimiento disparador. Vea también ALIAS.

Ejemplos:

```
RENAME id TO id_usuario;
RENAME this_var TO esta_variable;
```

Nota: RENAME parece no funcionar desde PostgreSQL 7.3. La reparación es de baja prioridad puesto que ALIAS cubre la mayor parte de los usos prácticos de RENAME.

Expresiones

Todas las expresiones usadas en sentencias de PL/pgSQL son procesadas usando el ejecutor SQL regular del servidor. De hecho, una consulta como

```
SELECT expresión
```

se ejecuta usando al administrador SPI. Antes de la evaluación, las ocurrencias de los identificadores de variables de PL/pgSQL son reemplazados por los parámetros y el valor actual de las variables se pasa al ejecutor en el arreglo de parámetros. Esto permite que el plan de la consulta del **SELECT** sea preparado solamente una vez y después reutilizado para las evaluaciones subsecuentes.

La evaluación hecha por el analizador (parser) pricipal de PostgreSQL tiene algunos efectos secundarios en la interpretación de los valores constantes. En detalle existen diferencias entre lo que hacen estas dos funciones:

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS timestamp AS $$
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
        RETURN 'now';
    END;
$$ LANGUAGE plpgsql;
```

y

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

En en caso de logfunc1, el analizador principal de PostgreSQL sabe cuando preparar el plan para el **INSERT**, que la cadena 'now' debe interpretarse como timestamp debido a que su columna objetivo de logtable es de ese tipo. Asi, creará una constante con ella en este momento y su valor constante será utilizado en todas las invocaciones de logfunc1 durante la vida de la sesión. No es necesario decir que esto no es lo que el programador deseaba hacer.

En el caso de `logfunc2`, el analizador principal de PostgreSQL no sabe que tipo debe tener `'now'` y entonces regresa un valor de tipo text que contiene la cadena `now`. Durante la asignación subsiguiente a la variable local `curtime`, el intérprete de `PL/pgSQL` transforma esta cadena al tipo timestamp llamando a las funciones `text_out` y `timestamp_in` para la conversión. Así, la hora y la fecha calculadas en cada ejecución es la que espera el programador.

La naturaleza mutable de las variables registro presentan un problema en este ámbito. Cuando los campos de una variable registro son usados en expresiones o sentencias, los tipos de datos de los campos no deben cambiar entre llamadas de una misma expresión, puesto que la expresión será planeada usando los tipos de dato que estén presentes cuando la expresión sea alcanzada por primera vez. Tenga en cuenta esto al escribir procedimientos disparadores que manejen eventos para más de una tabla. (Cuando se necesario, puede utilizar **EXECUTE** para evitar este problema).

Sentencias Básicas

En esta sección y las siguientes describimos todos los tipos de sentencias que entiende explicitamente `PL/pgSQL`. Cualquier cosa que no sea reconocida como una de estos tipos de sentencias se presume que es un comando SQL y se envía a la máquina de la base de datos principal para ejecutarse (después de la sustitución de cualquier variable de `PL/pgSQL` usada en la sentencia). Así, por ejemplo, los comandos SQL **INSERT**, **UPDATE** y **DELETE** pueden considerarse como sentencias de `PL/pgSQL`, pero no se listan aquí específicamente.

Asignación

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

```
identificador := expresión;
```

Tal como se escribió arriba, la expresión en tal sentencia es evaluada por medio de un comando SQL **SELECT** enviado a la máquina de la base de datos principal. La expresión debe producir un valor único.

Si el tipo de dato del resultado de la expresión no coincide con el tipo de dato de la variable, o la variable tiene un tamaño/precisión específico (como char(20)), el valor resultante será convertido implícitamente por el intérprete de `PL/pgSQL` usando el tipo de resultado de la función de salida (output_function) y el tipo de variable de la función de entrada (input_function). Observe que esto puede resultar potencialmente en un error en tiempo de ejecución generado por la función de entrada, si la forma de la cadena del valor resultante no es aceptable para la función de entrada.

Ejemplos:

```
id_usuario := 20;
impuesto := subtotal * 0.15;
```

SELECT INTO

El resultado de un comando **SELECT** que produce columnas múltiples (pero sólo un renglón) puede ser asignado a una variable registro, variable tipo-renglón o una lista de variables escalares. Esto se realiza con:

```
SELECT INTO meta expresiones_select FROM ...;
```

en donde *meta* puede ser una variable registro, una variable renglón o una lista de variables simples y campos registro/renglón separados por comas. Las *expresiones_select* y el resto del comando son iguales que en el SQL regular.

Observe que es muy diferente de la interpretación normal de **SELECT INTO** de PostgreSQL, en donde la meta INTO es una tabla recién creada. Si desea crear una tabla a partir de una resultado de un **SELECT** dentro de una función PL/pgSQL, use la sintaxis **CREATE TABLE ... AS SELECT**.

Si un renglón o una lista de variables se usa como meta, los valores seleccionados deben coincidir exactamente con la estructura de la meta, u ocurrirá un error en tiempo de ejecución. Cuando la meta es una variable registro, automáticamente se configura siguiendo la estructura del tipo renglón de la columnas del resultado de la consulta.

Las sentencia **SELECT** es la misma que el comando **SELECT** normal y puede utilizarse con todo su poder, con la excepción de la claúsula INTO.

La claúsula INTO puede aparecer casi en cualquier lugar en la sentencia **SELECT**. Suele escribirse ya sea antes de SELECT como se muestra arriba o justo antes de FROM — es decir, justo antes o justo después de la lista de las *expresiones_select*.

Si la consulta regresa cero renglones, se asignan valores nulos a la(s) meta(s). Si la consulta regresa múltiples renglones, se asigna el primer renglón a la(s) meta(s) y el resto se descarta. (Observe que "el primer renglón" no está bien definido a menos que haga uso de ORDER BY).

Puede verificar la variable especial FOUND (vea la sección de nombre *Obteniendo el Estado del Resultado*) después de una sentencia **SELECT INTO** para determinar si la asignación fue correcta, es decir, que la consulta regresó al menos un renglón. Por ejemplo:

