FUNDAMENTOS DE PROGRAMACIÓN EN LENGUAJE PL/pgSQL

Book · August 2019

CITATIONS
0

READS
3,287

2 authors, including:

Jorge Domínguez Chávez
Universidad Politécnica Territorial del Estado Aragua
66 PUBLICATIONS **22** CITATIONS

SEE PROFILE

FUNDAMENTOS DE PROGRAMACIÓN EN LENGUAJE PL/pgSQL

Este libro trata de funciones, procedimientos, paquetes y disparadores que es la forma común de programar del lado del servidor en PostgreSQL, es una introducción a la programación en PL/pgSQL.

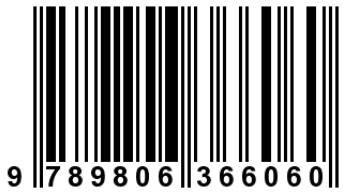
JORGE DOMÍNGUEZ CHÁVEZ

Publicado por

IEASS, Editores

ORCID: 0000-0002-5018-3242

ISBN 9789806366060



Publicado por IEASS, Editores
ieass@blogspot.com
Venezuela, 2019

La portada y contenido de este libro fue editado y maquetado en TeXstudio 2.12.10

FUNDAMENTOS DE PROGRAMACIÓN EN LENGUAJE PL/pgSQL

JORGE DOMÍNGUEZ CHÁVEZ

UNIVERSIDAD POLITÉCNICA TERRITORIAL DE ARAGUA

Venezuela, 2019

Índice general

Índice de figuras

Índice de tablas

Prólogo

PostgreSQL es un sistema gestor de base de datos objeto-relacional potente y flexible. Fue desarrollado en el departamento de Ciencias de la Computación en la Universidad de California en Berkeley, distribuido bajo licencia BSD y su código fuente está disponible libremente. Es el sistema de gestión de bases de datos de código abierto más potente de la actualidad.

PostgreSQL utiliza el modelo cliente/servidor y los multiprocesos en vez de multihilos para garantizar la estabilidad del sistema. Un fallo en uno de los procesos no afectará el resto y el sistema continuará funcionando.

PostgreSQL es un sistema de gestión de base de datos relacional orientada a objetos y libre, publicado bajo la licencia BSD.

Como muchos otros proyectos de código abierto, el desarrollo de PostgreSQL no es manejado por una sola empresa sino que es dirigido por una comunidad de desarrolladores y organizaciones comerciales las cuales trabajan en su desarrollo. Dicha comunidad es denominada el PGDG (PostgreSQL Global Development Group). Con el soporte para transacciones distribuidas, PostgreSQL se integra en un sistema distribuido formado por varios recursos (como una base de datos PostgreSQL, otra Oracle, una cola de mensajes IBM MQ JMS y un ERP SAP).

Los usuarios crean sus propios tipos de datos, los que pueden ser indexables gracias a la infraestructura GiST de PostgreSQL. Existen variadas aplicaciones, como los tipos de datos GIS, creados por el proyecto PostGIS.

Esta potencia y flexibilidad implican complejidad.

Para desarrollar programas, PostgreSQL incluye a PL/pgSQL, lenguaje de procedimientos, que combina las sentencias del lenguaje de programación para controlar el flujo del programa y las sentencias SQL que acceden a una base de datos PostgreSQL, para desarrollar programas para el procesamiento de datos. PL/pgSQL, se ejecuta en múltiples entornos, cada uno de los cuales le agrega diferentes ventajas y características.

Definimos que PL/pgSQL es un lenguaje imperativo del gestor de base de datos PostgreSQL.

Ejecuta comandos SQL mediante un lenguaje de sentencias imperativas y uso de funciones, dando mucho control automático que las sentencias SQL básicas.

Desde PL/pgSQL se realizan cálculos complejos y crear nuevos tipos de datos de usuario.

Como un verdadero lenguaje de programación, dispone de estructuras de control repetitivas y condicionales, además de la posibilidad de creación de funciones que son llamadas en sentencias SQL normales o ejecutadas en eventos de tipo TRIGGERS[1].

PL/pgSQL es un lenguaje de procedimientos que se incluye en PostgreSQL y con él se desarrollan programas para procesar datos combinando las sentencias PL/pgSQL que controlan el flujo del programa y SQL que acceden a una base de datos PostgreSQL.

PL/pgSQL es una herramienta implementada para la manipulación de datos de forma interna y externa, en nuestra aplicaciones. Además, está disponible en diversos entornos, cada uno de los cuales agrega diferentes ventajas y características.

PL/pgSQL es un sofisticado lenguaje de programación utilizado para acceder a PostgreSQL, desde diversos entornos, está integrado con el servidor de base de datos, tal que es procesado rápida y eficientemente.

Es común que los desarrolladores de aplicaciones no utilicen las prestaciones de las bases de datos relacionales modernas, en ocasiones simplemente por desconocer las ventajas que le ofrecen o por desconocer su manejo.

Dentro de PostgreSQL se desarrollan funciones en varios lenguajes.

El lenguaje PL/pgSQL es uno de los más utilizados dentro de PostgreSQL, debido a que guarda cierta similitud con PL/SQL de Oracle y a su facilidad de uso.

En este libro, que es introducción a la programación de funciones, procedimientos y disparadores en PL/pgSQL, se presentan la sintaxis, el control de flujo y características de dicho lenguaje; además, de presentan ejercicios y casos de estudio reales.

[1] Disparador, gatillo en castellano

Acrónimos

Palabra	Descripción
ACID	Atomicidad, Consistencia, aIslamiento, Durabilidad.
API	interfaz de programación de aplicaciones.
Bit	binary digit, dígito binario.
buffer	porción, de tamaño variable, de la MP utilizada en caché de páginas.
DLL	librerías de enlaces dinámicos para crear y definir nuevas bases de datos, campos e índices.
DML	librerías de gestión dinámicas para generar consultas para ordenar, filtrar y extraer datos de la base de datos.
GIS	sistema de información geográfica.
GUI	Abreviatura de Graphic User Interface o interfaz gráfica de usuario.
IDE	Entorno Integrado de Desarrollo, es una aplicación para desarrollar que va mucho más allá de lo que ofrece un simple editor.
MVCC	Acceso concurrente multiversión
POO	Programación Orientada a Objetos.
PL	lenguaje procedural o PL
SQL	lenguaje de consultas estructuradas.
SPI	interfaz de programación del servidor.
Puntero (pointer)	dirección de memoria
VCS	sistema de control de versiones o Version Control System.

1

Introducción

> *Una de las principales ventajas de ejecutar la programación en el servidor de base de datos es que las consultas y el resultado no tienen que ser transportadas entre el cliente y el servidor, ya que los datos residen en el propio servidor. Además, el gestor de base de datos puede planificar optimizaciones en la ejecución de la búsqueda y actualización de datos.*
>
> **Wikipedia**

Este libro se centra en la programación de funciones, procedimientos, paquetes y triggers[1] debido a que es la forma común de programar del lado del servidor en PostgreSQL; y como tal, es una introducción a la programación de funciones, procedimientos y triggers en PL/pgSQL, lenguaje procedural que combina las sentencias PL/pgSQL que controlan el flujo del programa y SQL que acceden a una base de datos PostgreSQL. Aprenderemos a reconocer un lenguaje de procedimiento, la utilidad de los mismos, y sentaremos las bases de su funcionamiento, que desarrollaremos a lo largo del curso con multitud de ejemplos prácticos.

PL/pgSQL es un lenguaje imperativo del gestor de base de datos PostgreSQL. Ejecuta comandos SQL mediante un lenguaje de sentencias imperativas y uso de funciones lo que genera un mayor control automático sobre el procesamiento de datos que el dado por las sentencias SQL básicas.

Desde PL/pgSQL se realizan cálculos complejos y crear nuevos tipos de datos de usuario. Como un verdadero lenguaje de programación, dispone de estructuras de control repetitivas y condicionales, además de la creación de funciones que son llamadas en sentencias SQL normales o bien ejecutadas en eventos de tipo trigger.

Es válido aclarar que todo lo que ocurre dentro de las funciones, procedimientos, paquetes y

[1] En castellano: disparador o gatillo.

disparadores definidos por el usuario en PostgreSQL lo hace de forma transaccional, es decir, se ejecuta todo o nada; de ocurrir algún fallo el propio PostgreSQL realiza un deshacer - ROLLBACK - las operaciones realizadas.*En PL/pgSQL se desarrollan programas para procesar datos con las sentencias PL/pgSQL que controlan el flujo del programa combinando con sentencias SQL que acceden a una base de datos PostgreSQL.*

Los objetivos para PL/pgSQL son ser un lenguaje procedural cargable que:

- *sea usado para crear funciones y disparadores,*
- *añada estructuras de control al lenguaje SQL,*
- *realice cálculos complejos,*
- *herede los tipos definidos por el usuario, funciones y operadores,*
- *sea definido para ser validado por el servidor y*
- *sea sencillo de utilizar.*

PL/pgSQL está incluido en PostgreSQL a partir de sus primeras versiones y es un lenguaje influenciado directamente de PL/SQL de Oracle; desarrollar en PL/pgSQL es agradable y rápido, especialmente si desarrollamos en otros lenguajes procedurales de bases de datos, como PL/SQL de Oracle.

PL/pgSQL, agrupa las consultas SQL, evitando la saturación del tráfico en la red entre el cliente y el servidor de bases de datos. Además, ofrece un grupo de ventajas adicionales entre las que destacan: incluir estructuras iterativas y condicionales, heredar todos los tipos de datos, funciones y operadores definidos por el usuario, mejorar el rendimiento de cálculos complejos y definir funciones para los triggers. Todo esto le otorga mayor potencialidad al combinar las ventajas de un lenguaje procedural y la facilidad de SQL.

PL/pgSQL es un lenguaje de procedimientos para desarrollar programas que procesen datos combinando las sentencias PL/pgSQL que controlan el flujo del programa y los comandos SQL que acceden a una base de datos PostgreSQL.

PL/pgSQL interpreta el código fuente en texto plano y produce un árbol de instrucciones binarias, internas. Esto lo hace la primera vez que es llamado (dentro de cualquier proceso principal). El árbol de instrucciones traduce la estructura de la sentencia 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 como SPI_prepare y SPI_saveplan. Vea Apéndice D, página 180.

Las siguientes visitas a dicha función reutilizan el plan preparado. Una vez que PL/pgSQL realiza un plan de consulta para una determinada función, reusará dicho plan durante toda la vida de la conexión a la base de datos. Esto significa una ganancia en el rendimiento, pero puede causar algunos problemas si altera dinámicamente su esquema de base de datos.

Una desventaja es que los errores en una expresión específica puede no ser detectados hasta que no se llegue a esa parte de la función durante la ejecución.

Este libro se centra en los dos primeros tipos de funciones, para ser utilizadas como procedimientos y para triggers, debido a que es la forma más común de programar en el servidor PostgreSQL.

1.1. Ventajas de usar PL/pgSQL

Mayor rendimiento (vea Sección 1.1.1, página 3) Soporte SQL (vea Sección 1.1.2, página 3) Portabilidad (vea Sección 1.1.3, página 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 comando SQL debe ser ejecutado individualmente por el servidor de bases de datos.

Esto significa que su aplicación cliente envía cada consulta al servidor de bases de datos, espera a que se procese, recibe el resultado, realiza cálculo, y luego envía otras consultas al servidor. Todo esto incurre en una comunicación entre procesos, el cual puede sobrecargar la red, si su cliente se encuentra en una máquina distinta al servidor de bases de datos.

Con PL/pgSQL agrupamos cálculo y consultas dentro del servidor de bases de datos, teniendo así la potencia de un lenguaje procedural y la sencillez de uso del SQL, ahorrando tiempo porque no tiene la sobrecarga de una comunicación cliente/servidor. Esto redunda 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; éstas, operan en cualquier plataforma donde PostgreSQL corra. Así, podremos reusar el código y reducir costos 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:

1. *Usar un editor de texto y recargar el archivo con psql*[2].
2. *Usar la herramienta de usuario de PostgreSQL: PgAccess o PgAdmin.*

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 programar y/o ejecutar las funciones. Si lo está haciendo de ésta forma, es una buena idea escribir la función usando CREATE OR REPLACE FUNCTION. De esta forma recarga el archivo para actualizar la definición de la función. Veamos:

```
1 CREATE
2 OR REPLACE FUNCTION testfunc(INTEGER) RETURNS INTEGER AS '
3 ....
4 end; '
5 LANGUAGE 'plpgsql';
```

Mientras ejecuta psql, puede cargar o recargar la definición de dicha función con `\i filename .sql` *e inmediatamente utilizar comandos SQL para probar la función.*

Otra forma de desarrollar en PL/pgSQL es usando las herramientas administrativas gráficas de PostgreSQL: PgAccess o PgAdmin III.

Las cuales son herramientas de propósito general para diseñar, mantener, y administrar las bases de datos de PostgreSQL.

Características incluidas:

- *Entradas SQL aleatorias.*
- *escapar comillas simples.*
- *facilitar la recreación y depuración de funciones.*
- *Pantallas de información y asistentes para bases de datos, tablas, índices, secuencias, vistas, programas de arranque, funciones y lenguajes.*
- *Preguntas y respuestas para configurar Usuarios, Grupos y Privilegios.*
- *Control de revisión con mejora de la generación de script.*
- *Configuración de las tablas.*
- *Asistentes para importar y exportar datos.*
- *Asistentes para migrar Bases de datos.*
- *Informes predefinidos en bases de datos, tablas, índices, secuencias, lenguajes y vistas.*

[2]Cliente de línea de comando, y que es una herramienta muy potente para consultar la base de datos y/o ejecutar scripts.

- *Múltiplataforma (soporta windows, unix, macintosh).*

PgAdmin se distribuye separadamente de PostgreSQL y puede ser descargado desde la internet.

2

Lenguajes Procedurales

PostgreSQL, a partir de la versión 6.3, soporta la definición de lenguajes procedurales. En el caso de una función o procedimiento definido en un lenguaje procedural, la base de datos no tiene un conocimiento implícito sobre como interpretar el código fuente de las funciones. El manejador en sí es una función de un lenguaje de programación compilada en forma de objeto compartido y cargado cuando es necesario.

2.1. Instalación de lenguajes procedurales

Un lenguaje procedural se instala en la base de datos en tres pasos:

1. *El objeto compartido que contienen el manejador del lenguaje debe ser compilado e instalado. Por defecto, el manejador para PL/pgSQL está integrado e instalado en el directorio de bibliotecas de la base de datos. Si el soporte de Tcl/Tk está instalado y configurado, el manejador para PL/Tcl está integrado e instalado en el mismo sitio.*

   La escritura de un nuevo lenguaje procedural [1] *está fuera del alcance de este libro.*

2. *El manejador debe ser declarado mediante la orden:*

```
1 CREATE FUNCTION handler_function_name () RETURNS OPAQUE AS
2 ' path-to-shared-object ' LANGUAGE 'C';
```

   El calificador especial de tipo devuelto OPAQUE le indica a la base de datos que esta función no devuelve uno de los tipos definidos en la base de datos ni un tipo compuesto, y no es directamente utilizable en una sentencia SQL.

3. *El PL debe ser declarado con la orden:*

```
1 CREATE [ TRUSTED ] PROCEDURAL LANGUAGE 'language-name'
2 HANDLER handler_function_name
3 LANCOMPILER 'description';
```

[1] Procedural language, PL

La palabra clave opcional TRUSTED indica si un usuario normal de la base de datos, sin privilegios de superusuario, puede usar este lenguaje para crear funciones y procedimientos activadores. Dado que las funciones de los PL se ejecutan dentro de la aplicación de base de datos, sólo deberían usarse para lenguajes que no puedan conseguir acceso a las aplicaciones internas de la base de datos, o al sistema de archivos. Los lenguajes PL/pgSQL y PL/Tcl son manifiestamente fiables en este sentido.

Veamos:

- *La siguiente orden le dice a la base de datos donde encontrar el objeto compartido para el manejador de funciones que llama al lenguaje PL/pgSQL*

```
1 CREATE FUNCTION plpgsql_call_handler () RETURNS OPAQUE AS
2 '/usr/local/pgsql/lib/plpgsql.so' LANGUAGE 'C';
```

- *La orden:*

```
1 CREATE TRUSTED PROCEDURAL LANGUAGE 'plpgsql'
2 HANDLER plpgsql_call_handler
3 LANCOMPILER 'plpgsql';
```

  define que la función manejadora de llamadas previamente declarada debe ser invocada por las funciones y procedimientos disparadores cuando el atributo del lenguaje es 'plpgsql'

Las funciones manejadoras de PL tienen una interfase de llamadas especiales distintas al de las funciones de lenguaje C normales. Uno de los argumentos pasados al manejador es el identificador del objeto en las entradas de la tabla pg_proc para la función que ha de ser ejecutada. El manejador examina varios catálogos de sistema para analizar los argumentos de llamada de la función y los tipos de dato que devuelve. El texto fuente del cuerpo de la función se encuentra en el atributo prosrc de pg_proc . Debido a esto, en contraste con las funciones de lenguaje C, las funciones PL pueden ser sobrecargadas, como las funciones del lenguaje SQL. Puede haber múltiples funciones PL con el mismo nombre de función, siempre que los argumentos de llamada sean distintos.

Los lenguajes procedurales definidos en la base de datos template1 se definen automáticamente en todas las bases de datos creadas subsecuentemente. Así que el administrador de la base de datos puede decidir que lenguajes están definidos por defecto.

En el caso de una función o procedimiento definido en un lenguaje procedural, la base de datos no tiene un conocimiento implícito sobre como interpretar el código fuente de las funciones. El manejador en sí es una función de un lenguaje de programación compilada en forma de objeto compartido, y cargado cuando es necesario.

2.2. Ejemplo

La siguiente orden le indica a la base de datos donde encontrar el objeto compartido para el manejador de funciones que llama al lenguaje PL/pgSQL

```
1 CREATE FUNCTION plpgsql_call_handler () RETURNS OPAQUE AS
2 '/usr/local/pgsql/lib/plpgsql.so' LANGUAGE 'C';
```

La orden

```
1 CREATE TRUSTED PROCEDURAL LANGUAGE 'plpgsql'
2 HANDLER plpgsql_call_handler
3 LANCOMPILER 'plpgsql';
```

define que la función manejadora de llamadas previamente declarada debe ser invocada por las funciones y procedimientos disparadores cuando el atributo del lenguaje es "plpgsql".

Las funciones manejadoras de PL tienen una interfase de llamadas especial distinta del de las funciones de lenguaje C normales. Uno de los argumentos dados al manejador es el identificador del objeto en las entradas de la tabla pg_proc para la función que ha de ser ejecutada. El manejador examina varios catálogos de sistema para analizar los argumentos de llamada de la función y los tipos de dato que devuelve. El texto fuente del cuerpo de la función se encuentra en el atributo prosrc de pg_proc. Debido a esto, en contraste con las funciones de lenguaje C, las funciones PL pueden ser sobrecargadas, como las funciones del lenguaje SQL. Puede haber múltiples funciones PL con el mismo nombre de función, siempre que los argumentos de llamada sean distintos.

Los lenguajes procedurales definidos en la base de datos template1 se definen automáticamente en todas las bases de datos creadas subsecuentemente. Así que el administrador de la base de datos puede decidir que lenguajes están definidos por defecto.

2.3. PL/pgSQL

PL/pgSQL es un lenguaje procedural cargable para el sistema de bases de datos PostgreSQL. Este paquete fue escrito originalmente por Jan Wieck.

Al usar PL/pgSQL es posible realizar cálculos, manejo de cadenas y consultas dentro del servidor de la base de datos, combinando la potencia de un lenguaje procedimental y la facilidad de uso de SQL, minimizando el tiempo de conexión entre el cliente y el servidor.

2.3.1. Panorámica

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

- *pueda usarse para crear funciones y procedimientos disparados por eventos,*
- *añada estructuras de control al lenguaje SQL,*
- *pueda realizar cálculos complejos,*
- *herede todos los tipos definidos por el usuario, las funciones y los operadores,*
- *pueda ser definido para ser fiable para el servidor,*

- *sea fácil de usar, El gestor de llamadas PL/pgSQL analiza el texto de las funciones y produce un árbol de instrucciones binarias interno la primera vez que la función es invocada por una aplicación. El bytecode producido es identificado por el manejador de llamadas mediante el ID de la función. Esto asegura que el cambio de una función por parte de una secuencia DROP/CREATE tiene efecto, sin tener que establecer una nueva conexión con la base de datos.*

Para todas las expresiones y sentencias SQL usadas en la función, el interprete de bytecode de PL/pgSQL crea un plan de ejecución preparado usando los gestores de SPI, funciones SPI_prepare() y SPI_saveplan(), ver Apéndice D, página 180. Esto se hace la primera vez que las sentencias individuales se procesan en la función PL/pgSQL.

Así, una función con código condicional que contenga varias sentencias que puedan ser ejecutadas, sólo prepara y almacena las opciones que realmente se usan durante el ámbito de la conexión con la base de datos.

Excepto en el caso de funciones de conversión de entrada/salida y de cálculo para tipos definidos, cualquier cosa que pueda definirse en funciones de lenguaje C puede ser hecho con PL/pgSQL. Es posible crear funciones complejas de calculo y después usarlas para definir operadores o usarlas en índices funcionales.

2.3.2. Descripción

PL/pgSQL (procedual language/postgreSQL) es una extensión del SQL para la creación de procedimientos y funciones al estilo de los lenguajes tradicionales de programación.

2.3.3. Estructura de PL/pgSQL

En el lenguaje PL/pgSQL, las palabras clave e identificadores pueden usarse en mayúsculas y/o minúsculas.

PL/pgSQL es un lenguaje orientado a bloques. Un bloque se define como:

```
1 [..label..]
2 [DECLARE
3 declarations]
4 BEGIN
5 statements
6 END;
```

Puede haber cualquier número de subloques en la sección de sentencia de un bloque.

Los subloques pueden usarse para ocultar variables a otros bloques de sentencias. Las variables declaradas en la sección de declaraciones se inicializan a su valor por defecto cada vez que se inicia el bloque, no cada vez que se realiza la llamada a la función.

Es importante no confundir el significado de BEGIN/END en la agrupación de sentencias de PL/pgSQl y las ordenes de la base de datos para control de transacciones.

Las funciones y procedimientos disparadores no pueden iniciar o realizar transacciones y PostgreSQL no soporta transacciones anidadas.

2.3.4. Declaraciones

Todas las variables, filas y columnas que se usen en un bloque o subloque deben ser declaradas en la sección de declaraciones del bloque, excepto las variables de control de ciclo en un FOR que se itere en un rango de enteros. Los parámetros dados a una función PL/pgSQL se declaran automáticamente con los identificadores usuales, $n.

Las declaraciones tienen la siguiente sintaxis:

```
1 name [ CONSTANT ] >typ> [ NOT NULL ] [ DEFAULT | := value ];
```

Esto declara una variable de un tipo base especificado. Si la variable es declarada como CONSTANT, su valor no podrá ser cambiado. Si se especifica NOT NULL, la asignación de un NULL producirá un error en tiempo de ejecución. Dado que el valor por defecto de todas las variables es el valor NULL de SQL, todas las variables declaradas como NOT NULL han de tener un valor por defecto.

El valor por defecto es evaluado cada vez que se invoca la función. Así que asignar NOW a una variable de tipo DATETIME hace que tome fecha y hora de la llamada a la función, no el momento en que fue compilada a bytecode. `name class%ROWTYPE;` *Esto declara una fila con la estructura de la clase indicada. Ésta, ha de ser una tabla existente, o la vista de una base de datos. Se accede a los campos de la fila mediante la notación de punto. Los parámetros de una función pueden ser de tipos compuestos (filas de una tabla completas). En ese caso, el correspondiente identificador $n será un tipo de fila, pero ha de ser referido usando la orden ALIAS*[2]*.*

Sólo los atributos de usuario de una fila de tabla son accesibles en la fila, no se puede acceder a OID o a los otros atributos de sistema (dado que la fila puede ser de una vista, y las filas de una vista no tienen atributos de sistema útiles).

Los campos de un tipo de fila heredan los tipos de datos, tamaños y precisiones de las tablas. `name RECORD;`

Los registros son similares a los tipos de fila, pero no tienen una estructura predefinida. Se emplean en selecciones y ciclos FOR, para mantener una fila de la actual base de datos en una operación SELECT. El mismo registro puede ser usado en diferentes selecciones. El acceso a un campo de registro cuando no hay una fila seleccionada resultará en un error de ejecución.

Las filas NEW y OLD en un disparador se pasan a los procedimientos como registros. Esto es necesario porque en PostgreSQL un mismo procedimiento desencadenado puede tener sucesos disparadores en diferentes tablas. `name ALIAS FOR $n;`

Para una mejor legibilidad del código, es posible definir un alias para un parámetro posicional de una función. Estos alias son necesarios cuando un tipo compuesto se pasa como

[2]Nombre abreviado o seudónimo o nickname dado a una variable, fila, columna, tabla y/o esquema PostgreSQL.

argumento a una función. La notación punto $1.salary como en funciones SQL no se permiten en PL/pgSQL, veamos la siguiente sentencia:

```
RENAME oldname TO newname;
```

Esto cambia el nombre de una variable, registro o fila. Esto es útil cuando NEW o OLD es referido por parte de otro nombre dentro de un procedimiento desencadenado.

2.3.5. Tipos de datos

Los tipos de una variable son cualquiera de los tipos básicos existentes en la base de datos. En la sección de declaraciones TYPE se define como:

- *Postgres-basetype*
- *variable*
- *class.field*

Donde variable es el nombre de una variable, previamente declarada en la misma función, que es visible en este momento.

class es el nombre de una tabla existente o vista, donde field es el nombre de un atributo.

El uso de class.field%TYPE hace que PL/pgSQl busque las definiciones de atributos en la primera llamada a la función, durante toda la vida de la aplicación final.

Supongamos que tenemos una tabla con un atributo char(20) y algunas funciones PL/pgSQL, que procesan el contenido por medio de variables locales. Ahora, decidimos que char(20) no es suficiente, cerramos la tabla, y la recreamos con un atributo definido como char(40), tras lo que restaura los datos. Pero, olvidamos las funciones. Los cálculos internos de éstas truncarán los valores a 20 caracteres. Si hubieran sido definidos usando las declaraciones class.field%TYPE automáticamente se adaptan al cambio de tamaño, o al nuevo esquema de la tabla que define el atributo como de tipo texto.

2.3.6. Expressions

Todas las expresiones en las sentencias PL/pgSQL son procesadas usando BACKENDS de ejecución. Las expresiones que puedan contener constantes pueden de requerir evaluación en tiempo de ejecución (como NOW para el tipo DATATIME), dado que es imposible para el analizador de PL/pgSQL identificar los valores constantes distintos de la palabra clave NULL. Todas las expresiones se evalúan internamente ejecutando una consulta:

```
SELECT expression
```

Usando el gestor SPI[3]*. En la expresión, los identificadores de variables son sustituidos por parámetros, y los valores reales de las variables son pasados al ejecutor a través de la matriz de parámetros. Todas las expresiones usadas en una función PL/pgSQL son preparadas de*

[3] interfaz de programación del servidor.

una sola vez, y guardadas una única vez.

La comprobación de tipos hecha por el analizador principal de PostgreSQL tiene algunos efectos secundarios en la interpretación de los valores constantes. En detalle, hay una diferencia entre lo que hacen las siguientes dos funciones:

```
CREATE FUNCTION logfunc1 (text) RETURNS datetime AS '
DECLARE
logtxt ALIAS FOR $1;
BEGIN
INSERT INTO logtable VALUES (logtxt, "now");
RETURN "NOW";

END;
' LANGUAGE 'plpgsql';
```

y

```
CREATE FUNCTION logfunc2 (text) RETURNS datetime AS '
DECLARE
logtxt ALIAS FOR $1;
curtime DATETIME;
BEGIN
curtime := "NOW";
INSERT INTO logtable VALUES (logtxt, curtime);
RETURN curtime;
END;
' LANGUAGE 'plpgsql';
```

En el caso de logfunc1(), el analizador principal de PostgreSQL prepara la ejecución de INSERT para la cadena NOW, la cual debe ser interpretada como una fecha, el campo objeto de 'logtable'tiene ese tipo. Así, hará una constante de ese tipo y su valor se emplea en todas las llamadas a logfunc1(), durante toda la vida útil de ese proceso.

En el caso de logfunc2(), el analizador principal de PostgreSQL no sabe cual es el tipo de NOW, por lo que devuelve un tipo de texto, que contiene la cadena NOW. Durante la asignación a la variable local 'curtime', el interprete PL/pgSQL asigna a esta cadena el tipo fecha, llamando a las funciones text_out() y datetime_in() para realizar la conversión.

Esta comprobación de tipos realizada por el analizador principal de PostgreSQL fue implementado antes de que PL/pgSQL estuviera totalmente terminado. Es una diferencia entre las versiones 6.3 y 6.4 de postgreSQL, y afecta a todas las funciones que usan la planificación realizada por el gestor SPI. El uso de variables locales en la manera descrita es la única forma actual de que PL/pgSQL interprete esos valores correctamente.

Si los campos del registro son usados en expresiones o sentencias, los tipos de datos de campos no deben cambiarse entre llamadas de una misma expresión. Tenga esto en cuenta cuando escriba procedimientos triggers que gestionen eventos en más de una tabla.

2.3.7. Sentencias

Cualquier cosa no comprendida por el analizador PL/pgSQL, tal como se ha especificado, será enviada al gestor de bases de datos, para su ejecución. La consulta resultante no devolverá dato(s).

2.3.8. Asignación

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

```
identifier := expression;
```

Si el tipo de dato resultante de la expresión no coincide con el tipo de dato de la variable, o tiene un tamaño o precisión conocido (como char(29)), el resultado es ajustado implícitamente por el interprete de bytecode de PL/pgSQL, para ello usa los tipos de las variables para las funciones de entrada y los tipos resultantes en las funciones de salida. Nótese que esto puede, potencialmente, producir errores de ejecución generados por los tipos de las funciones de entrada.

Una asignación de una selección completa en un registro o fila puede hacerse del siguiente modo: `SELECT expressions INTO target FROM ...;`

target puede ser un registro, una variable de fila o una lista separada por comas de variables y campo de de registros o filas.

Si una fila o una lista de variables se usa como objetivo, los valores seleccionados deben coincidir con la estructura de los objetivos o se producirá un error de ejecución. La palabra clave FROM puede preceder a cualquier calificador válido, agrupación, ordenación, etc. que pueda pasarse a una sentencia SELECT.

Existe una variable especial llamada FOUND de tipo booleano, que puede usarse inmediatamente después de SELECT INTO para comprobar si una asignación ha tenido éxito.

```
1 SELECT * INTO myrec FROM EMP WHERE empname = myname;
2 IF NOT FOUND THEN
3 RAISE EXCEPTION "empleado % no encontrado", myname;
4 END IF;
```

Si la selección devuelve múltiples filas, sólo la primera se mueve a los campos objetivo. Todas las demás se descartan.

2.3.9. Llamadas a otra función

Todas las funciones definidas en una base de datos PostgreSQL devuelven un valor. Por tanto, la forma normal de llamar a una función es ejecutar una consulta SELECT o realizar una asignación (que da lugar a un SELECT interno de PL/pgSQL). Pero hay casos en que no interesa saber los resultados de las funciones, como en la sentencia:

```
PERFORM query
```

La cual ejecuta un `SELECT query` *en el gestor SPI, y descarta el resultado. Los identificadores como variables locales son sustituidos en los parámetros.*

Devolviendo el resultado de la función `RETURN expression`

La función termina y el valor de expression se devolverá al ejecutor superior.

El valor devuelto por una función no puede quedar sin definir. Si el control alcanza el fin del bloque de mayor nivel de la función, sin encontrar una sentencia RETURN, ocurre un error de ejecución.

Las expresiones resultantes son ajustadas en los tipos devueltos por la función, tal como se ha descrito en el caso de las asignaciones.

2.3.10. Terminando la ejecución y mensajes

Como hemos indicado en los códigos anteriores, existe una sentencia RAISE que envía mensajes del sistema de registro de PostgreSQL.

```
RAISE level
FOR " [,
IDENTIFIER [...]];
```

Dentro del formato, usamos "%" para los subsecuentes identificadores, separados por comas. Los posibles niveles son DEBUG (suprimido en las bases de datos de producción), NOTICE (escribe en el registro de la base de datos y lo envía a la aplicación del cliente) y EXCEPTION (escribe en el registro de la base de datos y termina la transacción).

Condiciones

```
IF expression THEN
  statements
[ELSE
  statements]
END IF;
```

Donde expression devuelve un valor, que al menos, pueda ser adaptado en un tipo booleano.

Ciclos

Hay varios tipos de ciclos.

```
[--- label ---]
LOOP
statements
END LOOP;
```

Se trata de un ciclo no condicional que debe ser terminado de forma explicita, mediante una sentencia EXIT. La etiqueta opcional puede ser usado por las sentencias EXIT de otros ciclos anidados, para especificar el nivel del ciclo que ha de terminarse.

```
[ --- label --- ]
WHILE expression LOOP
statements
END LOOP;
```

Se trata de un ciclo condicional que se ejecuta mientras la evaluación de expression sea cierta.

```
[--- label ---]
FOR name IN [ REVERSE ]
express .. expression LOOP
statements
END LOOP;
```

Se trata de un ciclo que itera sobre un rango de valores enteros. La variable name se crea automáticamente con el tipo entero, y existe sólo dentro del ciclo.

Las dos expresiones dan el limite inferior y superior del rango y son evaluados sólo cuando se entra en el ciclo. El paso de la iteración es siempre 1.

```
[..label..]
FOR record | row IN select_clause LOOP
statements
END LOOP;
```

EL registro o fila se asigna a todas las filas resultantes de la cláusula de selección, y la sentencia se ejecuta para cada una de ellas. Si el ciclo se termina con una sentencia EXIT, la última fila asignada es aún accesible después del ciclo.

```
EXIT [ label ] [ WHEN expression ];
```

Si no se incluye label, se termina el ciclo más interno, y se ejecuta la sentencia que sigue a END LOOP. Si se incluye label debe ser la etiqueta del ciclo actual u de otro de mayor nivel. EL ciclo indicado se termina, y el control se pasa a la sentencia de después del END del ciclo o bloque correspondiente.

2.3.11. Procedimientos desencadenados

PL/pgSQL puede ser usado para definir procedimientos desencadenados por eventos.

Con la orden CREATE FUNCTION se crean estos procedimientos, igual que una función, pero sin argumentos, y devuelven un tipo OPAQUE.

Hay algunos detalles específicos de PostgreSQL cuando se usan funciones como procedimientos desencadenados. En primer lugar, disponen de algunas variables especiales que se crean en los bloques de mayor nivel de la sección de declaración. Para consultar las variables especiales, veamos el apéndice A, página **??**.

En segundo lugar, devuelven NULL o una fila o registro que contenga la estructura de la tabla que activa el procedimiento desencadenado. Los procedimientos desencadenados activados por AFTER devuelven un valor NULL, sin producir efectos secundarios. Los activados

por BEFORE indican al gestor que no realice la operación sobre la fila actual cuando devuelva NULL. En cualquier otro caso, la fila o registro devuelta sustituye a la fila insertada o actualizada. Es posible reemplazar valores individuales directamente en una sentencia NEW y devolverlos, o construir una nueva fila o registro y devolverla.

2.3.12. Excepciones

PostgreSQL no dispone de un modelo de manejo de excepciones muy elaborado. Cuando el analizador, el optimizador o el ejecutor deciden que una sentencia no puede ser procesada, la transacción completa es terminada y el sistema vuelve al ciclo principal para procesar la siguiente consulta de la aplicación cliente.

Es posible introducirse en el mecanismo de errores para detectar cuando sucede esto.

Lo que no es posible es saber qué ha causado en realidad la terminación (un error de conversión de entrada/salida, un error de punto flotante, un error de análisis). Es posible que la base de datos haya quedado en un estado inconsistente, por lo que volver a un nivel de ejecución superior o continuar ejecutando comandos puede corromper toda la base de datos. E incluso aunque se pudiera enviar la información a la aplicación cliente, la transacción ya se abría terminado, por lo que carecería de sentido el intentar reanudar la operación.

Por todo esto, lo que hace PL/pgSQL cuando se produce un error de ejecución durante la corrida de una función o procedimiento trigger es enviar mensajes de depuración a nivel DEBUG, indicando en qué función y donde (número de línea y tipo de sentencia) sucedió el error.

2.3.13. Código

A continuación se incluyen algunas funciones para demostrar lo fácil que es escribirlas en PL/pgSQL.

Un detalle importante al escribir funciones en PL/pgSQL es el manejo de la comilla simple. *Vea la tabla 2.1, página 17.*

Tabla 2.1: Esquema de Escapado de Comillas Simples.

No. de Comillas	Uso
1	Para iniciar/terminar cuerpos de función.
2	En asignaciones, estamentos SELECT, para delimitar cadenas, etc.
4	Cuando necesite dos comillas simples en su cadena de resultado sin terminar dicha cadena.
6	Cuando quiera comillas dobles en su cadena de resultado y terminar dicha cadena.
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.

El texto en las funciones en CREATE FUNCTION es una cadena de texto. Las comillas simples en el interior de una cadena literal deben de duplicarse o anteponerse de una barra invertida. Mientras tanto, duplique las comillas sencillas como en los códigos siguientes.

2.3.14. Algunas funciones sencillas en PL/pgSQL

A continuación, dos funciones similares a sus contrapartidas que estudiamos en el lenguaje C.

```
1 CREATE FUNCTION add_one (int4) RETURNS int4 AS '
2 BEGIN
3 RETURN $1 + 1;
4 END;
5 ' LANGUAGE 'plpgsql';
```

y la función:

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

2.3.15. Funciones PL/pgSQL para tipos compuestos

De nuevo, la siguiente función PL/pgSQL tienen su equivalente en lenguaje C.

```
1 CREATE FUNCTION c_overpaid (EMP, int4) RETURNS bool AS '
2 DECLARE
3 emprec ALIAS FOR $1;
```

```
4  sallim ALIAS FOR $2;
5  BEGIN
6  IF emprec.salary ISNULL THEN
7  RETURN "f";
8  END IF;
9  RETURN emprec.salary > sallim;
10 END;
11 ' LANGUAGE 'plpgsql';
```

2.3.16. Procedimientos desencadenados en PL/pgSQL

Los procedimientos desencadenados aseguran que, cada vez que se inserte o actualice un fila en la tabla, se incluye el nombre del usuario, la fecha y hora. Si se proporciona un nombre de empleado y que el salario tiene un valor positivo.

```
1  CREATE TABLE emp (
2  empname text,
3  salary int4,
4  last_date datetime,
5  last_user name);
6  CREATE FUNCTION emp_stamp () RETURNS OPAQUE AS
7  BEGIN
8  - Check that empname and salary are given
9  IF NEW.empname ISNULL THEN
10 RAISE EXCEPTION "empname cannot be NULL value";
11 END IF;
12 IF NEW.salary ISNULL THEN
13 RAISE EXCEPTION "% cannot have NULL salary", NEW.empname;
14 END IF;
15 - Who works for us when she must pay for?
16
17 IF NEW.salary < 0 THEN
18 RAISE EXCEPTION "% cannot have a negative salary", NEW.empna
19 END IF;
20 - Remember who changed the payroll when
21 NEW.last_date := "NOW";
22 NEW.last_user := getpgusername();
23 RETURN NEW;
24 END;
25 ' LANGUAGE 'plpgsql';
26 CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
27 FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

2.4. Diferencias entre lenguajes procedural y no procedural

2.4.1. Programación procedural

En la programación procedural se describe, paso a paso, un conjunto de instrucciones a ejecutar para cambiar el estado del programa y encontrar la solución, es decir, elaborar un algoritmo donde se describen los pasos necesarios para solucionar el problema.

2.4.2. Programación no procedural

La programación no procedural dice QUÉ, en lugar de CÓMO. Cuando las cosas se pueden ordenar en una serie, es natural preguntar si hay un primer o un último elemento en la serie.

Una vez que un programa se hace más pequeño que otro y desarrolla la misma función, es natural preguntarse si hay una manera de hacer el programa más pequeño para que haga lo mismo. De manera similar, la noción de un programa rápido nos hace buscar el más eficiente y eficaz. Una manera de juzgar a un lenguaje de programación es verificar que sea de más alto-nivel que otro. Un lenguaje es de más alto nivel que otro si codificamos el mismo programa con menos trabajo. Otra manera de expresar esto es que un lenguaje es de más alto nivel cuando es menos procedural. En otras palabras, en un lenguaje de más alto nivel nos concentramos más en QUÉ se está haciendo y no en CÓMO se está haciendo.

2.4.3. Programación funcional

En ciencias de la computación, la programación funcional es un paradigma de programación declarativa basado en el uso de funciones matemáticas, en contraste con la programación imperativa, que enfatiza los cambios de estado mediante la mutación de variables. La programación funcional tiene sus raíces en el cálculo lambda, un sistema formal desarrollado en los años 1930 para investigar la definición de función, la aplicación de las funciones y la recursión. Muchos lenguajes de programación funcionales pueden ser vistos como elaboraciones del cálculo lambda.

En la práctica, la diferencia entre una función matemática y la noción de una "función" utilizada en la programación imperativa, es que las funciones imperativas pueden tener efectos secundarios, como cambiar el valor de cálculos realizados previamente. Por esta razón, carecen de transparencia referencial; es decir, la misma expresión sintáctica puede resultar en valores diferentes en varios momentos de la ejecución del programa. Con código funcional, en contraste, el valor generado por una función depende exclusivamente de los argumentos alimentados a la función. Al eliminar los efectos secundarios se puede entender y predecir el comportamiento de un programa mucho más fácilmente. Esta es una de las principales motivaciones para utilizar la programación funcional.

2.4.4. Programación declarativa

En la programación declarativa las sentencias describen el problema a solucionar, pero no las instrucciones necesarias para solucionarlo. Esto último se realiza mediante mecanismos internos de inferencia de información a partir de la descripción realizada.

Consideremos el siguiente caso: requerimos ordenar un arreglo. ¿Cómo expresamos este problema en un lenguaje no procedural? Tenemos que describir qué significa "Ordenar un arreglo". Decimos que B es un orden de A si, y sólo si, B es una permutación de A y B está ordenado. También tenemos que describir lo que significa una "permutación de un arreglo" y lo que significa un "arreglo ordenado". Esto último dice que B está ordenado si `B[i] <= B[j]` *para cada* `i<j`*. Entonces, es responsabilidad del sistema no procedural determinar cómo crear un arreglo B que sea una permutación ordenada de un arreglo A dado.*

2.4.5. Programación local y deducción automática

La programación no procedural es una de las área de investigación de la inteligencia artificial[4]*. La prueba automática de teoremas. El objetivo es desarrollar programas que puedan construir pruebas formales de proposiciones establecidas en un lenguaje simbólico. Entonces a modo de síntesis decimos que en la programación imperativa describe, paso a paso, un conjunto de instrucciones que deben ejecutarse para variar el estado del programa y encontrar la solución, es decir, un algoritmo en el que se describen los pasos necesarios para solucionar el problema. En contraste, en la programación declarativa las sentencias que se utilizan lo que hacen es describir el problema que se quiere solucionar, pero no las instrucciones necesarias para solucionarlo. Esto último se realiza mediante mecanismos internos de inferencia de información a partir de la descripción realizada.*

[4] AI Artificial Intelligence

3

El programa

Un problema tiene dos soluciones: una lógica y la otra de programación.

La primera se resuelve mediante un algoritmo y la segunda a través de un lenguaje de programación.

Programar no es sólo codificar. Básicamente implica estructurar la solución de un problema y, luego refinarla, paso a paso. Cuando refinamos en un nivel de profundidad suficiente, creamos un algoritmo. Entonces, traducimos cada paso del algoritmo en código para un programa de computación.

Tienes un problema que necesita ser resuelto. Necesitas escribir una secuencia de operaciones, a nivel general, que se ejecutan para resolver el problema. Empiezas enfocándote en una operación a la vez y encuentras que las operaciones deben ser refinadas en pasos más detallados. Procedemos al siguiente nivel y refinamos los pasos propuestos. Este proceso de refinamiento se repite hasta llegar a un nivel de suficiente profundidad para empezar a codificar.

Crear un algoritmo para resolver un problema es, en general, la tarea más laboriosa e importante de la programación. Muchos desarrolladores cometen el error de codificar de una vez, lo cual hace que se enfoquen en detalles de programación y olviden el problema a resolver. Esto produce un código no estructurado e ineficiente difícil de entender y mantener.

Por eso enfatizamos que debemos estructurar lógicamente nuestros pensamientos y construir un buen algoritmo antes de codificar.

Para programar necesitas un lenguaje de programación, en este caso PL/pgSQL, con el cual se desarrollan programas para procesar información combinando las sentencias de procedimiento PL/pgSQL que controlan el flujo del programa y las sentencias SQL que acceden a la base de datos PostgreSQL.

Aprender a programar en PL/pgSQL es el objetivo principal de este libro.

Al igual que en Lenguaje C, todo programa es una función y toda función es un programa; y toda función devuelve un resultado. Así que aprenderemos a desarrollar funciones, insumo básico de la programación PL/pgSQL.

Todo programa PL/pgSQL tiene dos parte: una declarativa y un cuerpo de programa. En la parte declarativa se declaran las variables que serán utilizadas en el programa; éstas, debe tener un tipo, el cual NO puede ser cambiado durante la ejecución de dicho programa; el cuerpo del programa contiene las sentencias y comandos para su ejecución.

Veamos nuestro primer programa, HolaMundo:

```
1 debian=# CREATE FUNCTION HolaMundo() RETURNS CHAR
2 debian-# AS ' BEGIN RETURN "Hola Mundo PostgreSQL" ; END; '
3 debian-# LANGUAGE 'plpgsql';
4 CREATE
```

Esta función tiene tres partes:

- *El encabezado que define el nombre de la función y el tipo de retorno.*
- *El cuerpo de la función, es una cadena de texto (por lo tanto, siempre va entre comillas dobles).*
- *La especificación del lenguaje utilizado.*

La función creada tiene las mismas características que las funciones integradas en PostgreSQL.

Mediante el comando SELECT la invocamos:

```
1 debian=# SELECT HolaMundo();
2 holamundo
3 (1 row)
```

Este programa es un procedimiento almacenado como una unidad de programa en una base de datos. Con PL/pgSQL creamos muchos tipos de unidades de acceso a base de datos, incluyendo bloques anónimos[1]*, procedimientos, funciones y paquetes.*

La función se elimina mediante el comando DROP FUNCTION.

```
1 debian=# DROP FUNCTION HolaMundo();
2 DROP
```

Antes de aprender a programar con PL/pgSQL es necesario comprender la programación básica, la construcción de una unidad de programa y los comandos que tiene.

Veamos un punto técnico, como:

```
1 CREATE FUNCTION populate() RETURNS INTEGER AS '
2 DECLARE
3 -- Declarations
4 BEGIN
5 PERFORM my_function();
```

[1]Un bloque anónimo no tiene nombre y no se almacena de forma permanente como un archivo o en una base de datos PostgreSQL, simplemente envía el bloque al servidor de base de datos para que lo procese en tiempo de ejecución

```
6 END;
7 ' LANGUAGE 'plpgsql';
```

Si ejecutamos la función anterior, la cual hace referencia al OID[2] para my_function() en el plan de consulta producido para el sentencia PERFORM. Luego, si eliminamos y volvemos a crear la función my_function(), entonces populate() no encuentra la función my_function(). Debemos 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, construimos las consultas dinámicas usando la sentencia de PL/pgSQL EXECUTE, Ver Sección 6.6, página 75 -al costo de construir un nuevo plan de consultas en cada ejecución.

> ***NOTA:*** *La sentencia EXECUTE de PL/pgSQL no está relacionada con EXECUTE soportado por el motor PostgreSQL. Las sentencias del motor no puede ser usadas en las funciones PL/pgSQL (y no es necesario hacerlo).*

3.1. Los comentarios

Para hacer comentarios en un programa PL/pgSQl. Un par de guiones "–" comienza un comentario que se extiende hasta el fin de la linea; mientras que los caracteres "/" comienzan un bloque de comentarios que se extiende hasta que se encuentre un "*/". Los bloques de comentarios no pueden anidarse pero un par de guiones pueden encerrarse en un bloque de comentario, o ocultar los limitadores de estos bloques.*

3.2. La asignación

Hacemos una asignación con:

```
identifier := expresión;
```

Para más información, vea 2.3.8, página 13.

Existe una variable especial llamada FOUND, tipo booleano, que puede usarse inmediatamente después de SELECT INTO para comprobar si una asignación ha tenido éxito. Veamos el siguiente código, que busca un empleado en la tabla empname:

```
1 SELECT * INTO myrec FROM EMP WHERE empname = myname;
2 IF NOT FOUND THEN
3   RAISES EXCEPTION "empleado % no encontrado", myname;
```

[2]En los modelos OO, la identidad se representa con el identificador de objeto, OID en inglés. Teóricamente, es un objeto único e irrepetible en el tiempo y el espacio; permite que dos objetos idénticos puedan diferenciarse, no es importante que el usuario conozca los OID, lo importante es que los diferencie el sistema.