```
SELECT INTO mireg * FROM emp WHERE empnombre= minombre;
IF NOT FOUND THEN
    RAISE EXCEPTION 'no se encontró al empleado %', minombre;
END IF;
```

Para probar si un resultado registro/renglón es nulo puede usar la condicional IS NULL. Sin embargo, no existe una manera de decir si han sido descartados algunos renglones adicionales. Aquí hay un ejemplo que maneja el caso en que no se han regresado renglones:

```
DECLARE
    reg_usuarios  RECORD;
BEGIN
    SELECT INTO reg_usuarios * FROM usuarios WHERE id_usuario=3;

    IF reg_usuarios.homepage IS NULL THEN
        -- El usuario no digitó un homepage, regresar "http://"
        RETURN 'http://';
    END IF;
END;
```

Ejecución de una Expresión o Consulta Sin Resultados

En algunas ocasiones se desea evaluar un expresión o una consulta y descartar sus resultados (típicamente debido a que se está llamando a una función que tiene efectos

secundarios útiles, pero valores de resultados inútiles). Para hacer esto en PL/pgSQL utilice la sentencia **PERFORM**:

```
PERFORM consulta;
```

Esto ejecuta la *consulta* y descarta el resultado. Escriba la *consulta* de la misma manera en que lo haría en un comando **SELECT**, pero reemplace la palabra reservada inicial **SELECT** con **PERFORM**. Las variables PL/pgSQL serán sustituídas dentro de la consulta de la manera usual. También a la variable especial FOUND se le asignará el valor de verdadero si la consulta produce al menos un renglón o falso si no produce renglones.

> **Nota:** Uno esperaría que un **SELECT** sin una clúasula INTO haría lo mismo, pero hasta el momento la única manera aceptada de hacerlo es con **PERFORM**.

Un ejemplo:

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

No Hacer Absolutamente Nada

Algunas veces una sentencia comodín que no hace nada es útil. Por ejemplo, puede indicar que una rama del una cadena if/then/else se encuentra vacía deliberadamente. Para hacer esto, use la sentencia **NULL**:

```
NULL;
```

Por ejemplo, los siguientes dos fragmentos de código son equivalentes:

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        NULL;  -- ignore el error
END;

BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN  -- ignore el error
END;
```

Dejamos a su elección el que quiera usar.

> **Nota:** En el PL/SQL de Oracle las sentencias vacias no están permitidas, así que las sentencias **NULL** son *requeridas* para situaciones como ésas. PL/pgSQL permite, en cambio, el escribir nada.

Ejecución de Comandos Dinámicos

Con frecuencia tendrá necesidad de generar comandos dinámicos dentro de sus funciones PL/pgSQL, esto es, comandos que involucren diferentes tablas o diferentes tipos, cada vez que son ejecutados. Los intentos normales de PL/pgSQL para guardar los planes para los comandos no trabajarán es tales escenarios. Para manejar esa clase de problemas se proporcion el comando **EXECUTE**:

```
EXECUTE cadena-de-comando;
```

En donde *cadena-de-comando* es una expresión que da como resultado una cadena (de tipo text), la cual contiene el comando a ser ejecutado. Esta cadena se alimenta, literalmente, a la máquina SQL.

Observe en particular que no se realiza ninguna sustitución de variables PL/pgSQL en la cadena de comando. Los valores de las variables deben ser insertados en la cadena de comando al momento de construirse.

A diferencia de otros comandos en PL/pgSQL, un comando ejecutado por una sentencia **EXECUTE** no se prepara y guarda sólo una vez durante la vida de la sesión. En cambio, el comando se prepara cada vez que se ejecuta la sentencia. La cadena de comando puede se creada dinámicamente dentro de la función para ejecutar acciones en diferentes tablas y columnas.

Los resultados de un **SELECT** son descartados por el comando **EXECUTE** y el **SELECT INTO** no está soportado actualmente dentro de **EXECUTE**. Así que no hay manera de extraer un resultado de un **SELECT** creado dinámicamente usando el comando **EXECUTE** simple. Sin embargo, existen otras maneras de hacerlo: una implica el usar la forma del ciclo **FOR-IN-EXECUTE** descrita en la sección de nombre *Ciclos a Través de Resultados de Consultas*, y la otra es utilizar un cursor con **OPEN-FOR-EXECUTE**, como se describe en la sección de nombre *Apertura de Cursores*.

Cuando trabaje con comandos dinámicos tendrá que contender frecuentemente con el escape de las comillas sencillas. El método recomendado para entrecomillar texto fijo en el cuerpo de su función es el entrecomillar con el signo de dólares. (Si tiene código heredado que no use el entrecomillado con el signo de dólares, refiérase a la visión general en la sección de nombre *Manejo de las Comillas*, la cual de ahorrará algunos esfuerzos al traducir dicho código a un esquema más razonable).

Los valores dinámicos que se van a insertar dentro de la consulta construída requieren un manejo especial, puesto que ellos mismos pueden contener comillas. Un ejemplo (presupone que usted está usando entrecomillado con el signo de dólares para toda la función, de tal manera que las comillas no necesitan ser dobles):

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(nombrecol)
        || ' = '
        || quote_literal(nuevovalor)
        || ' WHERE llave= '
        || quote_literal(llavevalor);
```

Este ejemplo ilustra el uso de las funciones quote_ident(text) y quote_literal(text). Por seguridad, las variables que contienen valores que deben ser cadenas literales en el comando construído deben pasarse a quote_literal. Ambas siguen los pasos apropiados para regresar el texto de entrada encerrado en comillas sencillas o dobles respectivamente, con cualquier caracter especial embebido escapado de la manera correcta.

Observe que el entrecomillado con el símbolo de dólares es útil solamente para entrecomillar texto fijo. Sería muy mala idea el tratar de hacer el ejemplo anterior como

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(nombrecol)
```

```
        || ' = $$'
        || nuevovalor
        || '$$ WHERE llave= '
        || quote_literal(llavevalor);
```

debido a que podría fallar si el contenido de `nuevovalor` contiene `$$`. La misma objeción podría aplicarse a cualquier otro delimitador del tipo dólar que usara. Asi que, para entrecomillar de manera segura el texto que no conozca de inicio, *debe* usar `quote_literal`.