```
4 END IF;
```

Si la selección devuelve múltiples filas, sólo la primera se asigna a los campos objetivo; el resto se descartan.

Si no encuentra al empleado, la sentencia RAISES envía un mensaje de error. Vea la sección 3.11, página 58.

3.3. Las variables y constantes

En la sección de declaraciones, declaramos a las variables y constantes que preceden a un bloque. También, inicializamos sus valores por defecto cada vez que el bloque es introducido, y no sólo una vez por cada llamada a la función.

En el código:

```
1  CREATE FUNCTION algunafuncion() RETURNS INTEGER AS '
2  DECLARE
3  cantidad INTEGER := 30;
4  BEGIN
5  RAISES NOTICE "La cantidad aquí es %", cantidad;
6  cantidad := 50;
7  --
8  -- Crea un subbloque
9  --
10 DECLARE
11 cantidad INTEGER := 80;
12 BEGIN
13 RAISES NOTICE "La cantidad aquí es %",cantidad;
14 END;
15 RAISES NOTICE "La cantidad aquí es %",cantidad;
16 RETURN cantidad;
17 END;
18 ' LANGUAGE 'plpgsql';
```

las variables, constantes, filas y registros usados son declarados en la sección de declaraciones del bloque.

Las variables PL/pgSQL son del tipo de dato SQL, como INTEGER, VARCHAR y CHAR.

Veamos algunas declaraciones de variables:

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

La sintaxis general para una declaración de variables es:

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

La cláusula DEFAULT especifica el valor inicial asignado a la variable cuando el bloque es ejecutado. Si DEFAULT no es facilitada, la variable es inicializada al valor SQL NULL.

La opción CONSTANT previene la asignación de otros valores a la variable, así que su valor se mantiene durante la duración del bloque. Si se especifica NOT NULL, una asignación de un valor NULL genera 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 ejecutado. Así, la asignación de NOW a una variable de tipo TIMESTAMP provoca que la variable de la función tenga la fecha y hora actual[3]*, y no la fecha y hora de su compilación.*

Veamos un código:

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

3.3.1. Constantes y variables con valores por omisión

La declaraciones tienen las siguiente sintaxis:

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

El valor de una variable declarado como CONSTANT no puede ser modificado.

Si especificamos NOT NULL, la asignación de un valor NULL causa un error en tiempo de ejecución. Puesto que el valor por omisión de todas las variables de SQL es el valor NULL, las declaradas como NOT NULL deben contar con un valor por omisión específico

Veamos el código:

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

Donde:

- *La cláusula CONSTANT especifica que la variable es constante, por lo que el valor inicial asignado a ella se mantendrá, a menos que la variable especificada sea inicializada con el valor nulo.*
- *La cláusula NOT NULL especifica que la variable no puede tener asignado un valor nulo (generando un error en tiempo de ejecución en caso de que ocurra); todas las variables declaradas como NOT NULL deben tener especificado un valor no nulo.*
- *La cláusula DEFAULT (o :=) asigna un valor a la variable.*

[3] Fecha y hora del sistema

Pueden declarase variables de la forma mostrada en las siguientes lineas:

- *impuesto CONSTANT numeric := 0.3; – variable de tipo numeric con un valor constante – de 0.3*
- *incremento integer DEFAULT 31; – variable de tipo entero con valor 31*
- *region varchar := "Occidente"; – variable de tipo varchar con el valor "Occidente" asignado*
- *fecha date NOT NULL:= '1984-08-08'; – variable de tipo fecha no nula y con valor 1984-08-08*

3.3.2. Variables pasadas a las funciones

Las variables que se pasan a las funciones son denominadas con los identificadores $1, $2, etc. (el máximo es 16).

Veamos algunos códigos:

```
1 CREATE FUNCTION iva_venta(REAL) RETURNS REAL AS '
2 DECLARE
3 subtotal ALIAS FOR $1;
4 BEGIN
5 return subtotal * 1.15;
6 END;
7 ' LANGUAGE 'plpgsql';
8 \begin{lstlisting}
```

y

```
1 CREATE FUNCTION instr(VARCHAR,INTEGER) RETURNS INTEGER AS '
2 DECLARE
3 v_string ALIAS FOR $1;
4 index ALIAS FOR $2;
5 BEGIN
6 -- Algunos cáculos van aquí.
7 END;
8 ' LANGUAGE 'plpgsql';
```

3.4. Atributos

Usando los atributos %TYPE and %ROWTYPE, declaramos variables con el mismo tipo de dato o estructura de otro item de la base de datos (como un campo de una tabla).

%TYPE proporciona el tipo de dato de una variable o de una columna. Se utiliza para declarar variables que almacenen valores de bases de datos.

Si tenemos una columna llamada user_id en la tabla users. Para declarar una variable con el mismo tipo de dato que el usado en nuestra tabla de usuarios, hacemos:

```
user_id users.user_id% TYPE;
```

Al usar %TYPE nos despreocupamos de los cambios futuros en la definición de la tabla.

Mientras que con nombre tabla%ROWTYPE declaramos una fila con la estructura de la tabla especificada. La cual puede ser una tabla o una vista existente en la base de datos. Los campos de la fila se acceden con la notación punto.

Los parámetros de una función pueden ser de tipo compuesto (filas de una tabla). Es este caso, el identificador correspondiente $n es del tipo ROWTYPE, pero debe usarse el comando ALIAS.

Solamente los atributos del usuario de la tabla son accesibles en la fila, ni los OID ni otros atributos del sistema (debido a que la fila puede ser de una vista). Los campos de un ROWTYPE heredan el tamaño de los campos o la precisión de los tipos de dato char(), etc.

```
DECLARE
users_rec users% ROWTYPE;
user_id users% TYPE;
BEGIN
user_id := users_rec.user_id;
...
CREATE FUNCTION cs_refresh_one_mv(integer) RETURNS INTEGER 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
  RAISES EXCEPTION ''View '' || key || '' not found'';
RETURN 0;
END IF;
-- La columna mv_name de cs_materialized_views almacena
-- los nombres de las vistas.
TRUNCATE TABLE table_data.mv_name;
INSERT INTO table_data.mv_name || '' '' || table_data.mv_query;
return 1;
END;
' LANGUAGE 'plpgsql';
```

3.5. Los operadores

PostgreSQL proporciona un gran número de tipos de operadores. Éstos, están declarados en el catálogo del sistema pg_operator. Cada entrada en pg_operator incluye el nombre del

procedimiento que implementa el operador y las clases OID de los tipos de entrada y salida.

3.5.1. Operadores Generales

Tabla 3.1: Operadores generales.

OPERADOR	DESCRIPCIÓN	UTILIZACIÓN
<	Menor que?	1 <2
<=	Menor o igual que?	1 <= 2
<>	No igual?	1 <>2
=	Igual?	1 = 1
>	Mayor que?	2 >1
>=	Mayor o igual que?	2 >= 1
>||	Concatena strings	'Postgre'|| 'SQL'
>!!=	NOT IN	3 !!= i

3.5.2. Operadores Numéricos

Tabla 3.2: Operadores númericos

OPERADOR	DESCRIPCIÓN	UTILIZACIÓN
!	Factorial	3!
!!	Factorial (operador izquierdo)	!!3
%	Módulo	5 %4
%	Truncado	%4.5
*	Multiplicación	2*3
+	Suma	2+3
-	Resta	2-3
/	División	4/2
:	Exponencial Natural	:3.0
;	Logaritmo Natural	(;5.0)
	Valor Absoluto	-5.0
^	Exponencial	2.0^3.0
|/	Raíz Cuadrada	|/25.0
||/	Raíz Cúbica	||/27.0

3.5.3. Operadores de Intervalos de Tiempo

Tabla 3.3: Operadores de tiempo

OPERADOR	DESCRIPCIÓN
#<	Intervalo menor que?
#<=	Intervalo menor o igual que?
#<>	Intervalo no igual que?
#=	Intervalo igual que?
#>	Intervalo mayor que?
#>=	Intervalo mayor o igual que?
<#>	Convertir a un intervalo de = tiempo
«	Intervalo menor que?
|	Comienzo de intervalo
≃	Parecido a
<?>	Tiempo dentro del intervalo?

3.5.4. Operadores IP V4 CIDR

Tabla 3.4: Operadores IP V4 CIDR

OPERADOR	DESCRIPCIÓN	UTILIZACIÓN
<	Menor que	'192.168.1.5'::cidr <'192.168.1.6'::cidr
<=	Menor o igual que	'192.168.1.5'::cidr <= '192.168.1.5'::cidr
=	Igual que	'192.168.1.5'::cidr = '192.168.1.5'::cidr
>=	Mayor o igual que	'192.168.1.5'::cidr >= '192.168.1.5'::cidr
>	Mayor que	'192.168.1.5'::cidr >'192.168.1.4'::cidr
<>	No igual que	'192.168.1.5'::cidr <>'192.168.1.4'::cidr
«	Está contenido en	'192.168.1.5'::cidr «'192.168.1/24'::cidr
«=	Está contenido en o es igual a	'192.168.1/24'::cidr «= '192.168.1/24'::cidr
»	Contiene	'192.168.1/24'::cidr »'192.168.1.5'::cidr
»=	Contiene o es igual que	'192.168.1/24'::cidr »= '192.168.1/24'::cidr

3.5.5. Operadores IP V4 INET

Tabla 3.5: Operadores IP V4 INET

OPERADOR	DESCRIPCIÓN	UTILIZACIÓN
<	Menor que	'192.168.1.5'::inet <'192.168.1.6'::inet
<=	Menor o igual que	'192.168.1.5'::inet <= '192.168.1.5'::inet
=	Igual que	'192.168.1.5'::inet = '192.168.1.5'::inet
>=	Mayor o igual que	'192.168.1.5'::inet >= '192.168.1.5'::inet
>	Mayor que	'192.168.1.5'::inet >'192.168.1.4'::inet
<>	No igual	'192.168.1.5'::inet <>'192.168.1.4'::inet
«	Está contenido en	'192.168.1.5'::inet «'192.168.1/24'::inet
«=	Esté contenido o es igual a	'192.168.1/24'::inet «= '192.168.1/24'::inet
»	Contiene	'192.168.1/24'::inet »'192.168.1.5'::inet
»=	Contiene o es igual a	'192.168.1/24'::inet »= '192.168.1/24'::inet

3.5.6. Operador de asignación

Tabla 3.6: Operador de asignación

OPERADOR	DESCRIPCIÓN
:= (dos puntos + igual)	

3.5.7. Operadores relacionales o de comparación

OPERADOR	DESCRIPCIÓN
=	(igual a)
<>	(distinto de)
<	(menor que)
>	(mayor que)
>=	(mayor o igual a)
<=	(menor o igual a)

3.5.8. Operadores lógicos

Tabla 3.7: Operadores lógicos

OPERADOR	DESCRIPCIÓN
AND	(y lógico)
NOT	(negación)
OR	(o lógico)
Operador de concatenación	||

3.6. Las estructuras de control

Las estructuras de control son una parte útil, e importante, de PL/pgSQL. Con estas estructuras de control, gestionamos datos de PostgreSQL de forma flexible y potente.

3.6.1. Condiciones

```
1 IF expresión THEN
2   sentencias
3 [ELSE
4   sentencias]
5 END IF;
```

Donde expresión devuelve un valor que, al menos, pueda ser un tipo booleano.

Las estructuras selectivas se utilizan para tomar decisiones lógicas. Existen en cuatro "formas" para el IF en PL/pgSQL :

```
1 IF ... THEN
2 IF ... THEN ... ELSE
3 IF ... THEN ... ELSE IF and
4 IF ... THEN ... ELSIF ... THEN ... ELSE
```

3.6.1.1. IF-THEN

Las sentencias IF-THEN son el formato más simple del IF:

```
1 IF expresión-booleana THEN
2   sentencias
3 END IF;
```

Las sentencias entre THEN y END IF se ejecutan si la condición se cumple. En caso contrario, son ignoradas.

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

3.6.1.2. IF-THEN-ELSE

Esta sentencias IF-THEN-ELSE gestiona dos acciones: para el caso verdad o para el caso falso:

```
1 IF expresión-booleana THEN
2   sentencias
3 ELSE
4   sentencias
5 END IF;
```

Las sentencias IF-THEN-ELSE añadidas al IF-THEN especifican unas sentencias alternativas que deben ser ejecutadas 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;
```

3.6.1.3. IF-THEN-ELSE IF

Las sentencias IF pueden anidarse, tal como en el siguiente caso:

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

Con éste formato, anidamos una sentencia IF dentro de la parte ELSE de una sentencia IF superior. Necesitamos una sentencia END IF por cada IF anidado, y uno para el IF-ELSE padre. Esto funciona, pero es tedioso cuando existen muchas alternativas a verificar.

3.6.1.4. IF-THEN-ELSIF-ELSE

IF-THEN-ELSIF-ELSE proporciona un método conveniente de verificar más de una alternativa por sentencia.

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

Formalmente es equivalente a comandos IF-THEN-ELSE-IF-THEN anidados, pero sólo necesitamos un END IF.

Veamos un código:

```
IF number = 0 THEN
  result := ''zero'';
ELSIF number > 0 THEN
  result := ''positive'';
ELSIF number < 0 THEN
  result := ''negative'';
ELSE
-- hmm, la única posibilidad es que el valor sea NULL
result := ''NULL'';
END IF;
```

La sección ELSE final es opcional.

3.6.2. CASE

La forma simple de CASE proporciona una ejecución condicional basada en la igualdad de operandos. La expresión de búsqueda es evaluada (una vez) y sucesivamente comparada a cada expresión en la cláusula WHEN. Si la coincidencia se cumple, entonces las sentencias correspondientes son ejecutadas, y el control pasa a la siguiente sentencias después del END CASE. Las expresiones subsecuentes a WHEN no son evaluadas. Si no hay coincidencia, la sentencia ELSE es ejecutada; pero si ELSE no está presente, entonces una excepción CASE_NOT_FOUND se produce.

Veamos el código siguiente:

```
CASE search-expression
WHEN expression [, expression [ ... ]] THEN
statements
[ WHEN expression [, expression [ ... ]] THEN
statements
... ]
[ ELSE
statements ]
END CASE;
```

Mientras que el código a continuación:

```
SELECT salary,
CASE WHEN department_id =90 THEN 'High Salary'
WHEN department_id =100 THEN '2nd grade salary'
ELSE 'Low Salary'
END
AS salary_status
FROM employees
LIMIT 15;
```

produce la salida:

```
 salary   |  salary_status
----------+------------------
24000.00 | High Salary
17000.00 | High Salary
17000.00 | High Salary
9000.00 | Low Salary
6000.00 | Low Salary
4800.00 | Low Salary
4800.00 | Low Salary
4200.00 | Low Salary
12000.00 | 2nd grade salary
9000.00 | 2nd grade salary
8200.00 | 2nd grade salary
7700.00 | 2nd grade salary
7800.00 | 2nd grade salary
6900.00 | 2nd grade salary
11000.00 | Low Salary
(15 rows)
```

En un caso más complejo, tenemos que:

```
CASE
WHEN boolean-expression THEN
statements
[ WHEN boolean-expression THEN
statements
... ]
[ ELSE
statements ]
END CASE;
```

La búsqueda en una sentencia CASE proporciona una ejecución condicional basada en la verdad de las expresiones booleanas. Cada expresión booleana en la cláusula WHEN es evaluada hasta que una produce un verdad. Entonces, las sentencias correspondientes son ejecutadas, y el control pasa a la siguiente sentencia después de END CASE. Las expresiones subsecuentes a WHEN no son evaluadas. Si un resultado verdad no se produce, la sentencia ELSE es ejecutada; pero si no está presente, entonces una excepción CASE_NOT_FOUND se produce.

Veamos el código:

```
CREATE OR REPLACE FUNCTION myfunc1 (x integer) RETURNS text AS $$
DECLARE
msg text;
BEGIN
CASE
WHEN x IN(2,4,6,8,10) THEN
msg := 'value even number';
WHEN x IN(3,5,7,9,11) THEN
```

```
msg := 'value is odd number';
END CASE;
RETURN msg;
END;
$$
LANGUAGE plpgsql
```

y aquí su salida:

```
postgres=# SELECT myfunc1(5);
myfunc1
----------------------
value is odd number
(1 row)
```

3.6.3. SWITCH

El equivalente SQL del SWITCH es el CASE.

3.6.4. Ciclos Simples

Con LOOP, EXIT, CONTINUE, WHILE, FOR y FOREACH organizamos nuestras funciones PL/pgSQL para repetir una serie de sentencias.

3.6.4.1. LOOP

LOOP define un ciclo incondicional que es repetido indefinidamente hasta que sea terminado por una sentencia EXIT o RETURN.

```
[<<etiqueta1>>]
LOOP
  sentencias
END LOOP;
```

El comando EXIT es usado en ciclos anidados para especificar en qué nivel de anidación debe terminar.

3.6.4.2. EXIT

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

Si no se proporciona etiqueta, el ciclo más interno es terminado y la sentencia debajo a END LOOP es ejecutada. Si se proporciona etiqueta, ésta identifica el ciclo o bloque actual o de nivel interno. Cuando el ciclo o bloque nombrado es terminado, el control continúa con la sentencia siguiente al correspondiente END del ciclo/bloque.

Si WHEN está presente, la salida del ciclo sólo ocurre si la condición especificada es cierta, de lo contrario pasa a la sentencia después de EXIT.

Códigos:

```
LOOP
-- algunas sentencias
  IF count > 0 THEN
    EXIT; -- exit loop
  END IF;
END LOOP;
```

y

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

3.6.4.3. CONTINUE

Veamos el siguiente comando:

```
CONTINUE [ label ] [ WHEN boolean-expression ];
```

Si el comando no existe, la siguiente iteración del ciclo más interno empieza. Es decir, todas la sentencias restantes en el cuerpo del ciclo son omitidas, y el control regresa a la expresión de control del ciclo para determinar si es necesario ejecutar otra iteración o ciclo. Si la etiqueta está presente especifica el ciclo cuya ejecución será continuado.

Si WHEN existe, la iteración siguiente en el ciclo empezará si, y sólo si, la expresión booleana es verdad. De lo contrario, el control pasa a la sentencia siguiente después de CONTINUE.

CONTINUE puede ser utilizado con todos los tipos de ciclo; no está limitado por ciclos incondicionales.

Código:

```
LOOP
-- some computations
EXIT WHEN count > 100;
CONTINUE WHEN count < 50;
-- some computations for count IN [50 .. 100]
END LOOP;
```

3.6.4.4. WHILE

El comando WHILE repite una secuencia de sentencias mientras que la condición se cumpla.

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

La expresión es verificada justo a la entrada al cuerpo del ciclo.

Código:

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
-- algunas sentencias aquí
END LOOP;
WHILE NOT boolean_expresión LOOP
-- algunas sentencias aquí
END LOOP;
```

3.6.4.5. FOR

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

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

Algunos ejemplos:

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

```
FOR i IN REVERSE 10..1 LOOP
-- algunas expresiones aquí
END LOOP;
```

3.6.4.6. Tópico especial

El siguiente código muestra los comandos aprendidos y su utilización:

```
-CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
+CREATE FUNCTION refresh_mviews() RETURNS integer AS $$
```

```
3  DECLARE
4  mviews RECORD;
5  BEGIN
6  -    RAISE NOTICE 'Refreshing materialized views...';
7  -
8  -    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY
       sort_key LOOP
9  +    RAISE NOTICE 'Refreshing all materialized views...';
10 +
11 +    FOR mviews IN
12 +       SELECT n.nspname AS mv_schema,
13 +              c.relname AS mv_name,
14 +              pg_catalog.pg_get_userbyid(c.relowner) AS owner
15 +         FROM pg_catalog.pg_class c
16 +    LEFT JOIN pg_catalog.pg_namespace n ON (n.oid = c.
       relnamespace)
17 +        WHERE c.relkind = 'm'
18 +     ORDER BY 1
19 +    LOOP
20
21 -        -- Now "mviews" has one record from cs_materialized_views
22 +        -- Now "mviews" has one record with information about the
        materialized view
23
24 -        RAISE NOTICE 'Refreshing materialized view %s ...',
       quote_ident(mviews.mv_name);
25 -        EXECUTE format('TRUNCATE TABLE %I', mviews.mv_name);
26 -        EXECUTE format('INSERT INTO %I %s', mviews.mv_name,
       mviews.mv_query);
27 +        RAISE NOTICE 'Refreshing materialized view %.% (owner: %)
       ...',
28 +                     quote_ident(mviews.mv_schema),
29 +                     quote_ident(mviews.mv_name),
30 +                     quote_ident(mviews.owner);
31 +        EXECUTE format('REFRESH MATERIALIZED VIEW %I.%I', mviews.
       mv_schema, mviews.mv_name);
32 END LOOP;
33
34 RAISE NOTICE 'Done refreshing materialized views.';
```

3.6.5. Ciclos a través de una búsqueda

3.6.5.1. FOR – IN

Veamos un tipo diferente de ciclo FOR, iteramos a través de los resultados de una consulta y manipulamos sus datos. La sintaxis es:

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

A la variable registro o fila le son asignadas todas las filas resultantes de la consulta SELECT y el cuerpo del ciclo es ejecutado para cada fila. Veamos un caso:

```
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';
```

Si el ciclo es terminado por una sentencia EXIT, el valor del último registro asignado es todavía accesible tras la salida del ciclo.

la sentencia FOR-IN-EXECUTE es otra forma de iterar sobre registros:

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

Esto es igual a el formato anterior, excepto porque la sentencia origen SELECT es especificada como una expresión de texto, la cual es evaluada y replanificada en cada entrada al FOR. Esto facilita al desarrollador seleccionar la velocidad de una consulta o la flexibilidad para una consulta dinámica, tal como con una sentencia EXECUTE.

> ***NOTA:*** *PL/pgSQL distingue los dos tipos de ciclos FOR (enteros o regresadores de registros) comprobando si la variable destino mencionada justo antes del FOR ha sido declarada como variable tipo registro/fila. Si no es así, asume que es un 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.*

```
[<<etiqueta>>]
FOR record | row IN select_clause LOOP
  sentencias
END LOOP;
```

Al registro o fila se asigna a todas los registros resultantes de la clausula de selección, y la sentencia se ejecuta para cada uno de ellos. Si el ciclo se termina con una sentencia EXIT, el último registro asignado es aún accesible después del ciclo.

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

Si no se incluye etiqueta, se termina el ciclo más interno, y se ejecuta la sentencia que sigue a END LOOP. Si se incluye debe ser la etiqueta del ciclo actual o de otro de mayor nivel. EL ciclo indicado se termina, y el control se pasa a la sentencia de después del END del ciclo o bloque correspondiente.

Tenemos como objetivo escribir una función que actualice los datos de 2 columnas de una tabla si y sólo si hay datos para esa columna. Veamos el código propuesto que incluye algunos de los comandos vistos:

```
CREATE OR REPLACE FUNCTION p_update_locales_in_count() RETURNS
   INTEGER AS $BODY$
DECLARE
query_count          RECORD;
query_has_locales    RECORD;
real_data            RECORD;
BEGIN

FOR query_count IN  SELECT  co.neighborhood_code,co.city_code
FROM    conteo_manzanas_barrio_co co,sm_city ci
WHERE   co.city_code = ci.pk_city AND has_locales = TRUE
LIMIT 1 OFFSET 0 LOOP

SELECT  INTO real_data  sl.pk_locale,sl.name
FROM    sm_locale sl,sm_neighborhood sn,servcon_barrios sb
WHERE   sn.pk_neighborhood = query_count.neighborhood_code AND
sl.fk_pk_city = query_count.city_code AND
sb.cod_localidad = sl.locale_code AND
sb.cod_barrio = sn.neighborhood_code AND
sl.name = sb.nom_localidad;

UPDATE  conteo_manzanas_barrio_co
SET     locale_code = real_data.pk_locale, locale_name = real_data
   .name
WHERE   neighborhood_code = query_count.neighborhood_code;

END LOOP;

RETURN 0;
```

```
28 END $BODY$
29 LANGUAGE plpgsql;
```

3.6.6. Ciclos a través de arreglos

3.6.6.1. FOREACH

El ciclo FOREACH es parecido al ciclo FOR, pero en lugar de iterar por las filas devueltas por una consulta SQL, lo hace a través de los elementos de un arreglo. En general, el ciclo FOREACH recorre los componentes de una expresión compuesta. La sentencia FOREACH para un arreglo es:

```
1 [ label ]
2 FOREACH target [ SLICE number ] IN ARRAY expression LOOP
3 statements
4 END LOOP [ label ];
```

Si la opción SLICE[4]*, no existe o si está presente, el ciclo itera a través de elementos individuales de un arreglo producto de la evaluación de la expresión. La variable destino es asignada a cada elemento en la secuencia, y el cuerpo del ciclo es ejecutado para dicho elemento.*

Veamos el siguiente código:

```
1  CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
2  DECLARE
3  s int8 := 0;
4  x int;
5  BEGIN
6  FOREACH x IN ARRAY $1
7  LOOP
8  s := s + x;
9  END LOOP;
10 RETURN s;
11 END;
12 $$ LANGUAGE plpgsql;
```

Los elementos son recorridos en el orden de almacenamiento, sin considerar su número de dimensiones. Aunque el destino es generalmente sólo una sola variable, puede ser una lista de variables cuando el ciclo recorre un arreglo formado por valores compuestos (RECORD). En esos casos, para cada elemento del arreglo, las variables son asignadas de las columnas sucesivas del valor compuesto.

Con valores positivos de SLICE, el ciclo FOREACH itera a través de capas del arreglo más que por elementos individuales. El valor de SLICE debe ser una constantes entera o mayor que el número de dimensiones del arreglo. La variable destino debe ser un arreglo, y recibe partes sucesivas de él, cada parte es del número de dimensiones especificadas por el SLICE.

[4] Entendemos parte

Veamos:

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
DECLARE
x int[];
BEGIN
FOREACH x SLICE 1 IN ARRAY $1
LOOP
RAISE NOTICE "row = %", x;
END LOOP;
END;
$$ LANGUAGE plpgsql;
```

veamos:

```
SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);
```

NOTICE: row = 1,2,3 NOTICE: row = 4,5,6 NOTICE: row = 7,8,9 NOTICE: row = 10,11,12

3.6.7. Atrapando errores

Por defecto, cualquier error ocurrido en una función PL/pgSQL interrumpe su ejecución, y es concerniente a las transacciones. Podemos atrapar los errores y recuperarnos de ellos usando un bloque BEGIN con una cláusula EXCEPTION. La sintaxis es:

```
[ label ]
[ DECLARE
declarations ]
BEGIN
statements
EXCEPTION
WHEN condition [ OR condition ... ] THEN
handler_statements
[ WHEN condition [ OR condition ... ] THEN
handler_statements
... ]
END;
```

Si no ocurre un error, el bloque ejecuta todas las sentencias, y el control pasa a la siguiente sentencia después de END. Pero si un error ocurre dentro del bloque, la ejecución se interrumpe, y el control pasa a la lista de EXCEPTION. En ella se busca la primera coincidencia del error ocurrido. Si existe, las sentencias handler_statements son ejecutadas, y el control pasa a la siguiente sentencia después de END. Si no hay coincidencias, el error avanza fuera de la cláusula de EXCEPTION: donde el error puede ser atrapado por un bloque que contiene la EXCEPTION, pero si no es encontrada una coincidencia se interrumpe la ejecución de la función.

Los nombres de las condiciones pueden ser cualquiera de los mostrados en el Apéndice C.

OTHERS es el nombre de condiciones especiales que coinciden para cada tipo de error, excepto cuando usuamos QUERY_CANCELED. Es posible, pero frecuentemente desaconsejable, atrapar a QUERY_CANCELED por nombre. Los nombres de condición no distinguen entre minúsculas y/o mayúsculas. También, podemos especificar una condición de error por código SQLSTATE; veamos un código:

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

Si un nuevo error ocurre dentro de handler_statements, no puede ser atrapado por esta cláusula EXCEPTION, e ir fuera del bloque. Podemos enmarcarlo para atraparlo.

Cuando un error es atrapado por una cláusula EXCEPTION, las variables locales de la función PL/pgSQL permanecen con los mismos valores a cuando el error ocurrió, pero los cambios realizados en el estado de las bases de datos persistentes son recuperados ROLLED BACK. Consideremos el siguiente código:

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
x := x + 1;
y := x / 0;
EXCEPTION
WHEN division_by_zero THEN
RAISE NOTICE 'caught division_by_zero';
RETURN x;
END;
```

cuando el control asigna un valor a la variable y, fallará por ser una división entre cero, division_by_zero. El error será atrapado por la EXCEPTION. El valor devuelto en la sentencia RETURN es el valor incrementado de x, pero los efectos del comando UPDATE son recuperados ROLLED BACK. El comando INSERT anterior al bloque no es recuperado rolled back; sin embargo, al final el resultado en la base de datos contiene Tom Jones y no Joe Jones.

> ***NOTA:*** *Un bloque que con una cláusula EXCEPTION es significativamente más caro que entrar y salir de un bloque sin ella, no use EXCEPTION a menos que realmente la necesite.*

3.6.7.1. Excepciones con UPDATE/INSERT

El siguiente código gestiona la excepción en un UPDATE o INSERT:

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
LOOP
```

```
7  -- primero trata de actualizar según la clave (key)
8  UPDATE db SET b = data WHERE a = key;
9  IF FOUND THEN
10 RETURN;
11 END IF;
```

En el siguiente código. Cuando el registro no existe, trata de insertar la clave, si alguien más inserta la misma clave podemos tener una falla de clave única.

```
1  BEGIN
2  INSERT INTO db(a,b) VALUES (key, data);
3  RETURN;
4  EXCEPTION WHEN unique_violation THEN
5  -- No hacer nada, y el ciclo trata UPDATE nuevamente.
6  END;
7  END LOOP;
8  END;
9  $$
10 LANGUAGE plpgsql;
11
12 SELECT merge_db(1, 'david');
13 SELECT merge_db(1, 'dennis');
```

El código anterior supone que el comando INSERT causa el error de unique_violation, y no por, decirlo así, es un INSERT en una función de trigger sobre la tabla. También puede ser mal comportamiento si hay más de un índice único en la tabla, puesto que tratará la operación sin considerar de cuál de los índices causó el error. Para mayor seguridad podría estar usando las características siguientes presentadas a continuación para verificar que el error atrapado es el esperado.

3.6.8. Obteniendo información de los errores

Los gestores de excepciones con frecuencia necesitan identificar el error especifico que ocurrió. Hay dos formas de obtener información acerca de la excepción actual en PL/pgSQL: variables especiales y el comando GET STACKED DIAGNOSTICS.

Dentro de un gestor de excepción, la variable especial SQLSTATE contiene el código del error que corresponde a la excepción producida (Vea apéndice C, en la página 179. La variable especial SQLERRM contiene el mensaje de error asociado a la excepción. Estas variables son indefinidas fuera de los gestores de excepción.

Con el comando GET STACKED DIAGNOSTICS, podemos recuperar información de la excepción actual:

```
1 GET STACKED DIAGNOSTICS variable { = | := } item [ , ... ];
```

Cada ítem es una palabra clave que identifica un valor status a ser asignado a la variable especificada (la cual debería ser del tipo de dato correcto que recibe). Los estatus de los ítemes disponibles son mostrados en la Tabla 3.8.

Tabla 3.8: Diagnostico de ítemes de error

Nombre	Tipo	Descripción
RETURNED_SQLSTATE	text	SQLSTATE código de error de la excepción.
COLUMN_NAME	text	Nombre de la columna relacionada a la excepción.
CONSTRAINT_NAME	text	Nombre de la restricción relacionada a la excepción.
PG_DATATYPE_NAME	text	Nombre del tipo de dato relacionado a la excepción.
MESSAGE_TEXT	text	Mensaje de la excepción primaria.
TABLE_NAME	text	Nombre de la tabla relacionada a la excepción.
SCHEMA_NAME	text	Nombre del esquema relacionado a la excepción.
PG_EXCEPTION_DETAIL	text	Mensaje en detalle de la excepción, si procede.
PG_EXCEPTION_HINT	text	Consejo práctico sobre la excepción, si procede.
PG_EXCEPTION_CONTEXT	text	linea(s) de texto que describen la pila llamada al mismo tiempo que la excepción

Si la excepción no tiene un valor para un ítem, una cadena vacía es regresada.

Veamos el código a continuación:

```
1  DECLARE
2  text_var1 text;
3  text_var2 text;
4  text_var3 text;
5  BEGIN
6  -- some processing which might cause an exception
7  ...
8  EXCEPTION WHEN OTHERS THEN
9  GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
10 text_var2 = PG_EXCEPTION_DETAIL,
11 text_var3 = PG_EXCEPTION_HINT;
12 END;
```

3.6.9. Obteniendo información de la ubicación de ejecución

El comando GET DIAGNOSTICS previamente descrito, recupera información acerca del estado de la ejecución actual, donde el comando GET STACKED DIAGNOSTICS reporta la información acerca del estado de ejecución del error previo. Su ítem de estatus

PG_CONTEXT es útil para identificar la ubicación de ejecución actual. PG_CONTEXT regresa una cadena de texto con las lineas que describen la pila llamada. La primera linea se refiere a la función y comando GET DIAGNOSTICS de ejecución actuales. Las siguientes lineas se refieren a las funciones invocadas más que a la pila de llamadas. Veamos el código:

```
CREATE OR REPLACE FUNCTION outer_func() RETURNS integer AS $$
BEGIN
RETURN inner_func();
END;
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION inner_func() RETURNS integer AS $$
DECLARE
stack text;
BEGIN
GET DIAGNOSTICS stack = PG_CONTEXT;
RAISE NOTICE E'--- Call Stack ---\n%', stack;
RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT outer_func();

NOTICE:  --- Call Stack ---
PL/pgSQL function inner_func() line 5 at GET DIAGNOSTICS
PL/pgSQL function outer_func() line 3 at RETURN
CONTEXT:  PL/pgSQL function outer_func() line 3 at RETURN
outer_func
------------
1
(1 row)
```

`GET STACKED DIAGNOSTICS ... PG_EXCEPTION_CONTEXT` *devuelve el mismo tipo de traza de la pila, pero describiendo la ubicación donde el error fue detectado, más que la ubicación real.*

3.7. Los bloques

Un programa PL/pgSQL se estructura usando distintos bloques que agrupan declaraciones y sentencias relacionadas. Cada bloque del programa cumple una tarea específica y resuelve un problema en particular. Tal organización, el programa Pl/pgSQL es fácil de comprender y mantener. Las palabras clave y los identificadores pueden escribirse mezclando letras mayúsculas y minúsculas.

Un bloque se define de la siguiente manera:

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

```
3   declaraciones]
4 BEGIN
5   sentencias
6 EXCEPTION
7   gestor de excepción (opcional)
8 END;
```

El cuerpo del programa empieza con BEGIN y termina con EXCEPTION que inicia la sesión de manejo de excepciones del bloque; si el bloque no incluye algún controlador de excepciones, el cuerpo del programa termina con END.

Pueden existir varios bloques anidados en la sección de sentencias de un bloque. El anidamiento pueden ser usado para ocultar las variables a los bloques más externos.

Normalmente una de las sentencias es el valor de retorno, usando la palabra clave RETURN.

Las variables declaradas en la sección que antecede a un bloque se inicializan a su valor por omisión cada vez que se entra al bloque, no solamente al ser invocada la función.

La estructura básica de un programa PL/pgSQL es el bloque, que incluye dos partes, la declaración de variables y la sección de sentencias.

```
1 DECLARE
2   sección de variables
3 BEGIN
4   sección de sentencias
5 END;
```

Tabla 3.9: Comandos para los bloques PL/pgSQL

Sentencia	Descripción
DECLARE BEGIN END	Bloque
:=	Asignación
SELECT INTO	Asignación desde un select
Sentencias sql	Cualquier sentencia sql
PERFORM	Realiza una llamada a comando sql
EXECUTE	Interpreta una cadena como comando sql
EXIT	Termina la ejecución de un bloque
RETURN	Termina la ejecución de una función
IF	Ejecuta sentencias condicionalmente
LOOP	Repite la ejecución de un conjunto de sentencias
WHILE	Repite un conjunto de sentencias mientras
FOR	Repite un conjunto de sentencias utilizando una variable de control
RAISES	Despliega un mensaje de error a advertencia

No confundir el uso de las sentencias de agrupamiento BEGIN/END de PL/pgSQL con los

comandos de la base de datos que sirven para el control de las transacciones. Las funciones y procedimientos disparadores no pueden iniciar o realizar transacciones y PostgreSQL no soporta transacciones anidadas.

Se debe tener en cuenta que el uso de BEGIN y END para agrupar sentencias en PL/pgSQL no es el mismo que al iniciar o terminar una transacción. Las funciones y los disparadores son ejecutados siempre dentro de una transacción establecida por una consulta externa.

Esta estructura en forma de bloque la observamos en la función en PL/pgSQL del siguiente ejemplo.

Función que retorna la suma 2 de números enteros.

```
1  CREATE OR REPLACE FUNCTION sumar(int, int) RETURNS int AS
2  $$
3  BEGIN
4  RETURN $1 + $2;
5  END;
6  $$ LANGUAGE plpgsql;
7  -- Invocación de la función
8  debian=# SELECT sumar(2, 3);
9  sumar
10 5
11 (1 fila)
```

Notamos que en este lenguaje procedural, al igual que en SQL, se mantienen las sentencias terminadas con punto y coma (;) al final de cada línea y para retornar el resultado se emplea la palabra reservada RETURN (para más detalles vea la sección 3.10, página 55).

Las etiquetas son necesarias cuando se desea identificar el bloque para ser usado en una sentencia EXIT o para calificar las variables declaradas en él; además, si son especificadas después del END deben coincidir con las del inicio del bloque.

Los bloques pueden estar anidados, por lo que aquel que aparezca dentro de otro debe terminar su END con punto y coma, no siendo requerido el del último END.

Cada sentencia en el bloque de sentencias puede ser un sub-bloque, para realizar agrupaciones lógicas o crear variables para un grupo de sentencias. En bloques externos, las variables pueden ser accedidas en un sub-bloque calificándolas con la etiqueta del bloque al que pertenecen. El código siguiente muestra el tratamiento de variables en bloques anidados y el acceso a variables externas mediante su calificación con el nombre del bloque al que pertenecen.

Empleo de variables en bloques anidados

```
1  CREATE FUNCTION incrementar_precio_porciento(id integer) RETURNS
       numeric AS
2  $$
3  <<principal>>
4  DECLARE
5  incremento numeric := (SELECT price FROM products WHERE prod_id =
       $1) *
```

```
0.3;
BEGIN
RAISES NOTICE "El precio después del incremento será de % Bs.S.",
    incremento;
-- Muestra el incremento en un 30%
<<excepcional>>
DECLARE
incremento numeric := (SELECT price FROM products WHERE prod_id
= $1) * 0.5;
BEGIN
RAISES NOTICE "El precio después del incremento excepcional será
    de %
Bs.S.", incremento; -- Muestra el incremento en un 50%
RAISES NOTICE "El precio después del incremento será de % Bs.S.",
principal.incremento; -- Muestra el incremento en un
30%
END;
RAISES NOTICE "El precio después del incremento será de % Bs.S.",
    incremento;
-- Muestra el incremento en un 30%
RETURN incremento;
END;
$$ LANGUAGE plpgsql;
```

–*Invocación de la función*

```
debian=# SELECT * FROM incrementar_precio_porciento(1);
NOTICE: El precio después del incremento será de 7.797 Bs.S.
NOTICE: El precio después del incremento excepcional será de
    12.995 Bs.S.
NOTICE: El precio después del incremento será de 7.797 Bs.S.
incrementar_precio_porciento
7.797
(1 fila)
```

3.8. Las 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 ciclo 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 dato 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. Sin la cláusula DEFAULT, entonces la variable es inicializada al valor NULL de SQL.

La opción CONSTANT previene la asignación 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 asigna a la variable la fecha y hora actual de la llamada, y no la fecha y hora de su compilación.

Veamos el siguiente código:

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

3.8.1. Alias para los parámetros de la función

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.

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

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';
```

3.8.2. Tipos Fila (Row)

Una variable de un tipo compuesto es denominada variable de fila (ROW-TYPE). 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.

```
nombre nombretabla%ROWTYPE;
```

En la actualidad, una variable de tipo fila sólo es declarada con 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 ... ;
```

```
7 RETURN in_t.f1 || use_t.f3 || in_t.f5 || use_t.f7;
8 END;
9 ' LANGUAGE 'plpgsql';
```

3.8.3. Records

Las variables de tipo registro (RECORD) son similares a las tipo fila, pero no tienen una estructura predefinida. La toman de la estructura actual de la fila que tienen asignada durante un comando SELECT o FOR. La subestructura de una variable tipo RECORD puede variar cada vez que se le asigne un valor.

```
1 nombre RECORD;
```

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 dato, sólo un contenedor.

3.8.4. Atributos

Usando los atributos %TYPE y %ROWTYPE, podemos declarar variables con el mismo tipo de dato o estructura que otro elemento de la base de datos (como un campo de una tabla).

```
1 variable%TYPE
```

%TYPE proporciona el tipo de dato de una variable o de una columna de base de datos. Podemos 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 dato que usuarios.user_id debemos escribir:

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

Usando %TYPE no necesita conocer el tipo de dato de la estructura referida, y lo más importante, si el tipo de dato del elemento referido cambia en el futuro (cambiamos la definición de tabla para user_id de INTEGER a REAL), no necesitará cambiar su definición de función.

```
1 tabla%ROWTYPE
```

%ROWTYPE proporciona el tipo de dato 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.

```
1 DECLARE
2 users_rec usuarios%ROWTYPE;
3 user_id usuarios.user_id%TYPE;
4 BEGIN
5 user_id
6 := 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';
```

3.8.5. RENAME

Usando la declaración RENAME cambiamos el nombre de una variable, registro o fila.

```
RENAME oldname TO newname;
```

Esto es útil si NEW o OLD debieran ser referidos por otro nombre dentro de un procedimiento trigger.

Código:

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

> ***NOTA:*** *RENAME parece que no funciona en PostgreSQL 7.3. ALIAS cubre la mayoría de los usos prácticos de RENAME.*

3.9. 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 (como 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 debe ser interpretada como un TIMESTAMP debido a que el campo destino de logtable es de ese tipo. Así, crea una constante a partir de él y su valor 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 desarrollador.

En el caso de logfunc2(), el intérprete principal de PostgreSQL no sabe a qué tipo debe 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 desarrollador.

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 dato que estaba presente cuando la expresión fue analizada por vez primera. Recuerde esto al escribir procedimientos trigger que manejen eventos para más de una tabla (EXECUTE puede ser usado para resolver este problema cuando sea necesario).

3.10. Retorno de valores

PL/pgSQL implementa 2 comandos que permiten devolver datos de una función: RETURN y RETURN NEXT|QUERY.

3.10.1. RETURN

La cláusula RETURN es empleada cuando la función no devuelve un conjunto de datos.

Tiene la forma:

```
1 RETURN expresión ;
```

Para su empleo debe tener en cuenta que esta cláusula:

- *Devuelve el valor de evaluar la expresión terminando la ejecución de la función.*
- *De haber definido una función con parámetros de salida no es necesario especificar ninguna expresión en ella.*
- *En funciones que regresen el tipo de dato VOID puede emplearse sin especificar ninguna expresión para terminar la función tempranamente.*
- *No debe dejar de especificarse en una función (excepto en los casos mencionados anteriormente), ya que genera un error en tiempo de ejecución.*
- *Lo que se devuelve debe tener el mismo tipo de dato que el declarado en la cláusula RETURNS en el encabezado de la función.*

El código siguiente muestra su empleo en el cuerpo de una función.

Utilización de la cláusula RETURN para devolver valores y culminar la ejecución de una función que dado el identificador de un producto devuelve su título, empleo de la cláusula RETURN para devolver un dato escalar

```
1 CREATE FUNCTION devolver_producto(integer) RETURNS varchar AS
2 $$
3 DECLARE
4 prod varchar;
5 BEGIN
6 SELECT title INTO prod FROM products WHERE prod_id = $1;
7 RETURN prod;
8 END;
9 $$ LANGUAGE plpgsql;
```

Veamos la misma función anterior pero empleando parámetros de salida, notamos que en este caso la cláusula RETURN no necesita una expresión asociada:

```
CREATE FUNCTION devolver_producto(integer, OUT varchar) RETURNS
   varchar AS $
BEGIN
SELECT title INTO $2 FROM products WHERE prod_id = $1;
RETURN;
END;
$$ LANGUAGE plpgsql;
```

Mientras que la siguiente función devuelve todos los datos del identificador de un producto, utilizando la cláusula RETURN para devolver un dato compuesto; en este código se debe garantizar que el resultado devuelva una sola tupla; en caso contrario, devuelve la primera del resultado

```
CREATE FUNCTION datos_producto(integer) RETURNS products AS
$$
DECLARE
prod products;
BEGIN
SELECT * INTO prod FROM products WHERE prod_id = $1;
RETURN prod;
END;
$$ LANGUAGE plpgsql;
```

Ahora, veamos una función que devuelve un dato compuesto dado el identificador de un producto y la utilización de la cláusula RETURN:

```
CREATE FUNCTION datos_producto_titulo(integer) RETURNS varchar AS
$$
DECLARE
prod RECORD;
BEGIN
SELECT * INTO prod FROM products WHERE prod_id = $1;
RETURN prod.title;
END;
$$ LANGUAGE plpgsql;
```

Esta misma función regresa un tipo de dato compuesto por el nombre y precio del producto:

```
CREATE TYPE mini_prod AS (nombre varchar, precio numeric);
CREATE FUNCTION datos_producto_mini_prod(integer) RETURNS
   mini_prod AS
$$
DECLARE
prod mini_prod;
BEGIN
SELECT * INTO prod FROM products WHERE prod_id = $1;
RETURN mini_prod;
END;
$$ LANGUAGE plpgsql;
```

3.10.2. RETURN NEXT o RETURN QUERY

Veamos la utilización de la cláusula RETURN NEXT|QUERY para devolver el conjunto de valores resultante de una consulta. Tiene la forma:

```
RETURN NEXT expresión ;
RETURN QUERY consulta ;
```

Para su empleo se debe tener en cuenta que estas cláusulas se cumplan cuando:

- *Son empleadas si la función es declarada para devolver algun_tipo con SETOF.*
- *Mientras que `RETURN NEXT` se emplea con tipos de datos compuestos, RECORD o fila.*
- *Mientras que `RETURN QUERY` añade el resultado de ejecutar una consulta al conjunto resultante de la función.*
- *Ambos pueden ser usados en una misma función, concatenándose ambos resultados. No culminan la ejecución de la función, simplemente añaden cero o más filas al resultado de la función, puede emplearse un RETURN sin argumento para salir de la función.*
- *De declararse parámetros de salida se puede especificar el RETURN NEXT sin una expresión, la función debe declararse para que devuelva SETOF record.*

Función que devuelve todos los productos registrados en una tabla de la base de datos cuando su identificador es mayor que el pasado por parámetro, notamos el empleo de un FOR para iterar por el resultado de la consulta.

```
CREATE FUNCTION datos_producto_setof(integer) RETURNS SETOF
   products AS
$$
DECLARE
resultado products;
BEGIN
FOR resultado IN SELECT * FROM products where prod_id > $1 LOOP
RETURN NEXT resultado;
END LOOP;
RETURN; -- Opcional
END;
$$ LANGUAGE plpgsql;
```

La misma función pero empleando RETURN QUERY

```
CREATE FUNCTION datos_producto_setof(integer) RETURNS SETOF
   products AS
$$
BEGIN
RETURN QUERY SELECT * FROM products where prod_id > $1;
RETURN; -- Opcional
END;
$$ LANGUAGE plpgsql;
```

La misma función pero empleando RETURN QUERY y devolviendo RECORD. Para ejecutarla, especificamos qué estructura debe tener el resultado de la forma:

```
SELECT * FROM datos_producto_record(1000) AS (prod_id int,
   category integer, title character varying(50), actor character
   varying(50), price numeric(12,2), special smallint,
   common_prod_id integer)
```

```
CREATE FUNCTION datos_producto_record(integer) RETURNS SETOF
   record AS
$$
BEGIN
RETURN QUERY SELECT * FROM products where prod_id > $1;
RETURN; -- Opcional
END;
$$ LANGUAGE plpgsql;
```

3.11. Las excepciones

PostgreSQL no dispone de un modelo de manejo de excepciones muy elaborado.

Cuando el analizador, optimizador o ejecutor deciden que una sentencia no puede ser procesada, la transacción completa es interrumpida y el sistema vuelve al ciclo principal para procesar la siguiente consulta de la aplicación cliente.

Es necesario comprender el mecanismo de errores para detectar cuando sucede esto; lo que no es posible es saber qué ha causado en realidad la interrupción (un error de conversión de entrada/salida, un error de punto flotante, un error de análisis).

Es posible que la base de datos haya quedado en un estado inconsistente, por lo que volver a un nivel de ejecución superior o continuar ejecutando comandos puede corromper toda la base de datos. E incluso aunque se pudiera enviar la información a la aplicación cliente, la transacción se habrá interrumpido, por lo que carece de sentido el intentar reanudar la operación.

Por todo esto, lo único que hace PL/pgSQL cuando se produce una excepción durante la ejecución de una función o procedimiento disparador es enviar mensajes de depuración al nivel DEBUG, indicando en qué función, número de línea y tipo de sentencia ha ocurrido el error.

Veamos una función, con una consulta pasada por parámetro, valida si se puede ejecutar en la base de datos:

```
CREATE OR REPLACE FUNCTION valida_consulta(consulta varchar)
RETURNS void AS
$$
BEGIN
EXECUTE $1;
```

```
6  EXCEPTION
7  WHEN syntax_error THEN
8  RAISES EXCEPTION "Consulta con problemas de sintaxis";
9  WHEN undefined_column OR undefined_table THEN
10 RAISES EXCEPTION "Columna o tabla no válida";
11 END;
12 $$ LANGUAGE plpgsql;
```

Invocación de la función valida_consulta:

```
1  debian=# SELECT * FROM valida_consulta('select * from catego');
2  ERROR: Columna o tabla no válida
```

Los códigos analizados previamente han generado mensajes utilizando la cláusula RAISES con la opción NOTICE.

La sentencia RAISES envía mensajes de tres niveles de severidad:

1. *DEBUG. El mensaje se escribe en la bitácora del sistema (logs).*
2. *NOTICE. El mensaje se escribe en la bitácora y en el cliente psql.*
3. *EXCEPTION. El mensaje se escribe en la bitácora y aborta la transacción.*

El mensaje puede incluir valores de variables mediante el carácter "%":

1. *RAISES debug funcion(): ejecutada con éxito;*
2. *RAISES notice "El valor % se tomo por omisión", variable;*
3. *RAISES excepción "El valor % está fuera del rango permitido", variable;*

Además, la cláusula RAISES permite las opciones DEBUG, LOG, INFO, WARNING y EXCEPTION, ésta última utilizada por defecto:

- *NOTICE: es utilizado para hacer notificaciones.*
- *WARNING: es utilizado para hacer advertencias.*
- *LOG: es utilizado para dejar constancia en los logs de PostgreSQL del mensaje o error.*
- *EXCEPTION: es utilizado para lanzar una excepción, cancelándose todas las operaciones realizadas previamente en la función.*

El código siguiente muestra casos donde es utilizado RAISES con varios de los niveles de mensajes.

3.11.1. Mensajes usando la opción RAISES: EXCEPTION, LOG y WARNING

Veamos una función que devuelve el título de un producto, dado un identificador de producto, utilizando la cláusula RETURN para devolver un dato escalar, y haciendo uso de FOUND y EXCEPTION para determinar si lo encontró:

```
1  CREATE FUNCTION producto(integer) RETURNS varchar AS
2  $$
3  DECLARE
4  prod varchar;
5  BEGIN
6  SELECT title INTO prod FROM products WHERE prod_id = $1;
7  IF not FOUND THEN
8  RAISES EXCEPTION "Producto % no encontrado", $1;
9  END IF;
10 RETURN prod;
11 END;
12 $$ LANGUAGE plpgsql;
```

Invocación de la función:

```
1  debian=# SELECT producto(1000000);
2  ERROR: Producto 1000000 no encontrado
```

Utilización de la cláusula RETURN para devolver un dato escalar, uso de FOUND, LOG y WARNING para determinar si lo encontró y/o registrar el error en el log de PostgreSQL.

```
1  CREATE OR REPLACE FUNCTION producto(integer) RETURNS varchar AS
2  $$
3  DECLARE
4  prod varchar;
5  BEGIN
6  SELECT title INTO prod FROM products WHERE prod_id = $1;
7  IF not FOUND THEN
8  RAISES LOG "Producto % no encontrado", $1;
9  RAISES WARNING "Producto % no encontrado, es posible que no se
       haya
10 insertado aún", $1;
11 END IF;
12 RETURN prod;
13 END;
14 $$ LANGUAGE plpgsql;
```

Invocación de la función:

```
1  debian=# SELECT producto(1000000);
2  WARNING: Producto 1000000 no encontrado, es posible que no se haya
       insertado aún producto
3  (1 fila)
```

Notemos que se ha utilizado la variable especial FOUND, que es de tipo booleano, que por defecto en PL/pgSQL es FALSE y se activa luego de:

- *Una asignación de un SELECT INTO que haya generado un resultado efectivo almacenado en la variable.*
- *Las operaciones UPDATE, INSERT y DELETE que realizaron alguna acción efectiva sobre la base de datos.*
- *Las operaciones RETURN QUERY y RETURN QUERY EXECUTE que devuelvan algún valor.*

Existen otros casos donde se activa la variable FOUND, pero esto sale el alcance de este libro.