Obteniendo el Estado del Resultado

Existen varias maneras para determinar el efecto de un comando. El primer método consiste en usar el comando **GET DIAGNOSTICS**, que tiene la forma:

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

Este comando permite recuperar los indicadores del estado del sistema. Cada *`item`* es una palabra reservada que identifica a un valor del estado que se asignará a una variable específica (la cual debe ser del tipo adecuado para recibirlo). Los ítems de los estados disponibles actualmente son `ROW_COUNT`, la cantidad de renglones procesados por el último comando enviado al motor SQL, y `RESULT_OID`, el OID del último renglón insertado por el comando SQL. Observe que `RESULT_OID` solamente es útil después de un comando **INSERT**.

Un ejemplo:

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

El segundo método para determinar los efectos de un comando es revisar la variable espacial llamada `FOUND`, la cual es de tipo boolean. `FOUND` es falso de entrada dentro de cualquier llamada a una función `PL/pgSQL`. Su valor se establece por cada uno de los siguientes tipos de sentencias:

- Una sentencia **SELECT INTO** establece `FOUND` como verdadero si regresa un renglón, falso si ningún renglón es regresado.
- Un comando **PERFORM** define a `FOUND` como verdadero si produce (y descarta) un renglón, falso si ningún renglón es producido.
- Las sentencias **UPDATE**, **INSERT** y **DELETE** definen a `FOUND` como verdadero si al menos se afecta un renglón, como falso si ningún renglón es afectado.
- Una sentencia **FETCH** define a `FOUND` como verdadero si regresa un renglón, como falso si ningún renglón en regresado.
- Una sentencia **FOR** deja a `FOUND` con el valor de verdadero si itera una o más veces, en caso contrario lo pone en falso. Esto aplica a las tres variantes de la sentencias **FOR** (ciclos **FOR** enteros, ciclos **FOR** de conjuntos de registros y ciclos **FOR** de conjuntos de registros dinámicos). `FOUND` se establece de esta manera cuando se sale del ciclo **FOR**; dentro del la ejecución del ciclo, `FOUND` no se modifica por la sentencia **FOR**, aunque puede ser cambiado por la ejecución de otra sentencia dentro del cuerpo del ciclo.

`FOUND` es una variable local dentro de cada función `PL/pgSQL`; así que cualquier cambio en ella, afecta solo a la función actual.

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 los datos de PostgreSQL de una manera flexible y poderosa.

Regreso de una Función

Existen dos comandos que le permiten regresar datos desde una función: **RETURN** y **RETURN NEXT**.

RETURN

```
RETURN expresión;
```

RETURN con una expresión termina la función y regresa el valor de *expresión* a quién la está llamando. Esta forma debe ser usada para las funciones PL/pgSQL que no regresan un conjunto.

Cuando se regresa un tipo escalar puede usarse cualquier expresión. El resultado de la expresión será transformado (cast) automáticamente al tipo de retorno de la función, tal como se definió en la asignación. Para regresar un valor compuesto (renglón) debe escribir una variable registro o renglón como la *expresión*.

El valor de retorno de una función no puede quedarse indefinido. Si el control alcanza el final de bloque más externo de la función sin encontrar una sentencia **RETURN**, se producirá un error en tiempo de ejecución.

Si declaró que la función regrese void, también debe proporcionar una sentencia **RETURN**, pero en este caso la expresión que sigue a **RETURN** será opcional y será ignorada si está presente.

RETURN NEXT

```
RETURN NEXT expresión;
```

Cuando se declara que una función PL/pgSQL regrese SETOF *alguntipo*, el procedimiento a seguir es ligeramente diferente. En este caso, los items individuales a retornar se especifican en comandos **RETURN NEXT**, y después un comando final **RETURN** sin argumentos se utiliza para indicar que la función ha terminado de ejecutarse. **RETURN NEXT** puede usarse con tipos escalares y compuestos; en el último caso, sera regresado una "tabla" completa de resultados.

Las funciones que usan **RETURN NEXT** deben se llamadas de la siguiente manera:

```
SELECT * FROM alguna_func();
```

Es decir, las funciones deben se usadas como una tabla fuente en una claúsula FROM.

RETURN NEXT realmente no regresa de la función, simplemente almacena el valor de la expresión. Después, continúa la ejecución de la siguiente sentencia en la función PL/pgSQL. Al ejecutarse los comandos sucesivos **RETURN NEXT**, se va armando el resultado. Un **RETURN** final, sin argumentos, ocasiona que el control salga de la función.

> **Nota:** La implementación actual de **RETURN NEXT** para PL/pgSQL almacena el conjunto resultante completo antes de regresar de la función, como es explica arriba. Esto significa que si una función PL/pgSQL produce un conjunto resultante muy grande, el desempeño puede empobrecerse: los datos serán escritos en el disco para evitar el consumo de memoria, pero la función misma no regresará hasta que todo el conjunto resultante haya sido generado. Una versión futura de PL/pgSQL puede permitir a los usuarios el definir

funciones que regresen conjuntos que no tengan esa limitante. Por lo pronto, el momento en que los datos empiezan a escribirse en el disco es controlado por la variable de configuración `work_mem`. Los administradores que cuenten con suficiente memoria para almacenar conjuntos resultantes más grandes en la memoria, deben considerar el aumentar este parámetro.

Condicionales

Las sentencias `IF` le permiten ejecutar comandos cuando se dan ciertas condiciones. `PL/pgSQL` tiene cinco formas de `IF`:

- `IF ... THEN`
- `IF ... THEN ... ELSE`
- `IF ... THEN ... ELSE IF`
- `IF ... THEN ... ELSIF ... THEN ... ELSE`
- `IF ... THEN ... ELSEIF ... THEN ... ELSE`

IF-THEN

```
IF expresión-lógica THEN
    sentencias
END IF;
```

Las sentencias `IF-THEN` son las formas más simples de `IF`. Las sentencias entre `THEN` y `END IF` serán ejecutadas si la condición es verdadera. De otra manera, serán ignoradas.

Ejemplo:

```
IF v_id_usuario <> 0 THEN
    UPDATE usuarios SET email = v_email WHERE id_usuario= v_id_usuario;
END IF;
```

IF-THEN-ELSE

```
IF expresión-lógica THEN
    sentencias
ELSE
    sentencias
END IF;
```

Las sentencias `IF-THEN-ELSE` añaden funcionalidad a `IF-THEN` permitiéndole especificar un conjunto de sentencias alternativo que debe ejecutarse si la condición produce un valor de falso.

Ejemplo:

```
IF parentid IS NULL OR parentid = ”
THEN
    RETURN fullname;
ELSE
```

```
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;

IF v_cuenta> 0 THEN
    INSERT INTO usuarios_cuenta(count) VALUES (v_cuenta);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

IF-THEN-ELSE IF

Las sentencias IF pueden anidarse, como se muestra en el siguiente ejemplo:

```
IF demo_renglon.sexo = 'm' THEN
    pretty_sex := 'hombre';
ELSE
    IF demo_renglon.sexo = 'f' THEN
        pretty_sex := 'mujer';
    END IF;
END IF;
```

Cuando se usa esta forma, realmente se está anidano la sentencia IF dentro de la parte ELSE de la sentencia IF. Así, require una sentencia END IF para cada IF anidado y una para el padre IF-ELSE. Esto funciona, pero se vuelve tedioso cuando existen varias alternativas por revisar. De ahi que exista la siguiente forma.

IF-THEN-ELSIF-ELSE

```
IF expresión-lógica THEN
    sentencias
[ ELSIF expresión-lógica THEN
    sentencias
[ ELSIF expresión-lógica THEN
    sentencias
    ...]]
[ ELSE
    sentencias ]
END IF;
```

IF-THEN-ELSIF-ELSE proporciona un método más conveniente para revisar varias alternativas en una sentencia. Formalmente es equivalente a los comandos anidados IF-THEN-ELSE-IF-THEN, pero solo se necesita un END IF.

A continuación en ejemplo:

```
IF numero = 0 THEN
    resultado := 'cero';
ELSIF numero > 0 THEN
    resultado := 'positivo';
ELSIF numero < 0 THEN
    resultado := 'negativo';
ELSE
    -- hmm, la única otra posibilidad que el número sea nulo
    resultad := 'NULL';
END IF;
```

IF-THEN-ELSEIF-ELSE

ELSEIF es un alias de ELSIF.

Ciclos Simples

Con las sentencia LOOP, EXIT, WHILE y FOR, usted puede hacer que en sus funciones PL/pgSQL se repitan una serie de comandos.

LOOP

```
[<<etiqueta>>]
LOOP
    sentencias
END LOOP;
```

LOOP define un ciclo incondicional que se repite indefinidamente hasta que encuentra alguna sentencia EXIT o **RETURN**. La etiqueta opcional puede usarse por las sentencias EXIT en los ciclos anidados para especificar que nivel de anidamiento debe terminarse.

EXIT

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

Si no se proporciona una *etiqueta* se termina el ciclo más interno y se ejecuta a continuación la sentencia posterior a END LOOP. Si se define una *etiqueta*, ésta debe ser la etiqueta del nivel actual o de algún otro más externo del ciclo anidado o del bloque. Entonces, el ciclo o bloque nombrado se termina y el control continúa con la sentencia posterior al END del ciclo/bloque correspondiente.

Si WHEN está presente, la salida del ciclo ocurre solo si la condición especificada es verdadera, de otra manera, el control pasa a la sentencia después del EXIT.

EXIT puede utilizarse para provocar una salida temprana de todo tipo de ciclos; no está limitado al uso de ciclos condicionales.

Ejemplos:

```
LOOP
    -- algun procesamiento
    IF count > 0 THEN
        EXIT;  -- salir del ciclo
    END IF;
END LOOP;

LOOP
    -- algún procesamiento
    EXIT WHEN count > 0;  -- mismo resultado que en el ejemplo anterior
END LOOP;

BEGIN
    -- algún procesamiento
    IF stocks > 100000 THEN
        EXIT;  -- causa la salida del bloque BEGIN
    END IF;
END;
```

WHILE

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

La sentencia WHILE repite una secuencia de sentencias tanto como la expresión de la condición produzca un valor de verdadero. La condición se revisa justo antes de cada entrada al cuerpo del ciclo.

Por ejemplo:

```
WHILE cantidad_adeudo > 0 AND balance_certificado_regalo > 0 LOOP
    -- algún procesamiento
END LOOP;

WHILE NOT expresión_lógica LOOP
    -- algún procesamiento
END LOOP;
```

FOR (variante con enteros)

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

Este forma de FOR crea un ciclo que itera sobre un rango de valores enteros. La variable *nombre* se define de manera automática como de tipo integer y existe solo dentro del ciclo. Las dos expresiones que generan los límites inferior y superior del rango, son evaluados una sola vez al entrar al ciclo. El paso de la iteración es 1 generalmente, pero es -1 cuando se especifica REVERSE.

Algunos ejemplos de ciclos enteros FOR:

```
FOR i IN 1..10 LOOP
    -- algún procesamiento aquí
    RAISE NOTICE 'i es %', i;
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- algún procesamiento aquí
END LOOP;
```

Si el límite inferior es mayor que el límite superior (o menor que, en el caso de REVERSE), el cuerpo del ciclo no se ejecuta en absoluto. No se genera un error.

Ciclos a Través de Resultados de Consultas

Usando un tipo diferente de un ciclo FOR, puede iterar a lo largo de los resultados de una consulta y manipular los datos correspondientes. La sintaxis es:

```
[<<etiqueta>>]
FOR registro_o_renglón IN consulta LOOP
    sentencias
END LOOP;
```

A las variables registro o renglón les es asignado, de manera sucesiva, cada renglón resultante de la *consulta* (la cual debe ser un comando **SELECT**) y el cuerpo del ciclo se ejecuta para cada renglón. He aquí un ejemplo:

```
CREATE FUNCTION cs_refresca_mvistas() RETURNS integer AS $$
DECLARE
    mvistas RECORD;
BEGIN
    PERFORM cs_log('Refrescando las vistas materializadas...');

    FOR mvistas IN SELECT * FROM cs_vistas_materializadas
                 ORDER BY llave_orden LOOP

    -- Ahora "mvistas" tiene un registro de cs_vistas_materializadas

    PERFORM cs_log('Refrescando vistas materializadas' ||
              quote_ident(mvistas.mv_nombre) || ' ...');
        EXECUTE 'TRUNCATE TABLE ' || quote_ident(mvistas.mv_nombre);
    EXECUTE 'INSERT INTO ' ||
       quote_ident(mvistas.mv_nombre) || ' ' || mvistas.mv_consulta;
    END LOOP;

  PERFORM cs_log('Terminado el refresco de las vistas
                    materializadas.');
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Si el ciclo se termina por una sentencia `EXIT`, el valor del último renglón asignado se encuentra accesible después del ciclo.