3.11.2. Excepciones personalizadas

También puede utilizarse un bloque de excepciones para el tratamiento de las mismas como se muestra a continuación. Creando la excepción personalizada "P2222":

```
1 CREATE OR REPLACE FUNCTION s164() returns void as
2 $$
3 BEGIN
4 RAISES exception using message = "S 164", detail = "D 164", hint =
      "H 164", errcode = "P2222";
5 END;
6 $$ language plpgsql;
```

Creando una excepción personalizada que no asigna errm:

```
1 CREATE OR REPLACE FUNCTION s165() returns void as
2 $$
3 BEGIN
4 RAISES exception "%",'nada especificado";
5 END;
6 $$ language plpgsql;
```

Llamando a:

```
1 t=# do
2 $$
3 DECLARE
4 _t text;
5 BEGIN
6 PERFORM s165();
7 EXCEPTION WHEN SQLSTATE "P0001" then RAISES info "%","estado P0001
      atrapado: "||SQLERRM;
8 PERFORM s164();
9 END;
```

```
$$
;
INFO:
STATE P0001 CAUGTH: nada especificado
ERROR: S 164
DETAIL: D 164
HINT: H 164
CONTEXT: SQL STATEMENT "SELECT s164()"
PL/pgSQL function inline_code_block line 7 at PERFORM
```

Aquí se procesa el P0001 personalizado y P2222, no interrumpiendo la ejecución.

4

Interacción con la base de datos

El objetivo principal de usar PL/pgSQL es crear programar para acceder a la base de datos PostgreSQL y ejecutar sentencias SQL. Un programa PL/pgSQL manipula información de una base de datos a través de sentencias y cursores estándar DML[1].

Los programas PL/pgSQL pueden incluir comandos válidos como SELECT, INSERT, UPDATE o DELETE para manipular filas en una tabla de una base de datos PostgreSQL.

El siguiente código de un bloque Pl/pgSQL, anónimo, inserta un nuevo registro en la tabla partes.

```
1 DECLARE
2 newId INTEGER:=6;
3 newDesc VARCHAR(250):="teclado";
4 BEGIN
5   INSERT INTO partes VALUES (newId, new Desc);
6 END;
```

Una variable o constante en un bloque PL/pgSQL debe satisfacer los requisitos para una expresión en un comando DML. El código anterior usa variables locales para proporcionar los dos primeros valores en la cláusula VALUES de la sentencia INSERT.

Si consultamos la tabla partes observamos el nuevo registro.

```
1 select * from partes;
```

Las modificaciones de datos realizadas por los comandos INSERT, UPDATE y DELETE dentro de un bloque PL/pgSQL son parte de la transacción actual de la sesión. Aunque podemos incluir comandos COMMIT y ROLLBACK dentro de muchos tipos de bloques, el control de transacciones se realiza normalmente desde fuera de dichos bloques, de forma que los limites de la transacción son claramente visibles para aquellos que usan bloques PL/pgSQL.

[1]DML facilita buscar, añadir, suprimir y modificar datos de la base de datos. El DBMS proporciona una lenguaje de manipulación de datos (DML) completo. En los SGBDOO debe haber un lenguaje de programación de propósito general.

4.1. Asignar un valor a una variable con una consulta

Hemos visto la asignación de variables en el capítulo 3, página 21, ahora trabajamos con asignación de registros de las tablas de la base de datos.

- *Operador de asignación :=*
  - *variable[.campo] := EXPRESSION*
  - *Resultado de SELECT EXPRESSION*
- *SELECT EXPRESSION INTO variable;*
  - *Almacena el resultado de consultas complejas arbitrarias.*
  - *Soporta variables simples o registros/filas.*
  - *Puede ser utilizada en cursores.*

4.1.1. SELECT INTO

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

```
1 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.

Los programas PL/pgSQL usan la cláusula INTO del comando SELECT para asignar un valor específico de una tabla de una base de datos a una variable de programa. Intentemos este tipo de sentencia de asignación introduciendo el siguiente bloque PL/pgSQL, anónimo, que usa SELECT ... INTO para asignar un valor a una variable de programa.

```
1 DECLARE
2 partDesc VARCHAR(250);
3 BEGIN
4 SELECT descripcion INTO partDesc FROM partes WHERE Id = 6;
5 RETURN partDesc;
6 END;
```

La utilización de SELECT (sin INTO) ***NO*** *está permitido.*

Usos de los comandos INSERT, UPDATE y DELETE y de la variable FOUND:

- *los comandos INSERT / UPDATE / DELETE*
  - *REGRESAN/RETURNING EXPRESSION INTO variable;*
  - *Tiene la misma sintaxis que SELECT.*
- *La variable FOUND, Boolean, devuelve true si:*

- *ejecutamos SELECT, PERFORM, FETCH... si row(s) es producido.*
- *ejecutamos UPDATE, INSERT, DELETE... si row(s) es afectado.*
- *ejecutamos MOVE or FOR... si hace las cosas correctas.*
- *obtenemos DIAGNOSTICO/GET DIAGNOSTICS var = ROW_COUNT;*

Notemos que es diferente a la interpretación normal que hace PostgreSQL del SELECT INTO, donde el destino (target) INTO es una tabla recién creada, si requerimos crear una tabla a partir de un resultado de SELECT dentro de una función PL/pgSQL, usamos 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, automáticamente se configura al tipo fila de las columnas resultantes de la consulta.

Excepto por la cláusula INTO, el comando SELECT es igual que la consulta SELECT normal y puede usar todo su potencial.

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

Actualmente, la cláusula INTO puede aparecer en cualquier lugar en el comando SELECT, pero se recomienda ubicarla después de la palabra clave SELECT.

Podemos usar FOUND en un comando SELECT INTO para determinar si la asignación tuvo éxito (es decir, al menos fue devuelta una fila por el comando SELECT). Veamos un código simple, la búsqueda de un empleado en la tabla empname:

```
SELECT INTO myrec * FROM EMP WHERE empname = myname;
IF NOT FOUND THEN
  RAISE EXCEPTION "emplado % no encontrado", myname;
END IF;
```

Alternativamente, usamos el condicional IS NULL (o ISNULL) para probar si un resultado de RECORD/ROW es NULL. Consideremos que no hay forma de saber si han sido descartadas filas adicionales.

```
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;
```

4.2. Ejecutando una expresión o consulta sin resultado

Algunas veces requerimos evaluar una expresión o consulta pero descartar el resultado (porque llamamos a una función que tiene efectos útiles, pero un resultado inútil). Para hacer esto en PL/pgSQL, usamos el comando PERFORM:

```
1 PERFORM query;
```

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

> ***NOTA:*** *Uno espera que SELECT, sin cláusula INTO, acompañe al resultado pero la única forma aceptada es con PERFORM.*

Un ejemplo:

```
1 PERFORM create_mv("cs_session_page_requests_mv", my_query);
```

4.3. Ejecutando consultas dinámicas

Frecuentemente requerimos de consultas dinámicas dentro de nuestras funciones PL/pgSQL, es decir, ejecutar consultas que implican diferentes tablas o distintos tipos de datos. Los intentos normales de crear planes de consultas de PL/pgSQL no funcionarán en estos escenarios. Para manejar el problema, se proporciona el comando EXECUTE:

```
1 EXECUTE query-string;
```

donde query-string es una expresión de tipo text con la consulta a ser ejecutada. Esta cadena es leída literalmente por el motor SQL.

Consideremos, 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.

PostgreSQL necesita escapar las comillas simples dentro de la definición de función. Esto es importante recordarlo para el caso de crear funciones que generan otras funciones.

Cuando trabajamos con consultas dinámicas debe realizar el escape de comillas simples en PL/pgSQL. Vea la tabla 2.1, página 17 para una explicación detallada. Nos ahorrará esfuerzos.

Al contrario de otras consultas en PL/pgSQL, una consulta ejecutada por un comando EXECUTE no es preparada ni almacenada. Al contrario, la consulta es preparada cada vez que se ejecuta el comando. 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 es, actualmente, soportado dentro de EXECUTE. Así, la única forma de extraer un resultado

de un SELECT es usar, dinámicamente, el formato FOR-IN-EXECUTE.

Veamos el siguiente código:

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

que 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 pasadas a la función quote_literal().*

Ambas toman los pasos apropiados para devolver el texto introducido enmarcado entre comillas, simples o dobles, y con cualquiera carácter especial embebido, debidamente escapado.

Aquí tenemos un código de una consulta dinámica y el uso de EXECUTE:

```
CREATE FUNCTION cs_update_referrer_type_proc() RETURNS INTEGER AS
    '
DECLARE
referrer_keys RECORD; -- Declara registro generico para 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'''';'';
```

```
22 -- Esto funciona porque no estamos sustituyendo variables
23 -- De lo contrario, fallaría. Vea PERFORM para otra forma de
       ejecutar funciones
24 EXECUTE a_output; END; ' LANGUAGE 'plpgsql';
```

4.4. 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:

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

Este comando devuelve indicadores de estado del sistema. Cada elemento es una palabra clave, que identifica 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 consulta SQL. Notemos que RESULT_OID sólo es útil tras una consulta INSERT.

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

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

- *SELECT INTO establece FOUND a true si éste devuelve una fila, false si no se devuelve ninguna.*
- *PERFORM establece FOUND a true si produce una fila, false en caso contrario.*
- *UPDATE, INSERT, y DELETE establecen FOUND a true si al menos una fila es afectada, false en caso contrario.*
- *FETCH establece FOUND a true si devuelve una fila, false en caso contrario.*
- *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 comando FOR (ciclos enteros, de registro y de registros dinámicos). FOUND sólo se establece cuando el FOR termina; no es modificado por el comando FOR, aunque puede ser cambiado por la ejecución de otros comandos dentro del cuerpo del ciclo.*

FOUND es una variable local; cualesquier cambio afectará sólo a la función PL/pgSQL actual.

5

Tipo de registros

PL/pgSQL soporta, además, el tipo de dato fila (ROWTYPE), un tipo compuesto que almacena toda la fila de una consulta SELECT o FOR; donde sus campos son accesibles calificándolos, como cualquier otro tipo de dato compuesto (ver sección 5.1, página 69).

Para emplearlo, tenemos en cuenta que:

- *En esta estructura sólo son accesibles las columnas definidas por el usuario (no los OID u otras columnas del sistema).*
- *Los campos heredan el tamaño y precisión de los tipos de datos de los que son copiados.*

Una variable fila puede ser declarada para que tenga el mismo tipo de las filas de una tabla o vista mediante la forma:

```
1 nombre_tabla %ROWTYPE
```

Acción que también puede realizarse declarando la variable del tipo de la tabla de la que se quiere almacenar la estructura de sus filas:

```
1 variable nombre_tabla
```

Cada tabla tiene asociado un tipo de dato compuesto con su mismo nombre.

Códigos equivalentes empleando ambas formas son los siguientes:

```
1 cliente customers%ROWTYPE;
2 cliente customers;
```

5.1. Tipo de dato compuesto

Los tipos de datos definidos por el usuario son una de las funcionalidades que brinda PostgreSQL dentro de su capacidad de extensibilidad, que nos permite definir nuestros propios tipos de datos para un determinado resultado.

Esta funcionalidad tiene varias opciones de definición de tipos de datos personalizados, dentro

de los que se encuentran, entre otros, los enumerativos y los compuestos; los últimos son de gran utilidad sobre todo en ocasiones en que se necesita devolver de una función un resultado compuesto por elementos de varias tablas.

Dicho resultado consiste en la encapsulación de una lista de nombres, con sus tipos de datos, separados por coma. La sintaxis de definición es de la forma:

```
1 CREATE TYPE nombre AS ( [ nombre_atributo tipo_dato [,... ] ] ),
```

Un código de su empleo pudiera ser:

```
1 CREATE TYPE nombre_completo AS (nombre varchar, apellidos varchar)
    ;
```

5.2. Tipo de dato RECORD

PL/pgSQL soporta el tipo de dato RECORD, similar a ROWTYPE pero sin estructura predefinida, que toma de la fila actual asignada durante la ejecución del comando:

```
1 SELECT o FOR.
```

RECORD no es un tipo de dato verdadero sino un contenedor. Este tipo de dato no es el mismo concepto que cuando se declara una función para que devuelva un tipo RECORD; en ambos casos, la estructura de la fila actual es desconocida cuando la función está siendo escrita, pero para el retorno de una función la estructura es determinada cuando la llamada es revisada por el analizador sintáctico, mientras que la de la variable puede ser cambiada en tiempo de ejecución.

Puede ser utilizado para devolver un valor del que no se conoce tipo de dato, pero sí se debe conocer su estructura cuando se quiere acceder a un valor dentro de él, por ejemplo para acceder a un valor de una variable de tipo RECORD se debe conocer previamente el nombre del atributo para poder calificarlo y acceder al mismo.

6

Funciones

6.1. Introducción

Este capítulo se una introducción a la programación de funciones en PL/pgSQL. El cual es un lenguaje influenciado por PL/SQL de Oracle.

La programación tiene, entre objetivos, la agrupación de consultas SQL y evitar la saturación del tráfico en la red informática entre el cliente y el servidor de bases de datos. Además, de un grupo de ventajas adicionales entre las que destacan: incluir estructuras iterativas y condicionales; heredar todos los tipos de datos, funciones y operadores definidos por el usuario; mejorar el rendimiento de cálculos complejos y emplearse para definir funciones disparadoras (triggers). Todo esto le otorga mayor potencialidad al combinar las ventajas de un lenguaje procedural y la facilidad de SQL.

PL/pgSQL soporta funciones que retornan "filas", donde la salida es un conjunto de valores que pueden ser tratados igual a una fila retornada por una consulta.

Las funciones pueden ser definidas para ejecutarse con los derechos del usuario ejecutor o con los de un usuario previamente definido. El concepto de funciones, en otros DBMS, son muchas veces referidas como "procedimientos almacenados"[1].

6.2. Estructura

Al ser PL/pgSQL un lenguaje estructurado por bloques, su definición debe ser un bloque de la forma:

```
1 [ << etiqueta >> ]
2 [DECLARE
3 Declaraciones ...]
4 BEGIN
```

[1] stored procedures

```
5  Sentencias ...
6  END [ etiqueta ];
```

Las funciones y los triggers son ejecutados por una transacción establecida por una consulta externa. Esta estructura, en forma de bloque, la vemos en la función en PL/pgSQL mostrada en el código siguiente.

```
1  -- Función que regresa la suma 2 de números enteros
2  CREATE OR REPLACE FUNCTION sumar(int, int) RETURNS int AS
3  $$
4  BEGIN
5  RETURN $1 + $2;
6  END;
7  $$ LANGUAGE plpgsql;
8  -- Invocación de la función
9  dell=# SELECT sumar(2, 3);
10 sumar
11 5
12 (1 fila)
```

Notemos que en este lenguaje procedural, al igual que en SQL, se mantienen las sentencias terminadas con punto y coma (;) al final de cada línea y para devolver el resultado usamos la palabra reservada RETURN (para más detalles vea la sección 6.7, página 77).

Las etiquetas son necesarias para identificar el bloque a ser usado en una sentencia EXIT o para calificar las variables declaradas en él; además, si son especificadas después del END deben coincidir con las del inicio del bloque.

La figura 6.1, página 72 muestra el uso de las etiquetas en un bloque:

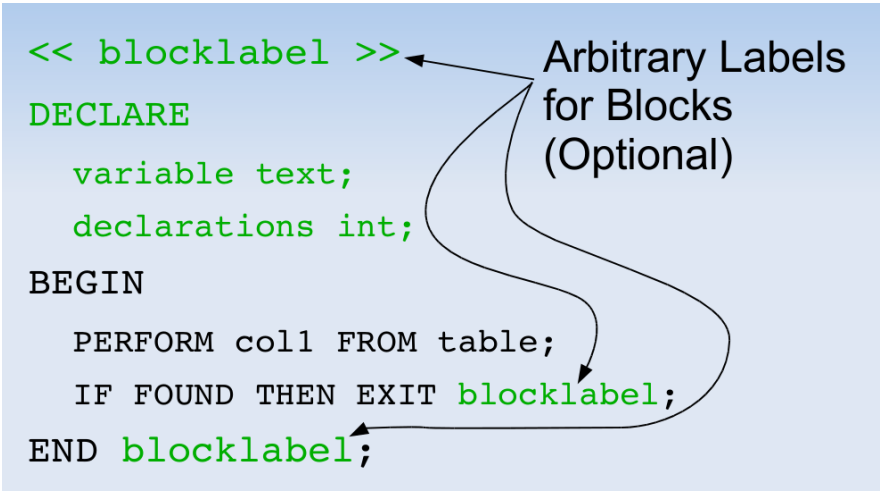


Figura 6.1: Usando las etiquetas en un bloque.

Los bloques pueden estar anidados, aquel que aparezca dentro de otro debe terminar su END con punto y coma, pero esto no siendo requerido en el último END.

Para recordar el uso del bloque vea Capítulo 3.7, página 46.

Veamos el siguiente código:

```
1  -- Empleo de variables en bloques anidados
2  CREATE FUNCTION incrementar_precio_porciento(id integer) RETURNS
       numeric AS
3  $$
4  <<principal>>
5  DECLARE
6  incremento numeric := (SELECT price FROM products WHERE prod_id =
       $1) *
7  0.3;
8  BEGIN
9  RAISE NOTICE 'El precio después del incremento será de % Bs.S.',
       incremento;
10 -- Muestra el incremento en un 30%
11 <<excepcional>>
12 DECLARE
13 incremento numeric := (SELECT price FROM products WHERE prod_id
14 = $1) * 0.5;
15 BEGIN
16 RAISE NOTICE 'El precio después del incremento excepcional será de
       %
17 Bs.S.', incremento; -- Muestra el incremento en un 50%
18 RAISE NOTICE 'El precio después del incremento será de % Bs.S.',
19 principal.incremento; -- Muestra el incremento en un
20 30%
21 END;
22 RAISE NOTICE 'El precio después del incremento será de % Bs.S.',
       incremento;
23 -- Muestra el incremento en un 30%
24 RETURN incremento;
25 END;
26 $$ LANGUAGE plpgsql;
27 --Invocación de la función
28 dell=# SELECT * FROM incrementar_precio_porciento(1);
29 NOTICE: El precio después del incremento será de 7.797 Bs.S.
30 NOTICE: El precio después del incremento excepcional será de
       12.995 Bs.S.
31 NOTICE: El precio después del incremento será de 7.797 Bs.S.
32 incrementar_precio_porciento
33 7.797
34 (1 fila)
```

6.3. Variables y constantes

Las variables pueden ser de cualquier tipo de dato SQL o definido por el usuario y deben ser declaradas en la sección DECLARE. Ver Capítulo 3.7, página 46.

La sintaxis general para la declaración de una variable es la siguiente:

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

6.4. Parámetros de funciones

En PL/pgSQL al igual que en SQL, vea la sección 6.4, página 74, se hace referencia a los parámetros de las funciones mediante la numeración $1, $2, $n o mediante un ALIAS, que puede crearse de 2 formas:

- *Nombrar el parámetro en CREATE FUNCTION (forma preferida).*
- *En la sección de declaraciones (única forma disponible previa a la versión 8.0 de PostgreSQL).*

El código a continuación muestra estas 2 formas de hacer referencia a los parámetros de las funciones en PL/pgSQL.

Creación de un ALIAS para el parámetro de la función duplicar_impuesto en el comando CREATE FUNCTION y en la sección DECLARE:

```
-- Función que define el alias de un parámetro en su definición
CREATE FUNCTION duplicar_impuesto(id integer) RETURNS numeric AS
$$
BEGIN
RETURN (SELECT tax FROM orders WHERE orderid = id) * 2;
END;
$$ LANGUAGE plpgsql;
-- Forma de especificar el alias de un parámetro en la sección
   DECLARE
CREATE FUNCTION duplicar_impuesto(integer) RETURNS numeric AS
$$
DECLARE
id ALIAS FOR $1;
BEGIN
RETURN (SELECT tax FROM orders WHERE orderid = id) * 2;
END;
$$ LANGUAGE plpgsql;
```

Es válido aclarar que el comando ALIAS no sólo se emplea para definir un parámetro, sino que puede ser utilizado para cualquier variable.

6.5. Consultas con resultados en una única fila

Para almacenar el resultado de un comando SQL que devuelve una fila se utiliza una variable de tipo RECORD, ROWTYPE o una lista de variables escalares, lo que puede hacerse añadiendo la cláusula INTO al comando (SELECT, INSERT, UPDATE o DELETE con cláusula RETURNING y comandos de utilidad que devuelven filas, como EXPLAIN), de la forma:

```
SELECT expresión INTO [STRICT] variable(s) FROM...;
INSERT ... RETURNING expresión INTO [STRICT] variable(s) ;
UPDATE ... RETURNING expresión INTO [STRICT] variable(s) ;
DELETE ... RETURNING expresión INTO [STRICT] variable(s) ;
```

Si tenemos más de una variable; éstas, deben estar separadas por coma. Si usamos una fila o lista de variables, las columnas resultantes de la consulta deben coincidir exactamente con la misma estructura de las variables y sus tipos de datos o se genera un error en tiempo de ejecución.

Si STRICT no es especificado, la variable almacena la primera fila devuelta por la consulta, también ocurre si usamos ORDER BY, o nulo si no hay resultado; siendo descartadas el resto de las filas. Si STRICT es especificado y si la consulta devuelve más de una fila se genera un error en tiempo de ejecución. No obstante, en el caso de INSERT, UPDATE o DELETE, aún cuando no sea especificado STRICT el error se genera si el resultado tiene más de una fila, y si no tiene la opción ORDER BY no se puede determinar cuál de las filas del resultado debería devolver.

El código siguiente muestra el uso de estos comandos para capturar una fila del resultado.

Captura de una fila resultado de consultas SELECT, INSERT, UPDATE o DELETE:

```
-- Guardar el resultado del SELECT en la variable fecha
SELECT orderdate INTO fecha FROM orders WHERE orderid = 1;
-- Guardar el resultado de firstname del SELECT en la variable
   nombre
SELECT firstname INTO nombre FROM customers ORDER BY firstname
   DESC;
-- Guardar el resultado del UPDATE en la variable pd de tipo
   record
UPDATE products SET title = 'Habana Eva' WHERE prod_id = 1
   RETURNING * INTO pd;
```

6.6. Comandos dinámicos

Existen escenarios donde es inevitable generar comandos dinámicos en las funciones PL/pgSQL, o sea, comandos que involucren diferentes tablas o tipos de datos cada vez que sean ejecutados. Para ello, utilizamos la sentencia EXECUTE con la sintaxis:

```
EXECUTE cadena [ INTO [ STRICT ] variable(s) ] [ USING expresión
    [,...] ];
```

Donde:

- *cadena: expresión de tipo texto que contiene el comando a ser ejecutado.*
- *variable(s): almacena(n) el resultado de la consulta, puede ser de tipo RECORD, fila o lista de variables simples separados por coma, con las especificaciones explicadas previamente para el empleo de la cláusula INTO (vea 6.5, página 75, que genera resultado con una única fila), de no ser especificada son descartadas las filas resultantes.*
- *expresión USING: suministra valores a ser insertados en el comando.*

Cada vez que el EXECUTE es ejecutado, la sentencia contenida en este comando es planeada, por lo que la cadena puede ser creada dinámicamente dentro de la función.

Este comando es especialmente útil cuando necesitamos usar valores de parámetros en la cadena a ser ejecutada que involucren tablas o tipos de datos dinámicos.

Para su empleo tenemos en cuenta los siguientes elementos:

- *Los símbolos de los parámetros ($1, $2, $n) pueden ser usados solamente para valores de datos, no para hacer referencia a tablas o columnas.*
- *Un EXECUTE con un comando CONST es equivalente a escribir la consulta directamente en PL/pgSQL, la diferencia radica en que EXECUTE replanifica el comando por cada ejecución generando un plan específico para los valores de los parámetros empleados, mientras que PL/pgSQL crea un plan genérico y lo reutiliza, se recomienda el uso de EXECUTE en situaciones donde el mejor plan dependa de los valores de los parámetros.*
- *La ejecución de consultas dinámicas requiere un manejo cuidadoso ya que pueden contener caracteres especiales, o acotados, que de no tratarse adecuadamente pueden generar errores en tiempo de ejecución. Para ello, se emplean las siguientes funciones:*
  - *quote_ident: empleada en expresiones que contienen identificadores de tablas o columnas.*
  - *quote_literal: empleada en expresiones que contienen cadenas literales.*
  - *quote_nullable: funciona igual que literal, pero es empleada cuando puede haber parámetros NULL, regresando una cadena NULL y no derivando en un error de EXECUTE al convertir todo el comando dinámico en NULL.*
- *Las consultas dinámicas pueden ser escritas, además, de forma segura, mediante la función FORMAT, que resulta ser una manera eficiente, ya que los parámetros no son convertidos a texto.*

Los siguientes códigos demuestran los elementos previamente analizados.

Utilización del comando EXECUTE en consultas constantes y dinámicas

```
1 -- Empleo del comando EXECUTE en consultas constantes
2 EXECUTE 'SELECT * FROM customers WHERE customerid = $1';
3 -- Consulta dinámica que recibe por parámetro el nombre de la
     tabla a eliminar
4 EXECUTE 'DROP TABLE IF EXISTS ' || $1 || ' CASCADE';
5 -- Consulta dinámica que actualiza un campo de la tabla products,
     recibiendo por
6 -- parámetros la columna a actualizar, el nuevo valor y su
     identificador
7 EXECUTE 'UPDATE products SET ' || quote_ident($1) || ' = ' ||
     quote_nullable($2) || '
8 WHERE prod_id = ' || quote_literal($3);
```

Notamos que para la ejecución de consultas dinámicas dejamos un espacio en blanco entre las cadenas de texto a unir de la consulta preparada; de no hacerse, se genera un error al convertir las cadenas en una sola, lo hace sin espacios entre los elementos concatenados.

6.7. Regresando datos desde una función

Hay dos comandos disponibles para regresar datos desde una función: RETURN y RETURN NEXT.

6.7.1. RETURN

En una función que regresa un tipo escalar, la expresión resultante es puesta en el tipo que regresa la función como es descrita en la asignación. El comando RETURN con una expresión termina la función y regresa el valor de la expresión al cliente. Esta forma es utilizada por funciones PL/pgSQL que regresan un sólo valor.