La sentencia `FOR-IN-EXECUTE` es otra manera de iterar sobre los renglones:

```
[<<etiqueta>>]
FOR registro_o_renglón IN EXECUTE texto_expresión LOOP
    sentencia
END LOOP;
```

Esta es similar a la forma anterior, excepto que la sentencia **SELECT** fuente se especifica como una expresion en cadena, la cual es evaluada y replaneada en cada entrada al ciclo `FOR`. Esto le permite al programador el seleccionar la velocidad de una consulta previamente planeada o la flexibilidad de una consulta dinámica, exactamente como una simple sentencia **EXECUTE**.

> **Nota:** El analizador de `PL/pgSQL` distingue actualmente las dos clases de ciclos `FOR` (enteros o resultados de consultas) revisando si los símbolos `..` aparecen fuera de los paréntesis entre `IN` y `LOOP`. Si no se observan los símbolos `..` entonces se presume que es un ciclo sobre renglones. Si se escriben mal los símbolos `..` llevará seguramente a una queja entre las líneas de "loop variable of loop over rows must be a record or row variable", en lugar de un simple error de sintaxis que usted esperaría obtener.

Atrapar los Errores

Por omisión, cualquier error que ocurra en una función `PL/pgSQL` aborta la ejecución de la función, en consecuencia, también la transacción que la envuelve. Usted puede atrapar los errores y recuperarse de ellos usando un bloque **BEGIN** con una cláusula `EXCEPTION`. La sintaxis es una extensión de la sintaxis normal de un bloque **BEGIN**.

```
[ <<etiqueta>> ]
[ DECLARE
    declaraciones ]
```

```
BEGIN
    sentencia
EXCEPTION
    WHEN condición [ OR condición ... ] THEN
        sentencias_del_manejador
    [ WHEN condición [ OR condición ... ] THEN
        sentencias_del_manejador
      ... ]
END;
```

Si no ocurre un error, esta forma de bloque simplemente ejecuta todas las *sentencias*, y el control pasa a la siguiente sentencia después de END. Pero si ocurre un error dentro de las *sentencias*, se abandona el resto del procesamiento, y el control se pasa a la lista EXCEPTION. En la lista se busca la primera *condición* que coincida con el error ocurrido. Si hay una coincidencia, se ejecuta la *sentencia_del_manejador* correspondiente, y el control pasa a la siguiente sentencia después de END. Si no hay alguna coincidencia, el error se propaga como si la cláusula EXCEPTION no estuviera ahí en absoluto: el error puede ser atrapado por un bloque envolvente con EXCEPTION, o si no hay ninguno se aborta el proceso de la función.

Los nombres de las *condiciones* pueden ser cualquiera de los mostrados en el apéndice de códigos de error. Una nombre de categoría coincide con cualquier error dentro de su categoría. El nombre de la condición especial OTHERS coincide con cualquier error, excepto QUERY_CANCELED. (Es posible, pero en ocasiones improbable, atrapar QUERY_CANCELED por nombre). Los nombres de las condiciones no son sensibles a las mayúsculas.

Si ocurre un error dentro de las *sentencias_del_manejador*, no puede ser atrapado por esta cláusula EXCEPTION, pero si se propaga. Una cláusula EXCEPTION envolvente podría atraparla.

Cuando se atrapa un error por una cláusulaEXCEPTION, las variable locales de la función PL/pgSQL se mantienen como estaban cuando ocurrió el error, pero todos los cambios al estado persistente de la base de datos dentro del bloque se deshacen. Como ejemplo, considere este fragmento:

```
INSERT INTO miagenda(nombre, apellido) VALUES('Joaquín', 'Sabina');
BEGIN
    UPDATE miagenda SET nombre = 'Joe' WHERE apellido = 'Sabina';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'division_by_zero atrapado';
        RETURN x;
END;
```

Cuando el control alcanza la asignación a y, fallará con un error division_by_zero. Este será atrapado por la cláusula EXCEPTION. El valor regresado en la sentencia **RETURN** será el valor incrementado de x, pero los efectos del comando **UPDATE** habrán sido deshechos. El comando **INSERT** que precede al bloque no se deshace, sin embargo, al final resulta que la base de datos contiene Joaquín Sabina, no Joe Sabina.

> **Sugerencia:** Un bloque que contiene una cláusula EXCEPTION es significativamente más costoso, para entrar y salir, que un bloque sin ella. Por tanto, no use EXCEPTION, a menos que sea necesario.

Cursores

En lugar de ejecutar una consulta completa de inmediato, es posible definir un *cursor* que encapsule la consulta, para después leer los resultados de la consulta recuperando solo algunos renglones a la vez. Una razón para proceder de esta manera es el evitar el consumo de la memoria cuando el resultado contiene una gran cantidad de renglones. (Sin embargo, los usuarios de PL/pgSQL normalmente no tienen que preocuparse de esto, pues el ciclo FOR usa de manera automática un cursor interno para evitar problemas con la memoria). Un uso más interesante es el regresar una referencia a un cursor que ha creado una función. Esto proporciona una manera eficiente de regresar desde las funciones conjuntos grandes de renglones.

Declaración de las Variables de Cursores

Todos los accesos a los cursores en PL/pgSQL se hacen por medio de variables de cursores, las cuales son siempre del tipo de dato especial refcursor. Una manera de crear una variable de cursor es simplemente declararla como una variable de tipo refcursor. Otra manera es usar la sintaxis de declaración de cursor, que en general es:

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

(FOR puede se reemplazado por IS, por compatibilidad con Oracle). Los *argumentos*, si se especifican, son una lista separada por comas de pares *nombre tipo_de_dato* que definen los nombres que serán reemplazados por los valores de los parámetros en la consulta específica. Los valores actuales a sustituir por esos nombres serán especificados más tarde, al abrirse el cursor.

Algunos ejemplos

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) IS SELECT * FROM tenk1
                  WHERE unique1 = llave;
```

Las tres variables tienen el tipo de dato refcursor, pero la primera puede usarse con cualquier consulta, mientras que las segunda tiene ya una consulta específica *ligada* a ella, y la última tiene una consulta parametrizada ligada a ella. (`llave` será reemplazada por un parámetro con valor entero cuando el cursor se abra). Se dice que la variable `curs1` está *desligada* puesto que no está ligada a ninguna consulta en particular.

Apertura de Cursores

Antes de que un cursor pueda ser usado para recuperar renglones, debe ser *abierto*. (Esta es la acción equivalente al comando de SQL **DECLARE CURSOR**). PL/pgSQL tiene tres formas de la sentencia **OPEN**, dos de la cuales usan variables de cursores no ligadas, mientras que la tercera usa una variable de cursor ligada.

OPEN FOR SELECT

```
OPEN cursor_no_ligado FOR SELECT ...;
```

La variable de cursor se abre y se le entrega la consulta específica que debe ejecutar. El cursor no puede estar ya abierto, y tiene que haber sido declarado como un cursor no ligado (es decir, como una variable refcursor simple). La consulta **SELECT** es tratada de la misma manera que otras sentencia **SELECT** en PL/pgSQL: los nombres de las variable PL/pgSQL se sustituyen y el plan de la consulta es almacenado para un posible reuso.

Un ejemplo:

```
OPEN curs1 FOR SELECT * FROM foo WHERE llave= millave;
```

OPEN FOR EXECUTE

```
OPEN cursor_no_ligado FOR EXECUTE cadena_de_comando;
```

La variable de cursor se abre y se le entrega la consulta específica que debe ejecutar. El cursor no puede estar ya abierto, y tiene que haber sido declarado como un cursor no ligado (es decir, como una variable refcursor simple). La consulta se específica como una expresión de cadena de la misma manera que el comando **EXECUTE**. Como siempre, esto otorga flexibilidad puesto que la consulta puede variar de una corrida a otra.

Un ejemplo:

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

Apertura de un Cursor Ligado

```
OPEN cursor_ligado [ ( valores_de_argumentos ) ];
```

Esta forma de **OPEN** se utiliza para abrir una variable de cursor cuya consulta se encontraba ligada a ella cuando fue declarada. El cursor no puede estar ya abierto. Una lista de los valores de las expresiones de los argumentos actuales debe aparecer si y solo si el cursor fue declarado para manejar argumentos. Esos valores serán sustituídos en la consulta. El plan de la consulta para un cursor ligado siempre se considera almacenable (cachable); para este caso no existe un equivalente de **EXECUTE**.

Ejemplos:

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

Uso de los Cursores

Una vez que se ha abierto un cursor, puede ser manipulado con las sentencias descritas aquí.

Estas manipulaciones no deben ocurrir en la misma función que abrió el cursor con el cual iniciar. Usted puede regresar un valor refcursor de una función y permitir que quien la llamó opere sobre el cursor. (Internamente, un valor refcursor es simplemente el nombre de la cadena de un llamado portal que contiene la consulta activa para ese cursor. Este nombre puede pasar de un lado a otro, asignarse a otra variable refcursor y demás, sin perturbar el portal).

Todos los portales se cierran implícitamente al terminar la transacción. Por tanto, un valor refcursor se puede usar para referirse a un cursor abierto solo hasta el final de la transacción.

FETCH

```
FETCH cursor INTO blanco;
```

FETCH recupera el siguiente renglón del cursor y lo inserta en el blanco, el cual puede ser una variable de renglón, una variable de registro o una lista de variables simples separadas por comas, exactamente como en **SELECT INTO**. De igual forma que con **SELECT INTO**, la variable especial FOUND puede ser revisada para saber si se obtuvo o no un renglón.

Un ejemplo:

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

CLOSE

```
CLOSE cursor;
```

CLOSE cierra el portal subyacente a un cursor abierto. Esto puede utilizarse para liberar recursos antes de que se termine la transacción, o para liberar la variable del cursor que desea abrir de nuevo.

Un ejemplo:

```
CLOSE curs1;
```

Regreso de Cursores

Las funciones PL/pgSQL pueden regresar cursores a quien las llama. Es útil para regresar varios renglones o columnas, especialmente para conjuntos de resultados muy grandes. Para hacerlo la función abre el cursor y regresa el nombre del cursor a quien la invoca (o simplemente abre el cursor usando un nombre de portal especificado por o conocido por quien lo invoca). El invocante puede entonces recuperar (fetch) los renglones del cursor. El cursor puede ser cerrado por el invocante, o aquel será cerrado automáticamente cuando se cierre la transacción.

El nombre del portal usado para un cursor puede ser especificado por el programador o generado automáticamente. Para especificar un nombre de un portal basta con asignar una cadena a la variable refcursor antes de abrirlo. El valor de la cadena de la variable refcursor será usado por **OPEN** como el nombre de el portal subyacente. Sin embargo, si la variable refcursor es nula, **OPEN** generará de manera automática un nombre que no genere conflicto con algún portal existente, y lo asignará a la variable refcursor.

> **Nota:** Una variable de cursor ligado se inicializa al valor de la cadena que representa su nombre, de tal manera que el nombre del portal es el mismo que el nombre de la variable de cursor, a menos que el programador lo sobreescriba por asignación antes de abrir el cursor. Pero una variable de cursor no ligado toma inicialmente el valor por defecto de nulo, así que recibirá un nombre único generado automáticamente, a menos que se sobreescriba.

El siguiente ejemplo muestra una manera de que un nombre de un cursor puede ser suministrado por el invocante:

```
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;
```

En el siguiente ejemplo se utiliza la generación automática del nombre del 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();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

El siguiente ejemplo muestra una manera de regresar varios cursores desde una sola función:

```
  CREATE FUNCTION mifunc(refcursor, refcursor) RETURNS
                      SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM tabla_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM tabla_2;
    RETURN NEXT $2;
    RETURN;
END;
$$ LANGUAGE plpgsql;

-- se require estar en una transacción para usar los cursores.
BEGIN;

SELECT * FROM mifunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

Errores y Mensajes

Utilice la sentencia **RAISE** para reportar mensajes y levantar (raise) errores.

```
RAISE nivel 'formato' [, variable [, ...]];
```

Los niveles posibles son DEBUG, LOG, INFO, NOTICE, WARNING y EXCEPTION. EXCEPTION levanta un error (lo cual normalmente aborta la transacción actual); los demás niveles solo generan mensajes de diferentes niveles de prioridad. Que los mensajes de una prioridad particular sean reportados al cliente, escritos en la bitácora del servidor o ambos, se controla por las variables de configuración correspondientes.