```
1 RETURN expression;
```

Para regresar un valor compuesto (fila), escribimos una expresión solicitante exactamente como lo requiere el conjunto de columnas. Esto puede requerir el uso de un patrón.

Si declaramos la función con parámetros de salida, escribimos RETURN con NO EXPRESIÓN. Los valores actuales del parámetro de salida son devueltos.

Si declaramos la función a devolver un NULL (VOID), una sentencia RETURN es usada para salir de la función; pero no escribimos una expresión después del RETURN.

El valor retorno de una función no puede ser indefinido. Si control alcanza el final del bloque de nivel superior de una función sin contactar un sentencia RETURN, un error en tiempo de ejecución ocurre. Esta restricción no se aplica a funciones con parámetros de salida y con aquellas que regresen valores nulos; sin embargo, en estos casos, una sentencia RETURN es

automáticamente ejecutada si el bloque de nivel superior termina.

Códigos que devuelven un tipo escalar:

```
RETURN 1 + 2;
RETURN scalar_var;
```

Códigos que devuelven un tipo compuesto:

```
RETURN composite_type_var;
RETURN (1, 2, 'three'::text);  -- must cast columns to correct
   types
```

6.7.2. RETURN NEXT y RETURN QUERY

En este caso, los ítemes a devolver son especificados en una secuencia de comandos RETURN NEXT o RETURN QUERY, y luego el comando RETURN final; sin argumentos, indica que la función ha terminado. Utilizamos RETURN NEXT combinando datos de tipo escalar y compuesto; con un resultado de tipo compuesto, una "tabla" entera de resultados es devuelta. RETURN QUERY regresa un conjunto de resultados. RETURN NEXT y RETURN QUERY pueden, libremente estar mezclados, en una función de un único resultado; en tal caso sus resultados son concatenados.

RETURN NEXT y RETURN QUERY no regresan un resultado de la función — simplemente agregan cero o más filas al conjunto resultante. La ejecución continua con la siguiente sentencia en la función PL/pgSQL. Como sucesivos comandos RETURN NEXT o RETURN QUERY son ejecutados, el conjunto resultado es formado. Un RETURN final, sin argumentos, pasa el control para salir de la función (o lo pasa para alcanzar el fin de la función).

RETURN QUERY tiene una variante en RETURN QUERY EXECUTE, que especifica la consulta a ejecutar es dinámica.

Veamos:

```
RETURN NEXT expression;
RETURN QUERY query;
RETURN QUERY EXECUTE command-string [ USING expression [, ... ] ];
```

Si lo parámetros son insertados en la consulta vía USING, de la misma forma que en un comando EXECUTE y declaramos la función con parámetros de salida, escribimos RETURN NEXT con no expression.

Cuando declaramos una función PL/pgSQL para devolver SETOF expresión, el procedimiento es diferente.

De cada ejecución, los valores reales de los parámetros de salida son grabados en una devolución eventual como una fila del resultado. Debemos declarar la función que devuelve un SETOF RECORD si hay múltiples parámetros de salida, o SETOF expression cuando hay un parámetro de salida de tipo expression, en orden de crear una función SET-RETURNING con parámetros de salida.

Códigos de funciones utilizando RETURN NEXT:

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');
```

```
CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
r foo%rowtype;
BEGIN
FOR r IN
SELECT * FROM foo WHERE fooid > 0
LOOP
-- can do some processing here
RETURN NEXT r; -- return current row of SELECT
END LOOP;
RETURN;
END
$BODY$
LANGUAGE plpgsql;
```

```
SELECT * FROM get_all_foo();
```

Códigos de funciones utilizando RETURN QUERY:

```
CREATE FUNCTION get_available_flightid(date) RETURNS SETOF integer
    AS
$BODY$
BEGIN
RETURN QUERY SELECT flightid
FROM flight
WHERE flightdate >= $1
AND flightdate < ($1 + 1);

-- Since execution is not finished, we can check whether rows were
    returned
-- and raise exception if not.
IF NOT FOUND THEN
RAISE EXCEPTION "No hay vuelos a %.', $1;
END IF;

RETURN;
END
$BODY$
LANGUAGE plpgsql;
```

Las cuales regresan formaciones disponibles o causan RAISES EXCEPTION si no las hay.

```
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

NOTA: *la implementación real de RETURN NEXT o RETURN QUERY almacena una resultado completo antes de devolver algo de la función. Esto significa que una función PL/pgSQL produce un conjunto resultante, la realización puede ser pobre: la data es escrita a disco para evitar colapso de memoria, ero la función misma no los devuelve hasta que el conjunto resultante es generado. Comúnmente, el punto en el cual la data empieza a ser escrita en disco es controlado por la variable de configuración work_mem. Los administradores de base de datos que tengan suficiente memoria para almacenar grandes conjunto resultantes deben considerar incrementar este parámetro.*

6.8. ¿Cómo ejecutar función de PostgreSQL en php con pdo?

Cómo llamar una función o procedimiento almacenado en PostgreSQL mediante PHP con PDO. Supongamos que tenemos una función en PostgreSQL:

```
1 buscarporcategoria(categoria character varying);
```

Y su contenido es el siguiente:

```
1  CREATE OR REPLACE FUNCTION public.buscarporcategoria(
2  _categoria_ character varying)
3  RETURNS TABLE(isbn character varying, titulo character varying,
       edicion integer, paginas integer, fecha date, nombre character
       varying, apellido character varying, categoria character
       varying, editorial character varying)
4  LANGUAGE 'plpgsql'
5
6  COST 100
7  VOLATILE
8  ROWS 1000
9  AS $BODY$
10
11
12 BEGIN
13 FOR nombre, apellido, isbn, titulo, edicion, paginas, fecha,
       categoria, editorial IN
14 SELECT
15 autores.nombre, autores.apellido,
16 libros.isbn, libros.titulo,
17 libros.edicion, libros.paginas, libros.fecha_publicacion,
18 categorias.categoria,
19 editoriales.editorial
20 FROM autores
21 INNER JOIN libros_por_autor ON autores.id = libros_por_autor.
       id_autores
```

```
22 INNER JOIN libros ON libros.id = libros_por_autor.id_libros
23 INNER JOIN libros_por_categoria ON libros_por_categoria.id_libros
      = libros.id
24 INNER JOIN categorias ON categorias.id = libros_por_categoria.
      id_categorias
25 INNER JOIN libros_por_editorial ON libros_por_editorial.id_libros
      = libros.id
26 INNER JOIN editoriales ON editoriales.id = libros_por_editorial.
      id_editoriales
27 WHERE categorias.categoria ILIKE '%'||_categoria_||'%'
28 LOOP
29 RETURN NEXT;
30 END LOOP;
31 RETURN;
32 END;
```

Esta consulta dentro del gestor funciona correctamente y devuelve lo requerido, si la ejecutamos desde PHP con PDO, nuevamente se ejecuta correcto:

```
1 $query = $this->con->prepare('SELECT buscarporcategoria(?)');
```

Realizamos un cambio de bindParam a bindValue puesto que la función no usa parámetro de entrada o salida sino más bien el valor "categoria", veamos:

```
1 $query->bindValue(1, $this->autor, PDO::PARAM_STR);
2 $query->execute();
3 $this->con->close();
4 return $query->fetch(PDO::FETCH_OBJ);
```

Es la forma correcta de ejecutar funciones de PostgreSQL y obtener los resultados como si la ejecuta desde el mismo gestor.

A propósito, así es como imprimimos los resultados de la consulta:

```
1  foreach( $libros as $libro ){
2    echo '<tr>
3    <td>'.$libro->isbn.'</td>
4    <td>'.$libro->titulo.'</td>
5    <td>'.$libro->edicion.'</td>
6    <td>'.$libro->paginas.'</td>
7    <td>'.$libro->fecha.'</td>
8    <td>'.$libro->nombre.'</td>
9    <td>'.$libro->apellido.'</td>
10   <td>'.$libro->categoria.'</td>
11   <td>'.$libro->editorial.'</td>
12   </tr>';
13
14 }
```

Señalamos que es conveniente hacer una prueba:

```
$result = $query->fetch(PDO::FETCH_ASSOC);
print "RESULTADO DE LA FUNCION";
print_r($result) ;
```

Antes de enviarlo a imprimir para saber si efectivamente se esta ejecutando la función o no.

7

Cursores

Los cursores son tablas temporales que ejecutan consultas; son soportados dentro de las funciones, como un SQL embebido, y pueden o no hacer copias de sus resultados; éstos, son de sólo lectura.

Un cursor es el nombre que recibe un apuntador (pointer) de sólo lectura hacia un conjunto de datos (resultset) que se obtiene de una consulta SQL asociada. Pensemos en términos de arreglos similar a los de un lenguaje de programación, los cursores procesan, una a una, las filas que componen un conjunto de resultados en vez de trabajar con todos los registros como en una consulta tradicional de SQL.

Los cursores pueden declararse en una consulta SQL con o sin parámetros, donde el tamaño del conjunto de datos depende del valor de los parámetros de la consulta.

Los cursores se emplean dentro de funciones PL/pgSQL para que las aplicaciones que accedan a PostgreSQL puedan utilizarlos más de una vez. Los cursores utilizan a:

Tabla 7.1: Comandos utilizados en los cursores

Comando	Descripción
BEGIN	Para indicar el comienzo de la operación.
DECLARE	Define un cursor para acceso a una tabla.
FETCH	Permite devolver las filas usando un cursor. El numero de filas devueltas es especificado por un número (#), este puede ser reemplazado por ALL que devuelve todas las filas del cursor. También, podemos utilizar los comandos BACKWARD y FORWARD para indicar la dirección.
CLOSE	Libera los recursos del cursor abierto.
COMMIT	Realiza la transacción actual.
END	Es un sinónimo en PostgreSQL de COMMIT. Finaliza la transacción actual.
ROLLBACK	Deshace la transacción actual tal que todas las modificaciones originadas por la misma sean restauradas.

Veamos el siguiente código:

```
1 BEING WORK;
2 DECLARE capital_1 CURSOR FOR
3 SELECT * FROM sucursal WHERE capital<20000;    -- CREA EL CURSOR
4 FETCH FORWARD 2 IN capital_1;      -- RECORRE ADELANTE DOS
    POSICIONES
5 FETCH BACKWARD 1 IN capital_1;     -- RECORRE ATRAS UNA POSICION
6 CLOSE capital_1;     -- CIERRA EL CURSOR
7 COMMIT WORK;     -- TERMINA LA TRANSACCION
```

Existen dos errores comunes cuando trabajamos con cursores:

- *Si tratas de abrir un cursor que ya se encuentra abierto PostgreSQL envía un mensaje de "cursor [name] already in use".*
- *Si tratas de ejecutar FETCH en un cursor que no ha sido abierto, PostgreSQL envía un mensaje de "cursor [name] is invalid".*

Con el comando FETCH obtenemos las filas, una por una, del conjunto de resultados; luego de cada fila es procesada, el cursor avanza a la siguiente fila y la fila procesada puede ser entonces utilizada dentro de una variable.

Además de ejecutar una consulta completa, al mismo tiempo, es posible configurar al cursor

para que la encapsule y luego lea del resultado de dicha consulta, obteniendo unas cuantas filas al mismo tiempo. Esto evita la sobrecarga de memoria cuando el resultado contiene numerosas filas. Los usuarios de PL/pgSQL no necesitan preocuparse por esto, ya que FOR utiliza un cursor interno para evitar problemas de memoria. Un uso adecuado es devolver una referencia a un cursor, facilitando al cliente leer las filas. Esto proporciona una forma eficiente de devolver conjuntos de filas desde las funciones.

7.1. Declarando variables para cursor

Todo acceso a los cursores en PL/pgSQL es a través de variables de cursor, las cuales son del tipo de dato especial refcursor. Una forma de crear esta variable es declararla como de tipo refcursor. Otra forma es usar la sintaxis de declaración de cursor, la cual es:

```
1 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 los nombres a ser reemplazados por los valores de parámetros en la consulta dada. Cuando el cursor es abierto sustituye los valores actuales para estos nombres, los cuales serán especificados más tarde. Veamos el código:

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

Las tres variables tienen el tipo de dato refcursor, pero la primera puede ser usada con cualquier consulta, la segunda tiene una consulta especificada para trabajar con ella, y la última tiene una consulta parametrizada (key es reemplazada por un parámetro de tipo entero cuando el cursor es abierto).

La variable curs1 es omitida, ya que no está asignada a una consulta en particular.

7.2. Tipos de Cursor

Los cursores, en una consulta, manipulan y procesan datos mediante algún lenguaje de programación. En PostgreSQL existen dos tipos de cursores, explícitos o implícitos, dependiendo de la necesidad de programación utilizamos uno u otro.

7.2.1. Cursor explícito

Los cursores explícitos son variables que almacenan datos de consultas de una o mas tablas; por tanto, el cursor está formado por una instrucción SELECT y debe ser declarado como una variable.

Veamos un código simple del uso de un cursor explícito:

```
1  -- CASO1 : Uso simple de cursores EXPLÍCITOS
2  CREATE OR REPLACE FUNCTION expl_cursor1() RETURNS SETOF clientes
       AS
3  $BODY$
4  DECLARE
5  -- Declaración explicita del cursor
6  cur_clientes CURSOR FOR SELECT * FROM clientes;
7  registro clientes%ROWTYPE;
8  BEGIN
9  -- Procesa el cursor
10 FOR registro IN cur_clientes LOOP
11 RETURN NEXT registro;
12 END LOOP;
13 RETURN;
14 END $BODY$ LANGUAGE 'plpgsql'
```

7.2.2. Cursor implícito

Por otra parte, los cursores implícitos están insertos directamente en el código como una instrucción SELECT, sin ser declarados previamente y, generalmente, requieren de alguna variable del tipo RECORD o ROWTYPE para las filas.

Un código simple del uso de cursor implícito es:

```
1  -- CASO2 : Uso simple de cursores IMPLICITOS
2  CREATE OR REPLACE FUNCTION impl_cursor2() RETURNS SETOF clientes
       AS
3  $BODY$
4  DECLARE
5  registro clientes%ROWTYPE;
6  BEGIN
7  -- Cursor IMPLICITO en el ciclo FOR
8  FOR registro IN SELECT * FROM clientes LOOP
9  RETURN NEXT registro;
10 END LOOP;
11 RETURN;
12 END $BODY$ LANGUAGE 'plpgsql'
```

Es probable que entre los dos casos antes descritos, resulte mas simple usar cursores implícitos ya que se declaran directamente en el ciclo FOR; sin embargo, muchas veces las consultas resultan extensas y/o complejas y terminan complicando la lectura del código, o bien es necesario reutilizar el cursor en mas de una oportunidad, en estos casos lo conveniente es usar cursores explícitos, todo depende de como se aborde el problema, pero la solución existe.

7.3. Abriendo Cursores

Antes de que el cursor sea utilizado para regresar filas, debe ser abierto (acción equivalente al comando SQL DECLARE CURSOR). PL/pgSQL tiene cuatro formas para el comando OPEN; dos usan variables cursor no asignadas y las otras dos usan variables cursor asignadas.

7.3.1. OPEN FOR SELECT

```
1 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 comandos SELECT en PL/pgSQL: los nombres de variables PL/pgSQL son sustituidos, y el plan de consulta es almacenado para su posible reutilización.

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

7.3.2. OPEN FOR EXECUTE

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

La variable cursor es abierta y se le pasa la consulta especificada a ser ejecutada.

El cursor no puede ser abierto aún, y declarado como cursor no asignado (esto es, una simple 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.

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

7.3.3. OPEN bound-cursor [(argument_values)]

Esta forma de OPEN es usada para abrir una variable cursor cuya consulta es asignada cuando ésta fue declarada.

```
1 OPEN bound-cursor [ ( argument\_values ) ];
```

El cursor no puede ser abierto todavía. Una lista de los argumentos de los valores de las expresiones debe aparecer si, y sólo si, el cursor fue declarado para tomar argumentos. Estos valores son 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-.

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

7.4. Usando Cursores

Una vez que el un cursor es abierto, puede ser manipulado con los comandos descritos en el B, página 176.

La manipulación no necesitan ocurrir en la misma función que abrió el cursor. Puede devolver un valor refcursor de una función y operar al cliente con el cursor (internamente, un valor refcursor es una cadena conteniendo la consulta activa para el cursor. Esta variable o cadena puede ser pasada y/o asignada a otras variables refcursor, sin problemas).

Al final de la transacción, implícitamente, todo es cerrado. Por tanto, un valor refcursor es útil para referir un cursor abierto sólo hasta el final de la transacción.

7.4.1. FETCH

FETCH recupera la siguiente fila del cursor en una variable de fila, de registro o varias separadas por comas. Como en una lista de variables simples, al igual que SELECT INTO. Si no hay fila siguiente, el objetivo se establece en NULL(s). Y al igual que con SELECT INTO, la función especial FOUND comprueba si se ha obtenido una fila o no.

```
1 FETCH [ direction { FROM | IN } ] cursor INTO target;
```

La cláusula direction puede ser cualquiera de las variantes permitidas en el comando FETCH de SQL, excepto las que recuperan más de una fila; es decir, puede ser NEXT, PRIOR, FIRST, LAST, ABSOLUTE count, RELATIVE count, FORWARD, o BACKWARD. Omitir direction es lo mismo que especificar SIGUIENTE. Si requiere BACKWARD, los valores de direction probablemente fallarán a menos que se haya declarado el cursor o que se abra con esta opción.

Una variable refcursor debe ser el nombre que hace referencia a un cursor abierto. Códigos:

```
1 FETCH curs1 INTO rowvar;
2 FETCH curs2 INTO foo, bar, baz;
3 FETCH LAST FROM curs3 INTO x, y;
4 FETCH RELATIVE -2 FROM curs4 INTO x;
```

y en notación simple es:

```
1 FETCH cursor INTO target;
```

7.4.2. CLOSE

Para cerrar un cursor abierto, utilizamos:

```
1 CLOSE cursor;
```

También, puede ser usado para liberar recursos antes del final de una transacción, o para liberar una variable cursor para abrirla de nuevo.

```
1 CLOSE curs1;
```

7.4.3. MOVE

MOVE posiciona un cursor sin devolver datos; trabaja como el comando FETCH, excepto que sólo posiciona el cursor y no recupera la fila actual. Como con SELECT INTO, la variable FOUND verifica que haya una siguiente fila a recuperar.

```
MOVE [ direction { FROM | IN } ] cursor;
```

Códigos:

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

7.4.4. UPDATE/DELETE WHERE CURRENT OF

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

Cuando un cursor es ubicado en la fila de una tabla, esa fila es actualizada o eliminada usando el cursor para identificarla. Hay restricciones para las cuales, la consulta del cursor puede ser no agrupada, por lo que es mejor utilizar FOR UPDATE en el cursor.

Un código:

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

7.5. Retornando Cursores

Las funciones PL/pgSQL devuelven cursores al cliente. Acción usada para devolver múltiples filas o columnas desde la función, la cual abre el cursor y devuelve el nombre del cursor. Éste, se puede recuperar (con FETCH) filas desde el cursor. El cursor puede ser cerrado por el remitente, o bien cerrado cuando termine la transacción.

El nombre del cursor devuelto por la función puede ser especificado por el cliente o generado automáticamente. El siguiente código muestra cómo un nombre de cursor es proporcionado por el cliente:

```
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 código usa la generación automática para nombrar un 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 código nombra un cursor que puede ser suministrado a un cliente:

```
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 código genera el nombre de un cursor:

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
ref refcursor;
BEGIN
OPEN ref FOR SELECT col FROM test;
RETURN ref;
END;
' LANGUAGE plpgsql;
```

Es necesario utilizar una transacción para usar cursores.

```
BEGIN;
SELECT reffunc2();
reffunc2
--------------------
<unnamed cursor 1>
(1 row)
FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

El siguiente código regresa múltiples cursores con una única función:

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF
   refcursor AS $$
BEGIN
OPEN $1 FOR SELECT * FROM table_1;
RETURN NEXT $1;
OPEN $2 FOR SELECT * FROM table_2;
RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;
```

Debe ser una transacción para usar cursores.

```
BEGIN;
SELECT * FROM myfunc('a', 'b');
FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

7.5.1. Ciclos a través del resultado de un cursor

Hay una variante de FOR para iterar a través de las filas recuperadas por un cursor. La sintaxis es:

```
[ <<label>> ]
FOR recordvar IN bound_cursorvar [ ( [ argument_name := ]
   argument_value [, ...] ) ] LOOP
statements
END LOOP [ label ];
```

La variable del cursor está ligada a una consulta cuando es declarada, sino no puede ser abierta. FOR abre el cursor y lo cierra cuando sale del ciclo. Una lista de argumentos aparece si, y sólo si, el cursor fue declarado con ellos. Los valores son sustituidos en la consulta, igual que ocurre en un OPEN.

La variable recordvar es definida de tipo record y sólo existe dentro del ciclo (cualquier definición de la variable es ignorada en el ciclo). Cada fila recuperada por el cursor es sucesivamente asignada a la variable record y el cuerpo del ciclo es ejecutado.

7.6. Declarando cursor con variables

El acceso a los cursores es a través de variables del tipo refcursor. Una forma de crear una variable para un cursor es declararla como refcursor. Otra, es utilizar la sintaxis de la declaración del cursor, la cual es:

```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR query;
```

FOR puede ser reemplazado por IS para compatibilidad con Oracle.

Si SCROLL es especificado, el cursor de desplaza hacia atrás; sino, es rechazado; si no hay especificación, es una consulta dependiente. Si especificamos argumentos, con comas separamos la lista de los tipos de datos definidos a ser reemplazado por valores de parámetros en una consulta dada. Cuando el cursor es abierto, los valores actuales sustituyen esos nombres.

7.6.1. Trabajando con fechas

Si realizamos una función que reciba como parámetros una fecha desde y una fecha hasta, cargamos el cursor con los datos de operaciones, lea el cursor y grabe dichos datos en la tabla "operacionespecial".

```
CREATE OR REPLACE FUNCTION OperacionCursor(FechDesde date,
   FechHasta date) RETURNS VOID AS
$OpCur$
DECLARE
OperCursor CURSOR FOR SELECT * FROM Tb_Detalle_Operacion
WHERE FechaOperacion BETWEEN FechDesde AND FechHasta;
Contenido Tb_Detalle_Operacion %ROWTYPE;
BEGIN
OPEN OperCursor;
LOOP
FETCH OperCursor INTO Contenido;
IF NOT FOUND THEN
EXIT;
END IF;
INSERT INTO Tb_OperacionesEspeciales(NroOperac, FechaOperacion,
   TotalOper,
MatriculaOdontologo, CodPaciente)
VALUES (Contenido.NroOperac, Contenido.FechaOperacion, Contenido.
   TotalOper,
Contenido.MatriculaOdontologo, Contenido.CodPaciente);
END LOOP;
CLOSE OperCursor;
END;
$OpCur$
LANGUAGE 'plpgsql'
```

El código funciona muy bien, tanto en la parte de argumentos como en la insertar. Es un caso típico que encontraremos con frecuencia en nuestros requerimientos de base de datos.

8

Procedimientos

Este capítulo presenta los llamados procedimientos almacenados[1] *en PL/pgSQL. Un procedimiento se define como un programa o función almacenado en la base de datos y listo para ser usado.*

Hay dos ventajas al utilizar estos procedimientos almacenados:

1. *La ejecución del procedimiento ocurre en el servidor de bases de datos. Lo cual aumenta el rendimiento de una aplicación al no haber tránsito de datos entre el cliente y el servidor, y no tener que procesar resultados intermedios para obtener el resultado final.*

2. *Al tener la lógica de la aplicación implementada en la base de datos no tenemos que implantarla en los clientes, con el consiguiente ahorro de lineas de código redundante y de complejidad.*

Si tenemos diferentes tipos de clientes implementados en diversos sistemas o lenguajes de programación y accediendo concurrentemente a la misma base de datos, no tenemos que programar la misma lógica en todos ellos, ya que están disponibles en la base de datos. Tenemos una API[2] *a la lógica de la aplicación lista para usarse desde los diferentes clientes.*

Un procedimiento se puede escribir en múltiples lenguajes de programación. En una instalación por defecto de PostgreSQL podemos tener varios lenguajes, vea el capítulo 11, página 126.

El único lenguaje disponible automáticamente es PL/pgSQL. Para utilizar PL/Perl, PL/Tcl o PL/Python debemos configurar/compilar PostgreSQL con estos parámetros:

```
1 --with-perl --with-tcl
2 --with-python.\salto
```

Definimos e instalamos un procedimiento en PL/pgSQL de la siguiente manera:

[1] stored procedures.

[2] Una API es un conjunto de funciones y procedimientos que cumplen una o muchas funciones con el propósito de ser utilizadas por otro software. Las siglas API vienen del inglés Application Programming Interface.