Dentro de la cadena de formato, % se reemplaza por la siguiente representacion de cadena del argumento opcional. Debe escribir %% para generar una % literal. Observe que los argumentos opcionales actualmente deben ser variables simples, no expresiones, y el formato debe ser una cadena literal simple.

En este ejemplo, el valor de v_job_id remplazará el % en la cadena:

```
RAISE NOTICE 'LLamando a cs_create_job(%)', v_job_id;
```

Este ejemplo abortará la transacción con el mensaje de error dado:

```
RAISE EXCEPTION 'ID no existente --> %', user_id;
```

Actualmente **RAISE EXCEPTION** genera siempre el mismo código SQLSTATE, P0001, sin importar con que mensaje se invoque. Es posible atrapar esta excepción con EXCEPTION ... WHEN RAISE_EXCEPTION THEN ... pero no hay manera de diferenciar un **RAISE** de otro.

Procedimientos Desencadenantes o Disparadores (Triggers)

PL/pgSQL puede usarse para definir procedimientos disparadores (triggers). Un procedimiento trigger se crea con el comando **CREATE FUNCTION**, declarándola como una función sin argumentos y un tipo de retorno trigger. Observe que la función debe ser declarada sin argumentos, aunque espere recibir argumentos especificados en **CREATE TRIGGER** — los argumentos de los triggers se pasan por medio de TG_ARGV, tal como se describe adelante.

Cuando una función PL/pgSQL es llamada como un trigger, se crean automáticamente varias variables especiales en el bloque más externo. Ellas son:

NEW

Tipo de dato RECORD; variable que almacena el nuevo renglón de la base de datos para las operaciones de **INSERT**/**UPDATE** para triggers a nivel de renglón. Esta variable es NULL en triggers a nivel de sentencia.

OLD

Tipo de dato RECORD; variable que almacena el renglón viejo de la base de datos para las operaciones de **UPDATE**/**DELETE** para triggers a nivel de renglón. Esta variable es NULL en triggers a nivel de sentencia.

TG_NAME

Tipo de dato name; variable que contiene el nombre del trigger lanzado actualmente.

`TG_WHEN`

Tipo de dato text; una cadena con `BEFORE` o `AFTER` dependiendo de la definición del trigger.

`TG_LEVEL`

Tipo de dato text; una cadena con `ROW` o `STATEMENT` dependiendo de la definición del trigger.

`TG_OP`

Tipo de dato text; una cadena con `INSERT`, `UPDATE` o `DELETE` la cual informa acerca de la operación para la que fue lanzado el trigger.

`TG_RELID`

Tipo de dato oid; el ID del objeto de la tabla que causó la invocación del trigger.

`TG_RELNAME`

Tipo de dato name; el nombre de la tabla que causó la invocación del trigger.

`TG_NARGS`

Tipo de dato integer; la cantidad de argumentos dados al procedimiento trigger en la sentencia **CREATE TRIGGER**.

`TG_ARGV[]`

Tipo de dato arreglo de text; los argumentos de la sentencia **CREATE TRIGGER**. El índice cuenta desde 0, Los índices inválidos (menores que 0 o mayores o iguales a `tg_nargs`) generan un valor nulo.

Una función trigger debe regresar ya sea un NULL o un valor registro/renglón que tenga exactamente la misma estructura de la tabla para la cual se inició el trigger.

Los triggers de nivel renglón lanzados `ANTES (BEFORE)` pueden regresar un nulo para notificar al administrador de triggers que debe saltarse el resto de la operación para este renglón (es decir, los triggers subsecuentes no deben dispararse, y el **INSERT/UPDATE/DELETE** no ocurre para este renglón). Si se regresa un valor no nulo entonces la operación procede con ese valor de renglón. El regresar un valor de renglón diferente del valor original de `NEW` altera el renglón que será insertado o actualizado (pero no tiene un efecto directo en el caso **DELETE**). Para alterar el renglón que se va a almacenar, es posible reemplazar valores particulares directamente en `NEW` y regresar el `NEW` modificado o construir completamente el nuevo registro que se va a regresar.

El valor de retorno de un trigger de nivel sentencia `BEFORE` o `AFTER` o un trigger de nivel renglón siempre se ignora; también puede ser nulo. Sin embargo, cualquiera de estos tipos de triggers pueden abortar la operación entera al levantarse un error.

El Ejemplo 1-1 muestra un ejemplo de un procedimiento trigger en `PL/pgSQL`.

Ejemplo 1-1. Un Procedimiento Trigger `PL/pgSQL`

Este ejemplo de trigger asegura que en cualquier momento en que se inserta o actualiza un renglón en una tabla, se estampen en el renglón el nombre del usuario y el tiempo actuales. También verifica que exista el nombre del usuario y que el salario tenga un valor positivo.

```
CREATE TABLE empleados (
    empleado_nombre    text NOT NULL,
    salario            integer
    ultima_fecha timestamp,
```

```
    ultimo_usuario text
);

CREATE FUNCTION empleado_stamp() RETURNS trigger AS $empleado_stamp$
    BEGIN
        -- Revisa que se existan el nombre y el salario
        IF NEW.empleado_nombre IS NULL THEN
            RAISE EXCEPTION 'empleado_nombre no puedo ser nulo';
        END IF;
        IF NEW.salario IS NULL THEN
            RAISE EXCEPTION '% no puede tener un salario nulo',
                        NEW.empleado_nombre;
        END IF;

        -- Nadie debe pagar por trabajar
        IF NEW.salario < 0 THEN
            RAISE EXCEPTION '% no puede tener un salario negativo',
                        NEW.empleado_nombre;
        END IF;

        -- Recuerda quien y cuando cambió la nómina
        NEW.ultima_fecha := 'now';
        NEW.ultimo_usuario := current_user;
        RETURN NEW;
    END;
$empleado_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER empleado_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE empleado_stamp();
```

Otra manera de llevar una bitácora en una tabla implica el crear una nueva tabla que contenga un renglón por cada insert, update o delete que ocurra. Esta técnica puedes considerarse como una auditoría a los cambios a una tabla. El Ejemplo 1-2 muestra un ejemplo de un procedimiento trigger de auditoría en PL/pgSQL.