```
CREATE [ OR REPLACE ] FUNCTION
nombre_funcion([ [ argmodo ] [ argnombre ] argtipo [, ...] ])
RETURNS tipo AS $$
[ DECLARE ]
[ declaraciones de variables ]
BEGIN
  aquí escribimos el código
END;
$$ LANGUAGE plpgsql
| IMMUTABLE | STABLE | VOLATILE
| CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
| [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
| COST execution_cost
| ROWS result_rows
| SET configuration_parameter { TO value | = value | FROM CURRENT
   }
;
```

Parece complicado, pero no, es más fácil de lo que parece y las opciones después de "language plpgsql" tienen unos valores por defecto que simplifican la definición de un procedimiento.

A continuación, veamos algunas de las opciones y valores más importantes.

argmodo: El modo de un argumento puede ser IN, OUT, or INOUT. Por defecto, es IN si no se define.

argtipo: Los tipos que utilizamos son los disponibles en PostgreSQL y los definidos por el usuario.

Las declaraciones de estas variables se realizan de la siguiente manera ($n = orden de declaración del argumento.):

```
nombre_variable ALIAS FOR $n;
nombre_variable [ CONSTANT ] tipo [ NOT NULL ] [ { DEFAULT | := }
   expresion ];
```

El cuerpo del procedimiento: en este lugar escribimos el código de un procedimiento. Como vemos se mantiene la estructura de dos partes: declaración y cuerpo principal del programa. *Ver Sección 3.7, página 46.*

8.1. Estructuras de control: IMMUTABLE | STABLE | VOLATILE

- *IMMUTABLE: Indica que la función no puede alterar a la base de datos y que siempre devuelve el mismo resultado, dados los mismos valores como argumentos. Estas funciones no realizan consultas en la base de datos.*
- *STABLE: Indica que la función no puede alterar a la base de datos y que siempre*

devuelve el mismo resultado en una consulta individual de una tabla, dados los mismos valores como argumentos. El resultado podría cambiar entre sentencias SQL.

- *VOLATILE: Indica que la función puede devolver diferentes valores, incluso dentro de una consulta individual de una tabla (valor por defecto).*

CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT:

- *CALLED ON NULL INPUT: Indica que la función se ejecuta aunque algunos de sus argumentos sean NULL. El usuario es responsable de comprobar si algún argumento es NULL (valor por defecto).*
- *RETURNS NULL ON NULL INPUT / STRICT: Indican que la función no se ejecuta y devuelve el valor NULL si alguno de sus argumentos es NULL.*
- *SECURITY INVOKER | SECURITY DEFINER:*
- *SECURITY INVOKER: Indica que la función se ejecuta con los privilegios del usuario que la llama (valor por defecto)*
- *SECURITY DEFINER: Indica que la función se ejecuta con los privilegios del usuario que la creó.*

El resto de opciones son avanzadas y podemos leer sobre ellas en la documentación oficial.

Aplicando la teoría que hemos visto, veamos unos cuantos códigos que aclaren como definir, instalar y usar un procedimiento almacenado en PL/pgSQL (estos códigos han sido comprobados en PostgreSQL 9.6.15 bajo sistema operativo Debian 9.9).

Creamos una base de datos para utilizarla con los códigos:

```
postgres@server:~$ psql
Welcome to psql (PostgreSQL) 9.6.15, the PostgreSQL interactive
   terminal.
Type:
\copyright for distribution terms
\h for help with SQL commands
\? for help with psql commands
\g or terminate with semicolon to execute query
\q to quit
postgres=# CREATE DATABASE test001;
CREATE DATABASE
postgres=# \c test001
You are now connected to database "test001".
debian=#
```

Lo primero es instalar el lenguaje PL/pgSQL, si no lo tenemos instalado.

```
CREATE PROCEDURAL LANGUAGE plpgsql;
```

Para que un usuario con acceso a la base de datos pueda usarla, sin ser el administrador PostgreSQL, utilizamos TRUSTED en el comando anterior.

```
1 CREATE TRUSTED PROCEDURAL LANGUAGE plpgsql;
```

A continuación, creamos el primer procedimiento.

```
1 CREATE OR REPLACE FUNCTION ejemplo() RETURNS integer AS $$
2 BEGIN
3 RETURN 104;
4 END;
5 $$ LANGUAGE plpgsql;
```

Este procedimiento se utiliza de la siguiente manera:

```
1 debian=# SELECT ejemplo();
2 ejemplo1
3 ----------
4 104
5 (1 row)
```

Ahora, definimos la función con un argumento:

```
1 CREATE OR REPLACE FUNCTION ejemplo(integer) RETURNS integer AS $$
2 BEGIN
3 RETURN $1;
4 END;
5 $$ LANGUAGE plpgsql;
```

Este procedimiento se escribe:

```
1 CREATE OR REPLACE FUNCTION ejemplo(numero integer) RETURNS integer
      AS $$
2 BEGIN
3 RETURN numero;
4 END;
5 $$ LANGUAGE plpgsql;
```

y

```
1 CREATE OR REPLACE FUNCTION ejemplo(integer) RETURNS integer AS $$
2 DECLARE
3 numero ALIAS FOR $1;
4 BEGIN
5 RETURN numero;
6 END;
7 $$ LANGUAGE plpgsql;
```

Este procedimiento se usa de la siguiente manera:

```
1 debian=# SELECT ejemplo(104);
2 ejemplo
3 ---------
```

```
104
(1 row)
```

Empecemos a complicar un poco las cosas usando dos argumentos y definiendo algunas variables:

```
CREATE OR REPLACE FUNCTION ejemplo(integer, integer) RETURNS
   integer AS $$
DECLARE
numero1 ALIAS FOR $1;
numero2 ALIAS FOR $2;
constante CONSTANT integer := 100;
resultado integer;
BEGIN
resultado := (numero1 * numero2) + constante;
RETURN resultado;
END;
$$ LANGUAGE plpgsql;
```

Este procedimiento se usa así:

```
debian=# SELECT ejemplo(2,2);
ejemplo
---------
104
(1 row)
```

A continuación, utilizamos una sentencia IF ... THEN en nuestra función:

```
CREATE OR REPLACE FUNCTION ejemplo_txt(integer, integer) RETURNS
   text AS $$
DECLARE
numero1 ALIAS FOR $1;
numero2 ALIAS FOR $2;
constante CONSTANT integer := 100;
resultado INTEGER;
resultado_txt TEXT DEFAULT 'El resultado es 104';
BEGIN
resultado := (numero1 * numero2) + constante;
IF resultado <> 104 THEN
resultado_txt :=
'El resultado NO es 104';
END IF;
RETURN resultado_txt;
END;
$$ LANGUAGE plpgsql;
```

Este procedimiento se usa de la siguiente manera:

```
debian=# SELECT ejemplo_txt(2,2);
ejemplo_txt
```

```
----------------------
El resultado es 104
(1 row)
debian=# SELECT ejemplo_txt(2,3);
ejemplo_txt
-------------------------
El resultado NO es 104
(1 row)
```

Podríamos seguir modificando y complicando el código, pero como introducción es suficiente para tener una idea de su funcionamiento.

Queda por decir que en la definición de un procedimiento no sólo se tiene en cuenta el nombre del mismo para diferenciarlo de otros, sino que también los argumentos de la función se tienen en cuenta. Como: ejemplo(), ejemplo(integer), ejemplo(integer, integer) y ejemplo(text) todos ellos procedimientos diferentes, aunque se llamen igual.

En psql existe un comando que muestra como una función está definida en la base de datos, este ejemplo se trabaja con psql desde la consola.

```
debian=# \x
Expanded display is on.
debian=# \df+ ejemplo
List of functions
-[ RECORD 1 ]-------+----------------------
Schema | public
Name | ejemplo
Result data type | integer
Argument data types |
Volatility | volatile
Owner | postgres
Language | plpgsql
Source code |
: BEGIN
:
RETURN 104;
: END;
:
Description
|
-[ RECORD 2 ]-------+----------------------
Schema | public
Name | ejemplo
Result data type | integer
Argument data types | integer
Volatility | volatile
Owner | postgres
Language | plpgsql
```

```
Source code |
: DECLARE
:
numero ALIAS FOR $1;
:
: BEGIN
:
RETURN numero;
: END;
:
Description
|
-[ RECORD 3 ]--
     -----+-------------------------------------------------
Schema | public
Name | ejemplo
Result data type | integer
Argument data types | integer, integer
Volatility | volatile
Owner | postgres
Language | plpgsql
Source code |
: DECLARE
: numero1 ALIAS FOR $1;
: numero2 ALIAS FOR $2;
:
: constante CONSTANT integer := 100;
: resultado integer;
:
: BEGIN
:
resultado := (numero1 * numero2) + constante;
:
:
RETURN resultado;
: END;
:
Description
|
debian=# \df+ ejemplo_txt
List of functions
-[ RECORD 1 ]--
     -----+-------------------------------------------------------
Schema | public
Name | ejemplo_txt
Result data type | text
```

```
Argument data types | integer, integer
Volatility | volatile
Owner | postgres
Language | plpgsql
Source code |
: DECLARE
: numero1 ALIAS FOR $1;
: numero2 ALIAS FOR $2;
:
: constante CONSTANT integer := 100;
: resultado INTEGER;
:
:
resultado_txt TEXT DEFAULT 'El resultado es 104';
:
: BEGIN
:
resultado := (numero1 * numero2) + constante;
:
:
:
IF resultado <> 104 THEN
resultado_txt :=
Create PDF in your applications with the Pdfcrowd HTML to PDF API
'El resultado NO es 104';
:
END IF;
:
:
RETURN resultado_txt;
: END;
:
Description
|
```

El resto es sólo cuestión de imaginación y de leer la literatura disponible a tal efecto. Las posibilidades son infinitas y una vez que empecemos a usar procedimientos almacenados no dejaremos de usarlos.

8.2. Regresando un Result Set de un procedimiento

Para obtener uno o más Result sets (cursores en PostgreSQL), utilizamos el tipo RETURN refcursor.

Código rápido:

```
-- Procedure that returns a single result set (cursor)
```

```
CREATE OR REPLACE FUNCTION show_cities() RETURNS refcursor AS $$
DECLARE
-- Declara una variable cursor
ref refcursor;
BEGIN
-- abre un cursor
OPEN ref FOR SELECT city, state FROM cities;
-- regresa el cursor
RETURN ref;
END;
$$ LANGUAGE plpgsql;
```

8.3. Regresando Múltiples Result Sets

Para regresar múltiples result set, especificamos un tipo RETURNS SETOF refcursor y usamos RETURN NEXT para regresar cada cursor:

```
-- Procedure that returns multiple result sets (cursors)
CREATE OR REPLACE FUNCTION show_cities_multiple() RETURNS SETOF
   refcursor AS $$
DECLARE
ref1 refcursor;           -- Declare cursor variables
ref2 refcursor;
BEGIN
-- Abrir primer cursor
OPEN ref1 FOR SELECT city, state FROM cities WHERE state = 'CA';
-- Return the cursor to the caller
RETURN NEXT ref1;
-- Abrir segundo cursor
OPEN ref2 FOR SELECT city, state FROM cities WHERE state = 'TX';
-- Return the cursor to the caller
RETURN NEXT ref2;
END;
$$ LANGUAGE plpgsql;
```

El procesamiento y diseño del procedimiento que regresa un result sets puede depender del cliente.

Cuando el cliente es PL/pgSQL, utilizamos PgAdmin, PgAccess u otra herramienta, incluso el terminal mismo, asumimos que un cliente llama un procedimiento y la salida del result set está en una herramienta PSQL, PgAdmin, PgAdmin o en otra función:

```
SELECT show_cities();
```

El resultado:

```
show_cities refcursor
<unnamed portal 1>
```

Esta consulta sólo regresa el nombre del cursor, no las filas del result set. Para obtenerlas, necesitamos usar la sentencia FETCH y especificar el nombre del cursor:

```
FETCH ALL IN "<unnamed portal 1>";
-- ERROR:  cursor "<unnamed portal 4>" does not exist
```

El problema es que el cursor está cerrado, porque no usamos una transacción. Iniciemos una, ejecutemos el procedimiento, y recuperamos las filas con FETCH:

```
-- Start a transaction
BEGIN;

SELECT show_cities();
-- Returns: <unnamed portal 2>

FETCH ALL IN "<unnamed portal 2>";
COMMIT;
```

Salida:

Tabla 8.1: Ciudades de Aragua y Carabobo, Venezuela

Ciudad	Estado
Maracay	Aragua
San Mateo	Aragua
La Victoria	Aragua
Cagua	Aragua
Turmero	Aragua
Valecia	Carabobo

8.4. Problema con el nombre del cursor

Hemos observado que el nombre del cursos puede cambiar, y esto es un inconveniente para: primero recuperar el nombre del cursor y, luego, usarlo en la sentencia FETCH.

Como opción podemos rediseñar el procedimiento y pasar el nombre del cursor como parámetro, tal que el cliente conozca cuál cursor traer:

```
-- Procedure that returns a cursor (its name specified as the
   parameter)
CREATE OR REPLACE FUNCTION show_cities2(ref refcursor) RETURNS
   refcursor AS $$
BEGIN
OPEN ref FOR SELECT city, state FROM cities;   -- Open a cursor
RETURN ref;             -- Return the cursor to the caller
END;
$$ LANGUAGE plpgsql;
```

Ahora, el cliente especifica un nombre predefinido:

```
-- Start a transaction
BEGIN;

SELECT show_cities2('cities_cur');
-- Returns: cities_cur

FETCH ALL IN "cities_cur";
COMMIT;
```

8.5. Procesando múltiples Result Sets

Si llamamos un procedimiento que regresa múltiples result sets en una aplicación PSQL, PgAdmin, PgAccess o en otra función, la consulta regresa los nombres de los cursores:

```
SELECT show_cities_multiple();
The result:

show_cities_multiple refcursor
<unnamed portal 3>
<unnamed portal 4>
So to fetch data, you can use a separate FETCH statements for each
    cursor.

-- Start a transaction
BEGIN;

SELECT show_cities_multiple();

FETCH ALL IN "<unnamed portal 3>";
FETCH ALL IN "<unnamed portal 4>";
COMMIT;
```

Salidat (2 result sets):

Tabla 8.2: Sólo ciudades de Aragua, Venezuela

Ciudad	Estado
Maracay	Aragua
La Victoria	Aragua
Cagua	Aragua

y

Tabla 8.3: Sólo ciudades de Carabobo, Venezuela

Ciudad	Estado
Tocuyito	Carabobo
Valecia	Carabobo

Rediseñamos la función, y pasamos el nombre del cursor como parámetro a los cursores para recibir los nombres de cursos predefinido:

```
1  -- Procedure that accepts cursor names as parameters
2  CREATE OR REPLACE FUNCTION show_cities_multiple2(ref1 refcursor,
       ref2 refcursor)
3  RETURNS SETOF refcursor AS $$
4  BEGIN
5  OPEN ref1 FOR SELECT city, state FROM cities WHERE state = '
       Aragura';   -- Open the first cursor
6  RETURN NEXT ref1;

       -- Return the cursor to the caller
7
8  OPEN ref2 FOR SELECT city, state FROM cities WHERE state = '
       Carabobo';   -- Open the second cursor
9  RETURN NEXT ref2;

       -- Return the cursor to the caller
10 END;
11 $$ LANGUAGE plpgsql;
```

Ahora, proporcionamos los nombres de cursor:

```
1  -- Start a transaction
2  BEGIN;
3
4  SELECT show_cities_multiple2('ca_cur', 'tx_cur');
5
6  FETCH ALL IN "ca_cur";
7  FETCH ALL IN "tx_cur";
8  COMMIT;
```

8.6. Procedimiento almacenado con ciclos anidados

El objetivo es actualizar los datos de 2 columnas de una tabla si, y sólo si, hay datos para esa columna.

```
1  CREATE OR REPLACE FUNCTION p_update_locales_in_count() RETURNS
       INTEGER AS $BODY$
2  DECLARE
```

```
query_count          RECORD;
query_has_locales    RECORD;
real_data            RECORD;
BEGIN

FOR query_count IN  SELECT  co.neighborhood_code,co.city_code
FROM    conteo_manzanas_barrio_co co,sm_city ci
WHERE   co.city_code = ci.pk_city AND has_locales = TRUE
LIMIT 1 OFFSET 0 LOOP

SELECT  INTO real_data  sl.pk_locale,sl.name
FROM    sm_locale sl,sm_neighborhood sn,servcon_barrios sb
WHERE   sn.pk_neighborhood = query_count.neighborhood_code AND
sl.fk_pk_city = query_count.city_code AND
sb.cod_localidad = sl.locale_code AND
sb.cod_barrio = sn.neighborhood_code AND
sl.name = sb.nom_localidad;

UPDATE  conteo_manzanas_barrio_co
SET     locale_code = real_data.pk_locale, locale_name = real_data
   .name
WHERE   neighborhood_code = query_count.neighborhood_code;

END LOOP;

RETURN 0;
END $BODY$
LANGUAGE plpgsql;
```

Ahora, lo llamamos así:

```
SELECT * FROM p_update_locales_in_count();
```

Esta es la forma de llamar procedimientos que tienen múltiples salidas, este procedimiento sólo regresa un valor true o false, entonces con llamarlo:

```
select p_update_locales_in_count()
```

es suficiente.

8.7. Tópico de estudio

El diagrama siguiente representa una base de datos para una universidad:

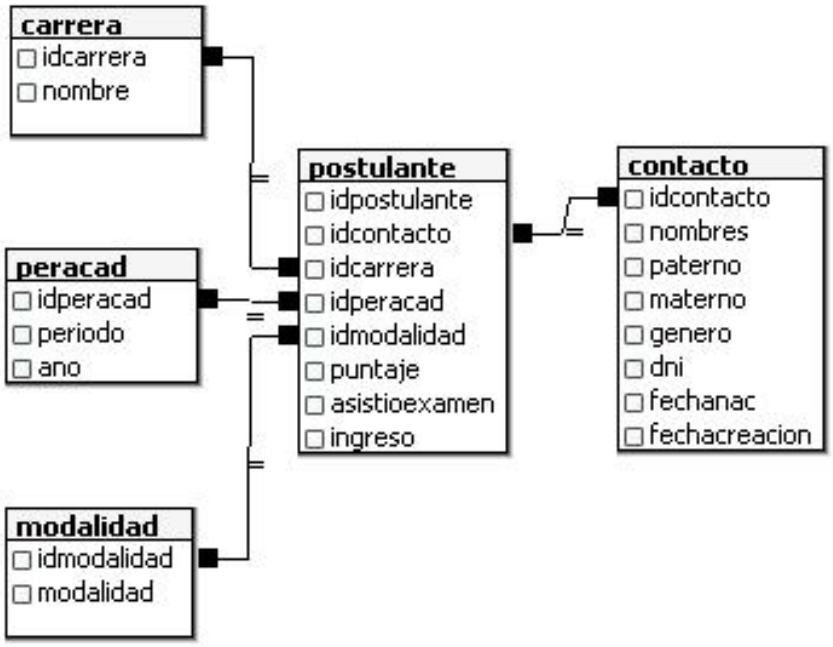


Figura 8.1: Base de datos de una Universidad

Es un modelo pequeño, pero sirve para nuestras futuras demostraciones con PL/pgSQL.

8.7.1. Descripción

Una universidad realiza el registro de contactos, que son las personas que podrían ser potenciales postulantes a diferentes carreras. Los postulantes deben pertenecer a un periodo académico y elegir una modalidad así como la carrera a la que quieren postular. Por lo general en un año solo existen dos periodos académicos, como en el año 2018 fueron: 2018-1 y 2018-2. El contacto debe tener los datos personales del sujeto así como la fecha de creación. El postulante debe tener registrado si asistió a su examen de admisión, así como el puntaje que alcanzó y si ingresó o no.

Con esos datos, comenzamos a escribir nuestras primeras consultas.

Debemos aclarar en es una buena costumbre crear funciones cuando se trata de recuperar datos, como por ejemplo una consulta que utilice la cláusula SELECT, y procedimientos almacenados únicamente cuando se trata de realizar una operación INSERT, UPDATE o DELETE.

En PL/pgSQL utilizamos un procedimiento almacenado para cualquiera de los 2 casos descritos, respetando la funcionalidad de cada uno.

8.7.2. Funciones

1) Creamos una función que devuelva los siguientes datos para las carreras ofrecidas por la universidad:

Periodo	101	309	310	Total
2005-1	1	7	7	15
2005-2	0	13	12	25
2006-1	0	17	18	35
2006-2	0	23	22	45
2007-1	0	27	28	55

```
CREATE OR REPLACE FUNCTION f_ejemplo_1()
RETURNS SETOF "record" AS
$BODY$
DECLARE
r RECORD;
BEGIN
FOR r IN
SELECT idperacad AS "Periodo"
,SUM(CASE WHEN IDCarrera = '101' THEN 1 ELSE 0 END ) AS "101"
,SUM(CASE WHEN IDCarrera = '309' THEN 1 ELSE 0 END ) AS "309"
,SUM(CASE WHEN IDCarrera = '310' THEN 1 ELSE 0 END ) AS "310"
,COUNT(*) AS "TOTAL"
FROM Persona.Postulante
GROUP BY IDPerAcad
LOOP
RETURN NEXT r;
END LOOP;
RETURN;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE;
```

```
SELECT * FROM f_ejemplo_1() AS ("Periodo" CHARACTER, "101" bigint,
   "309"
bigint,"310" bigint,"TOTAL" bigint);
```

2) Creamos una función que devuelva los siguientes datos de los estudiantes por carrera y periodo académico en la universidad:

Periodo	101	309	310	Total
2005-1	1	7	7	15
2005-2	0	13	12	25
2006-1	0	17	18	35
2006-2	0	23	22	45
2007-1	0	27	28	55
TOTAL	1	87	87	175

```
CREATE OR REPLACE FUNCTION f_ejemplo_2()
RETURNS SETOF "record" AS
$BODY$
DECLARE
r RECORD;
BEGIN
FOR r IN
SELECT idperacad AS "Periodo"
,SUM(CASE WHEN IDCarrera = '101' THEN 1 ELSE 0 END ) AS "101"
,SUM(CASE WHEN IDCarrera = '309' THEN 1 ELSE 0 END ) AS "309"
,SUM(CASE WHEN IDCarrera = '310' THEN 1 ELSE 0 END ) AS "310"
,COUNT(*) AS "TOTAL"
FROM Persona.Postulante
GROUP BY IDPerAcad
UNION
SELECT 'TOTAL' AS "Periodo"
,SUM(CASE WHEN IDCarrera = '101' THEN 1 ELSE 0 END ) AS "101"
,SUM(CASE WHEN IDCarrera = '309' THEN 1 ELSE 0 END ) AS "309"
,SUM(CASE WHEN IDCarrera = '310' THEN 1 ELSE 0 END ) AS "310"
,COUNT(*) AS total
FROM Persona.Postulante
LOOP
RETURN NEXT r;
END LOOP;
RETURN;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE;
```

```
SELECT * FROM f_ejemplo_2() AS ("Periodo" CHARACTER, "101" bigint,
   "309" bigint,"310" bigint,"TOTAL" bigint);
```

3) Creamos una función que liste todos los postulantes, la lista debe estar enumerada y ordenada por apellido paterno, materno y nombres.

Nº	Apellidos y Nombres
1	Alanya Padilla Alina Susan
2	Alarcon Castro Gustavo Claudio Andres
3	Alarco Lama Ricardo Rafael
...	...

```
CREATE OR REPLACE FUNCTION f_ejemplo_4()
RETURNS SETOF "record" AS
$BODY$
DECLARE
r RECORD;
```

```
BEGIN
FOR r IN
SELECT rownumber() "Numero",* FROM
(SELECT PC.Paterno || ' ' || PC.Materno || ' ' || PC.Nombres "
   Apellidos y Nombres"
FROM Persona.Contacto PC
INNER JOIN Persona.Postulante PP ON PP.IDContacto=PC.IDContacto
ORDER BY PC.Paterno,PC.Materno,PC.Nombres) AS tb2
LOOP
RETURN NEXT r;
END LOOP;
RETURN;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE;
```

```
--SELECT * FROM f_ejemplo_4() AS ("Numero" INTEGER, "Apellidos y
   Nombres" TEXT);
```

```
CREATE OR REPLACE FUNCTION rownumber() RETURNS integer AS $$
BEGIN
EXECUTE 'CREATE TEMP SEQUENCE "'||current_timestamp||'"';
RETURN nextval('"'||current_timestamp||'"');
EXCEPTION WHEN duplicate_table THEN RETURN nextval('"'||
   current_timestamp||'"');
END
$$ LANGUAGE 'plpgsql';
;;
```

4) Creamos un procedimiento almacenado para eliminar los postulantes registrados correspondientes a una modalidad.