Ejemplo 1-2. Un Procedimiento Trigger para Auditoría en PL/pgSQL

Este trigger de ejemplo asegura que cualquier inserción, actualización o borrado de un renglón en la tabla emp, sea registrada (i.e., auditada) en la tabla emp_audit. La hora actual y el nombre del usuario se estampan en el renglón, junto con el tipo de operación ejecutada.

```
CREATE TABLE empleados (
    empleado_nombre    text NOT NULL,
    salario            integer
);

CREATE TABLE empleados_audit(
    operacion         char(1)   NOT NULL,
    horafecha         timestamp NOT NULL,
    userid            text      NOT NULL,
    empleado_nombre   text      NOT NULL,
    salario integer
);

CREATE OR REPLACE FUNCTION procesa_empleados_audit() RETURNS TRIGGER AS '
    BEGIN
    --
    -- Crea un registro en empleado_audit para reflejar las operaciones
    -- realizadas en empleados utiliza las variables especiales TG_OP
    -- para efectuar la operación
        IF (TG_OP = "DELETE") THEN
            INSERT INTO empleados_audit SELECT "D", now(), user, OLD.*;
            RETURN OLD;
```

```
        ELSIF (TG_OP = ”UPDATE”) THEN
            INSERT INTO empleados_audit SELECT ”U”, now(), user,
                    NEW.empleado_nombre, NEW.salario;
            RETURN NEW;
        ELSIF (TG_OP = ”INSERT”) THEN
            INSERT INTO empleados_audit SELECT ”I”, now(), user, NEW.*;
            RETURN NEW;
        END IF;
    RETURN NULL; -- el resultado es ignorado puesto que este
                 -- es un trigger AFTER
    END;
’ language ’plpgsql’;

CREATE TRIGGER empleados_audit AFTER INSERT OR UPDATE OR
        DELETE ON empleados FOR EACH ROW EXECUTE PROCEDURE
    procesa_empleados_audit();
```

Un uso de los triggers es el mantener una tabla como resumen de otra tabla. El resumen resultante puede usarse en lugar de la tabla original para ciertas consultas — comúnmente con una gran ahorro en el tiempo de ejecución. Esta técnica se suele usar en Data Warehousing, en donde las tablas se datos medidos u observados (llamadas tablas factuales) pueden ser extremadamente grandes. El Ejemplo 1-3 muestra un ejemplo de un procedimiento trigger en PL/pgSQL que mantiene una tabla resumen para una tabla factual en un data warehouse.

Ejemplo 1-3. Un Procedimiento Trigger Para Mantener Una Tabla Resumen en PL/pgSQL

El esquema que se detalla aquí está basado parcialmente en el ejemplo *Grocery Store* de *The Data Warehouse Toolkit* por Ralph Kimball.

```
--
-- Tablas principales - time dimension y sales fact.
--
CREATE TABLE time_dimension (
    time_key                    integer NOT NULL,
    day_of_week                 integer NOT NULL,
    day_of_month                integer NOT NULL,
    month                       integer NOT NULL,
    quarter                     integer NOT NULL,
    year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key                    integer NOT NULL,
    product_key                 integer NOT NULL,
    store_key                   integer NOT NULL,
    amount_sold                 numeric(12,2) NOT NULL,
    units_sold                  integer NOT NULL,
    amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Tabla resumen - sales by time.
--
CREATE TABLE sales_summary_bytime (
    time_key                    integer NOT NULL,
    amount_sold                 numeric(15,2) NOT NULL,
    units_sold                  numeric(12) NOT NULL,
    amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON
                        sales_summary_bytime(time_key);
```

```
--
-- Función y trigger para corregir la(s) columna(s)
-- resumida(s) en UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime()
                    RETURNS TRIGGER AS $maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN

        -- Calcula la(s) cantidad(es) del incremento/decremento.
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;

        ELSIF (TG_OP = 'UPDATE') THEN

            -- Prohibe las actualizacioes que cambie la time_key -
            -- (probabablemente no muy oneroso, puesto que
            -- la mayoría de los cambios se harán como
            -- DELETE + INSERT).
            IF ( OLD.time_key != NEW.time_key) THEN
                RAISE EXCEPTION 'La actualización de time_key: % -> %
                 no está permitida', OLD.time_key, NEW.time_key;
            END IF;

            delta_time_key = OLD.time_key;
            delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
            delta_units_sold = NEW.units_sold - OLD.units_sold;
            delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

        ELSIF (TG_OP = 'INSERT') THEN

            delta_time_key = NEW.time_key;
            delta_amount_sold = NEW.amount_sold;
            delta_units_sold = NEW.units_sold;
            delta_amount_cost = NEW.amount_cost;

        END IF;


        -- Actualiza el renglón resumen con los nuevos valores.
        UPDATE sales_summary_bytime
            SET amount_sold = amount_sold + delta_amount_sold,
                units_sold = units_sold + delta_units_sold,
                amount_cost = amount_cost + delta_amount_cost
            WHERE time_key = delta_time_key;


        -- No debe haber ningún renglón con este time_key
        -- (e.g. ¡nuevos datos!).
        IF (NOT FOUND) THEN
            BEGIN
                INSERT INTO sales_summary_bytime (
                            time_key,
                            amount_sold,
                            units_sold,
                            amount_cost)
                    VALUES (
```

```
                                delta_time_key,
                                delta_amount_sold,
                                delta_units_sold,
                                delta_amount_cost
                               );
                EXCEPTION
                    --
                    -- Atrape la condición de competencia cuando dos
                    -- transacciones estén añadiendo datos para
                    -- un nuevo time_key.
                    --
                    WHEN UNIQUE_VIOLATION THEN
                        UPDATE sales_summary_bytime
                            SET amount_sold = amount_sold + delta_amount_sold,
                                units_sold = units_sold + delta_units_sold,
                                amount_cost = amount_cost + delta_amount_cost
                            WHERE time_key = delta_time_key;

                END;
            END IF;
            RETURN NULL;

        END;
    $maint_sales_summary_bytime$ LANGUAGE plpgsql;

    CREATE TRIGGER maint_sales_summary_bytime
    AFTER INSERT OR UPDATE OR DELETE ON sales_fact
        FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();
```