```
CREATE OR REPLACE PROCEDURE sp_ejemplo_12(p_IDModalidad varchar)
AS
BEGIN
DELETE FROM Persona.Postulante
WHERE IDModalidad=p_IDModalidad;

END
--EXEC sp_ejemplo_12('2');
--SELECT * FROM Persona.Postulante;
```

9

Triggers

Los triggers[1] *son una parte del SQL estándar y muy utilizados en aplicaciones donde hay un tratamiento de datos intensivo por naturaleza. Un trigger está asociado con una tabla, vista o tabla externa y es disparado siempre que ocurre un evento sobre ella. Los eventos triggers son INSERT, DELETE, UPDATE o TRUNCATE.*

Los trigger ayudan a cumplir restricciones, gestionar errores y supervisar los datos; son clasificados de acuerdo a: trigger antes (BEFORE), después (AFTER), o en vez de (instead of) sobre una transacción. Y son referidos como trigger BEFORE, trigger AFTER, y trigger INSTEAD OF respectivamente.

Existen dos tipos de trigger disponibles, a nivel de ROW (fila) el cual es activado cada vez que una fila es afectada por la sentencia que lo activa y de comando, que es activado una vez que fue ejecutado aún si no afecta cualquier fila. Mientras que los trigger TRUNCATE, ligados a una operación o vista, se activan a nivel comando.

Un trigger llama a una función escrita en PL/SQL or PL/pgSQL o en cualquier otro lenguaje compatible.

9.1. Características y reglas

Al definir un trigger y/o programar un procedimiento almacenado tenemos en cuenta las siguientes características y reglas más importantes:

1. *El procedimiento almacenado debe de definirse e instalarse antes de definir el propio trigger.*
2. *Un procedimiento a utilizar por un trigger no tiene argumentos y devuelve el tipo "trigger".*
3. *Un mismo procedimiento almacenado se puede utilizar por múltiples triggers en diferentes tablas.*

[1] trigger o gatillo.

4. *Procedimientos almacenados utilizados por triggers que se ejecutan una sola vez por comando SQL (statement-level) siempre devuelven NULL.*

5. *Procedimientos almacenados utilizados por triggers que se ejecutan una vez por linea afectada por el comando SQL (row-level) pueden devolver una fila de tabla.*

6. *Procedimientos almacenados utilizados por triggers que se ejecutan una vez por fila afectada por el comando BEFORE (row-level) y ejecutar el comando SQL que lo lanzó, pueden:*

   a) *Retornar NULL para omitir la operación en la fila afectada.*

   b) *O devolver una fila de tabla (RECORD).*

7. *Procedimientos almacenados utilizados por triggers que se ejecutan AFTER de ejecutar el comando SQL que lo lanzó, ignoran el valor de retorno, así que pueden retornar NULL sin problemas.*

8. *En resumen, independientemente de como se defina un trigger, el procedimiento almacenado utilizado por dicho trigger tiene que devolver un NULL, o bien un valor RECORD con la misma estructura que la tabla que lanzó dicho trigger.*

9. *Si una tabla tiene más de un trigger definido para un mismo evento (INSERT, UPDATE, DELETE), estos se ejecutarán en orden alfabético por el nombre del trigger. En el caso de triggers del tipo BEFORE / row-level, la file retornada por cada trigger, se convierte en la entrada del siguiente. Si alguno de ellos retorna NULL, la operación será anulada para la fila afectada.*

10. *Procedimientos almacenados utilizados por triggers pueden ejecutar sentencias SQL que a su vez pueden activar otros triggers. Esto se conoce como triggers en cascada. No existe límite para el número de triggers que se pueden llamar pero es responsabilidad del programador el evitar una recursión infinita de llamadas en la que un trigger se llame asi mismo de manera recursiva.*

Otra cosa a tener en cuenta es que, por cada trigger que definamos en una tabla, nuestra base de datos tendrá que ejecutar la función asociada a dicho trigger. El uso de triggers de manera incorrecta o poco efectiva afecta al rendimiento de nuestra base de datos. Los novicios deben entender como funcionan los triggers y hacer un uso correcto de los mismos antes de usarlos en sistemas en producción.

9.2. Definiendo un trigger

Existen numerosas formas de definir un trigger; debido a las múltiples opciones disponibles para ello. Nos enfocamos en un subconjunto de características para iniciarnos en el tema. Definimos un trigger mínimo de esta forma:

```
CREATE TRIGGER trigger_name { BEFORE | AFTER | INSTEAD OF }
{ UPDATE | INSERT | DELETE | TRUNCATE }
ON table_name
FOR EACH ROW EXECUTE PROCEDURE function_name()
```

Las variables son: trigger_name; que es el nombre del trigger, table_name es el nombre de la tabla asociada al trigger, y function_name es el nombre de la función almacenada. En caso de duda sobre funciones consultamos el capítulo 6, página 71.

Consideremos la sentencia:

```
UPDATE account\_current SET balance = balance + 100 WHERE balance > 100000;
```

Ejecutando esta sentencia afectamos más de una fila. Si definimos un trigger a nivel de fila sobre la tabla, el trigger es activado por cada fila afectada. Pero con una sentencia, es activado una sola vez.

9.3. Etapas

Para instalar un trigger hacemos dos cosas:

1. *Crear una función para el trigger.*
2. *Instalar el trigger en una(s) tabla(s).*

Creamos la función:

```
CREATE OR REPLACE FUNCTION actualizar_nombre() RETURNS TRIGGER AS
   $trigger_ejemplo$
BEGIN
NEW.nombre := NEW.pat_pad || ' ' || NEW.mat_pad || ' ' || NEW.
   nom_pad ;
RETURN NEW;
END;
$trigger_ejemplo$ LANGUAGE plpgsql;
```

Luego, creamos el trigger:

```
CREATE TRIGGER trigger_ejemplo
BEFORE INSERT OR UPDATE ON 'tu_tabla'
FOR EACH ROW EXECUTE PROCEDURE actualizar_nombre();
```

9.4. Validación de datos

Cuando escribimos aplicaciones a nivel de código tenemos mensajes informativos sobre errores que "hacen" el trabajo de depuración sencillo, simple y rápido. Sin embargo, los mensajes de error en violación de restricción no son muy explícitos y requieren de investigación para comprender que significan. ¿Qué pasa si realizamos validación de datos y generamos

nuestros propios mensajes de error cuando no se cumplen los requerimientos?

Esto haría el proceso de desarrollo fácil. Realizamos validación de datos con un trigger y generamos error de violación de restricción con los mensajes apropiados. Verificamos que los datos cumplen con los requerimientos antes de insertarlos en la tabla de la base de datos o terminamos la operación evitando el error; sí los datos cumplen con los requerimientos la operación continua.

Asumamos que tenemos una tabla llamada passwd que gestiona a los usuarios en una aplicación, queremos asegurar que el password o clave de los usuarios no tengan menos de 10 caracteres o sea NULL, y que su nombre no es NULL.

```
1 CREATE TABLE passwd (
2 id serial primary key,
3 name text,
4 password text,
5 position text
6 );
```

Asociamos un trigger a la tabla passwd para operaciones INSERT y UPDATE, en vez de definir restricciones a nivel de base de datos. Primero, escribimos la función almacenada. En caso de duda sobre funciones, consultar Capítulo 6, página 71.

```
1  CREATE FUNCTION passwd_func() RETURNS TRIGGER
2  AS $$
3  BEGIN
4  IF LENGTH(NEW.password) < 10 OR NEW.password IS NULL THEN
5    RAISE EXCEPTION "La clave no puede tener menos de 10 caracteres
       ni ser NULL";
6  END IF;
7  IF NEW.NAME IS NULL THEN
8    RAISE EXCEPTION 'El nombre no puede ser NULL';
9  END IF;
10 RETURN NEW;
11 END;
12 $$
13 LANGUAGE plpgsql;
```

La función almacenada regresa un TRIGGER, esencial para todas las funciones almacenadas que serán usadas como procedimientos almacenados en los triggers. La función NO tiene argumentos. Ciertas variables especiales son pasadas a la función almacenada por defecto, veamos A, página 174.

Hay ciertos argumentos que se pasan a las funciones almacenadas por defecto.

Nuestra función utiliza NEW para comprobar si los valores individuales de la nueva fila cumplen con las restricciones, y si no lo hacen, hacemos una excepción con un mensaje adecuado. La función almacenada devuelve un NULL o bien una fila que coincida con la estructura de la tabla que la activo. En este caso, devolvemos NEW que contiene la nueva fila a insertar o actualizar. Es posible cambiar los valores individuales de NEW y devolverla

modificada para que se inserte en la base de datos. Además, utilizamos triggers para rellenar columnas en una operación de INSERT INTO; esto se hace cuando realizamos una búsqueda de texto completo, ya que necesitamos rellenar una columna de'tsvector'.

Finalmente, definimos el trigger principal desencadenante.

```
1 CREATE TRIGGER passwd_trigger BEFORE INSERT OR UPDATE
2 ON passwd
3 FOR EACH ROW EXECUTE PROCEDURE passwd_func();
```

9.5. Auditando los cambios

Supongamos que tenemos una aplicación bancaria, ¿cómo hacer un seguimiento de los cambios realizados en los datos subyacentes? ¿Cómo rastrearlos cuando un hacker y/o un error comprometen los datos? No querríamos adivinar cuál era el valor antes de los cambios maliciosos. Lo que necesitamos es una forma de asegurar que cualquier cambio realizado sea el correcto y se mantenga al día; una especie de historia del estado de cada fila en una tabla. Esto se hace a nivel de triggers desencadenantes. Supongamos que tenemos una aplicación bancaria. Creamos una tabla separada para hacer un seguimiento de los cambios realizados en la tabla de nuestra cuenta.

Creamos la tabla account:

```
1 CREATE TABLE account (
2 id serial primary key,
3 name text,
4 debt int,
5 balance int
6 );
```

Creamos la tabla account_audit:

```
1  CREATE TABLE account_audit(
2  id serial primary key,
3  db_user text NOT NULL default session_user,
4  operation text,
5  account_id int,
6  account_name text,
7  debt int,
8  balance int,
9  created_at timestamp with time zone default current_timestamp
10 );
```

La función almacenada para el trigger de auditoría es:

```
1 CREATE FUNCTION account_audit_func()
2 RETURNS TRIGGER
3 AS $$
4 BEGIN
```

```
5  IF TG_OP = 'INSERT' THEN
6  INSERT INTO account_audit (operation, account_id, account_name,
       debt, balance) VALUES
7  (TG_OP, NEW.*);
8  RETURN NEW;
9  ELSIF TG_OP = 'UPDATE' THEN
10 INSERT INTO account_audit (operation, account_id, account_name,
       debt, balance) VALUES
11 (TG_OP, NEW.*);
12 RETURN NEW;
13 ELSIF TG_OP = 'DELETE' THEN
14 INSERT INTO account_audit (operation, account_id, account_name,
       debt, balance) VALUES
15 (TG_OP, OLD.*);
16 RETURN OLD;
17 END IF;
18 END;
19 $$
20 LANGUAGE 'plpgsql';
```

La función devuelve un TRIGGER y comprueba TG_OP para saber qué operación activo el trigger. Si es un INSERT agregamos la nueva fila en la tabla account_audit, NEW contiene la nueva fila a insertar en la tabla y finalmente devolvemos NEW.

Hacemos lo mismo para una operación UPDATE. Perp para DELETE insertamos los valores de OLD, que contienen las filas que están a punto de ser borradas, y devolvemos OLD para continuar la operación de borrado.

Finalmente, la definición del trigger es:

```
1 CREATE TRIGGER account_audit_trigger
2 AFTER INSERT OR UPDATE OR DELETE ON account
3 FOR EACH ROW EXECUTE PROCEDURE account_audit_func();
```

La palabra AFTER significa que la operación que activó el trigger se completa antes que se active. Podríamos utilizar paquetes de datos foráneos para enviar los datos de nuestro trigger de auditoría a una base de datos remota para evitar la pérdida de datos en caso de un fallo.

9.6. Caso de uso

Tenemos tres tablas en nuestra base de datos: account_current, account_savings y log.

```
1 CREATE TABLE account_current (
2 customer_id integer NOT NULL,
3 customer_name character varying,
4 balance numeric,
5 CONSTRAINT account_current_pkey PRIMARY KEY (customer_id)
6 )
```

```
CREATE TABLE account_savings (
customer_id integer NOT NULL,
customer_name character varying,
balance numeric,
CONSTRAINT account_savings_pkey PRIMARY KEY (customer_id)
)
```

```
CREATE TABLE log (
log_id serial NOT NULL,
log_time time with time zone,
description character varying,
CONSTRAINT log_pkey PRIMARY KEY (log_id)
)
```

Creamos tres tipos de triggers, UPDATE, INSERT y DELETE.

Las dos primeras tablas almacenan el nombre del cliente con un id único y su saldo contable.

Podríamos tener una tabla usuario y una tabla detalle; por simplicidad, omitimos la normalización. La tercera tabla, log, es una tabla de auditoría que almacena cada acción o transacción junto a la fecha, hora y descripción de lo ocurrido.

Los cuatro tipos de acciones posibles son:

- *Un cliente apertura una cuenta de ahorro. Es realizado por una consulta INSERT.*
- *Un cliente retira/ deposita dinero de/ a su cuenta. Es realizado por una consulta UPDATE.*
- *Un cliente envía/recibe dinero de otro cliente. Es realizado por una combinación de dos consultas UPDATE.*
- *Un cliente cierra su cuenta. Es realizado por una consulta DELETE.*

Sí Bob apertura una cuenta de ahorro, deposita Bs.S. 2000 y envía 300 a Tom y finalmente decide cerrar su cuenta e ir a otro banco, tenemos las siguientes entradas en la tabla log:

27	10:02:13.19-07	New customer added. Account type: Savings, Customer ID: 1, Name: Bob, Balance: 2000
28	10:05:50.573-07	Balance updated. Account type: Savings, Customer ID: 1. Old balance: 2000, New balance: 1700
29	10:06:50.262-07	Balance updated. Account type: Savings, Customer ID: 3. Old balance: 1000, New balance: 1300
30	10:07:48.173-07	Account deleted. Account type: Savings, Customer ID: 1

Figura 9.1: Archivo log de entradas de cliente Bob.

PostgreSQL tiene interfaces cliente para Perl, Tcl, Python y C, así como dos Lenguajes Procedurales (PL). También es posible llamar a funciones escritas en lenguaje C como acciones trigger.

9.7. Teoría asociada a los triggers

*Si un evento trigger ocurre, el administrador de triggers (llamado ejecutor) inicializa la estructura global TriggerData *CurrentTriggerData llama a la función trigger para procesar el evento. La función, sin argumentos y con retorno OPAQUE, debe ser creada antes que el trigger.*

La sintaxis para la creación de un trigger es la siguiente:

```
1 CREATE TRIGGER <trigger name> <BEFORE|AFTER> <INSERT|DELETE|UPDATE
    >
2 ON <relation name> FOR EACH <ROW|STATEMENT>
3 EXECUTE PROCEDURE <procedure name> (<function args>);
```

El nombre del trigger se usa para eliminarlo. Se usa como argumento del comando DROP TRIGGER.

La palabra BEFORE o AFTER determina si la función es llamada antes o después del evento.

El elemento INSERT|DELETE|UPDATE comando determina en que evento es llamada la función. Es posible especificar múltiples eventos utilizado el operador OR.

El nombre de la relación (relation name) determina la tabla afectada por el evento.

La instrucción FOR EACH determina si el trigger se ejecuta para cada fila afectada o bien antes (o después) hasta que la secuencia se haya completado.

El nombre del procedimiento (procedure name) es el nombre de la función llamada.

Los argumentos son pasados a la función en la estructura CurrentTriggerData. El propósito de pasar los argumentos a la función es permitir a diferentes triggers con requisitos similares llamar a la misma función.

Además, la función puede ser utilizada para disparar distintas relaciones (estas funciones son llamadas "general trigger funcions").

Como un ejemplo de lo descrito, escribimos una función con dos argumentos, dos nombres de campo, e insertemos el nombre del usuario y la fecha (timestamp) actual en ellos. Esto permite, utilizar los triggers en los eventos INSERT para realizar un seguimiento automático de la creación de registros en una tabla de transacciones. También, se utiliza para registrar actualizaciones en un evento UPDATE.

Las funciones trigger regresan un bloque de tuplas (HeapTuple) al ejecutor. Esto es ignorado para un trigger disparado tras (AFTER) en una operación INSERT, DELETE o UPDATE, pero permite lo siguiente a los triggers BEFORE:

- *retornar NULL e ignorar la operación para la tupla actual (y de este modo la tupla no será insertada/actualizada/borrada);*
- *devolver un puntero a otra tupla (sólo en eventos INSERT y UPDATE) con datos que serán insertados (como la nueva versión de la tupla actualizada en caso de UPDATE) en lugar de la tupla original.*

Notemos que no hay inicialización por parte del manejador CREATE TRIGGER. Además, si más de un trigger es definido para el mismo evento en la misma relación, el orden de ejecución de los triggers es impredecible.

Si una función trigger ejecuta consultas SQL (utilizando SPI) entonces estas funciones activan nuevos triggers. Esto es conocido como triggers en cascada. No hay limitación explicita en cuanto al número de triggers en cascada.

Si un trigger es activado por un INSERT e inserta una nueva tupla en la misma relación, el trigger será llamado de nuevo (por el nuevo INSERT). Actualmente, no se proporcionan mecanismos de sincronización para estos casos; pero esto puede cambiar. Por el momento, existe la función llamada funny_dup17() en tests de regresión que utiliza algunas técnicas para detener la recursividad (cascada) por si misma.

9.8. Interacción con el Trigger Manager

*Como hemos mencionado, cuando una función es llamada por el administrador de triggers (trigger manager), la estructura TriggerData *CurrentTriggerData no es NULL y se inicializa. Por lo cual es mejor verificar que CurrentTriggerData no sea NULL al principio y asignarle NULL justo después de obtener la información para evitar llamadas a la función trigger que no procedan del administrador de triggers.*

La estructura TriggerData se define en src/include/commands/trigger.h veamos:

```
1  typedef struct TriggerData {
2    TriggerEvent
3    tg_event;
4    Relation
5    tg_relation;
6    HeapTuple
7    tg_trigtuple;
8    HeapTuple
9    tg_newtuple;
10   Trigger
11   *tg_trigger;
12 } TriggerData;
```

tg_event describe los eventos para los que la función es llamada. Puede utilizar las siguientes macros para examinar a tg_event: En la tabla 9.1 se muestra la definición de los miembros como sigue:

Tabla 9.1: Descripción de tg_event activado. Macros para examinarlos

Tipo	Siempre T_TriggerData.
tg_event	
TRIGGER_FIRED_BEFORE	verdad si el trigger se activa antes de la operación.
TRIGGER_FIRED_AFTER	verdad si el trigger se activa después de la operación.
TRIGGER_FIRED_FOR_ROW	verdad si el trigger se activa para un evento de bajo-nivel.
TRIGGER_FIRED_FOR_STATEMENT	verdad si el trigger se activa para un evento de nivel de sentencia.
TRIGGER_FIRED_BY_INSERT	verdad si el trigger se activa por un INSERT.
TRIGGER_FIRED_BY_UPDATE	verdad si el trigger se activa por un UPDATE.
TRIGGER_FIRED_BY_DELETE	verdad si el trigger se activa por un DELETE.
tg_relation	
tg_relation	puntero para la relación que activa el trigger.
otros	
tg_trigtuple	puntero a la fila INSERTED, UPDATED, o DELETED que activa el trigger. Si e activa para INSERT or DELETE; es lo que regresar de la función para no reemplazar la fila con una diferente (en INSERT) o bien, omitir la operación.
tg_newtuple	puntero a la nueva versión de la fila, si se activa con UPDATE, y es NULL con INSERT o DELETE. Es lo que regresa de la función, en UPDATE, para no reemplaza la fila por una diferente, o bien omitimos la operación.
tg_trigger	puntero a una estructura de tipo Trigger, definida en utils/rel.h, ver Apéndice E, página 181.
tg_trigtuplebuf	buffer que contiene a tg_trigtuple, o InvalidBuffer si no hay una tupla o si no es almacenada en el disco.
tg_newtuplebuf	buffer que contiene a

```
typedef struct Trigger {
  Oid
  char
  Oid
  FmgrInfo
  int16
  bool
  bool
  bool
  bool
  int16
  int16
  char
} Trigger;
tgoid;
*tgname;
tgfoid;
tgfunc;
tgtype;
tgenabled;
tgisconstraint;
tgdeferrable;
tginitdeferred;
tgnargs;
tgattr[FUNC_MAX_ARGS];
**tgargs;
```

tg_name es el nombre del trigger, tg_nargs es el número de argumentos, tg_args es un array de punteros a los argumentos especificados en el CREATE TRIGGER. Los otros miembros son exclusivamente de uso interno.

10

Paquetes

No hay paquetes en PLpg/SQL, pero sí podemos colocar las funciones, cursores, etc. bajo un SCHEMA adicional y así simular su funcionamiento.

Los paquetes son la forma que Oracle proporciona para encapsular sentencias, procedimientos, comandos y funciones PL/SQL en una entidad, tal como las clases de Java, donde definimos métodos y objetos. Accedemos a estos objetos/métodos con un punto "." (dot).

Veamos un código de paquete Oracle de ACS 4 (ArsDigital Community System):

```
1  CREATE OR REPLACE PACKAGE BODY acs
2  AS
3  FUNCTION add_user (
4  user_id
5  IN users.user_id%TYPE DEFAULT NULL,
6  object_type
7  IN acs_objects.object_type%TYPE DEFAULT 'user',
8  creation_date IN acs_objects.creation_date%TYPE DEFAULT sysdate,
9  creation_user IN acs_objects.creation_user%TYPE DEFAULT NULL,
10 creation_ip
11 IN acs_objects.creation_ip%TYPE DEFAULT NULL,
12 ...
13 ) RETURN users.user_id%TYPE
14 IS
15 v_user_id
16 users.user_id%TYPE;
17 v_rel_id
18 membership_rels.rel_id%TYPE;
19 BEGIN
20 v_user_id := acs_user.new (user_id, object_type, creation_date,
21 creation_user, creation_ip, email, ...
22 RETURN v_user_id;
23 END;
24 END acs;
```

```
25 /
26 show errors
```

Migraremos a PostgreSQL creando los diferentes objetos del paquete Oracle como funciones con una convención de nombres estándar. Atendemos a algunos detalles, tales como la falta de parámetros por defecto en las funciones PL/pgSQL. El código anterior es como esto:

```
1  CREATE FUNCTION acs__add_user(INTEGER,INTEGER,VARCHAR,TIMESTAMP,
       INTEGER,INTEGER,...)
2  RETURNS INTEGER AS '
3  DECLARE
4  user_id ALIAS FOR $1;
5  object_type ALIAS FOR $2;
6  creation_date ALIAS FOR $3;
7  creation_user ALIAS FOR $4;
8  creation_ip ALIAS FOR $5;
9  ...
10 v_user_id users.user_id%TYPE;
11 v_rel_id membership_rels.rel_id%TYPE;
12 BEGIN
13 v_user_id
14 := acs_user__new(user_id,object_type,creation_date,creation_user,
       creation_ip, ...);
15 ...
16 RETURN v_user_id;
17 END;
18 ' LANGUAGE 'plpgsql';
```

> ***NOTA:*** *No existe mucha literatura disponible sobre migrar paquetes de Oracle a PL/pgSQL, menos sobre los errores en la migración.*

10.1. CREATE SCHEMA

Utilicemos CREATE SCHEMA para crear un nuevo esquema y sus objetos.

La sentencia CREATE SCHEMA crea el nuevo esquema en la base de datos actual. El siguiente código muestra su sintaxis:

```
1 CREATE SCHEMA [IF NOT EXISTS] schema_name;
```

En la sintaxis, tenemos que:

- *especificar el nombre del esquema después de CREATE SCHEMA. Dicho nombre debe ser único en la base de datos actual.*
- *hacer uso de la opción IF NOT EXISTS para condicionar la creación del nuevo esquema a hacerlo únicamente si no existe. Intentar crear un nuevo esquema que existe sin usar IF NOT EXISTS genera un error.*

Notemos que para ejecutar CREATE SCHEMA, debemos tener privilegio para CREATE en la base de datos actual.

También, podemos crear un esquema para un usuario:

```
CREATE SCHEMA [IF NOT EXISTS] AUTHORIZATION user_name;
```

En este caso, el esquema tiene el mismo nombre que user_name; PostgreSQL permite crear un esquema y una lista de objetos tales como tablas y vistas en una única sentencia, como sigue:

```
CREATE SCHEMA schema_name

CREATE TABLE table_name1 (...)

CREATE TABLE table_name2 (...)

CREATE VIEW view_name1

SELECT select_list FROM table_name1;
```

Notemos que cada sentencias no termina en punto y coma (;).

10.1.1. Códigos

Veamos algunos códigos que utilizan CREATE SCHEMA para una mejor comprensión.

La siguiente sentencia CREATE SCHEMA crea un nuevo esquema llamado marketing:

```
CREATE SCHEMA IF NOT EXIST marketing;
```

La siguiente sentencia regresa todos los esquemas de la base de datos actual:

```
SELECT * FROM pg_catalog.pg_namespace ORDER BY nspname;
```

Usamos CREATE SCHEMA para crear un esquema para un usuario.

Primero, creamos un nuevo usuario llamado dianella:

```
CREATE USER dianella WITH ENCRYPTED PASSWORD 'Postgr@s321!';
```

Segundo, creamos el esquema para dianella:

```
CREATE SCHEMA AUTHORIZATION dianella;
```

Tercero, creamos un nuevo esquema propiedad de dianella:

```
CREATE SCHEMA IF NOT EXISTS doe AUTHORIZATION dianella;
```

Los siguientes códigos muestran el uso de CREATE SCHEMA para crear un nuevo esquema llamado dasd. También creamos la tabla llamada deliveries y una vista llamada delivery_due_list que pertenecen al esquema dasd:

```
CREATE SCHEMA dasd

CREATE TABLE deliveries( id SERIAL NOT NULL,
customer_id INT NOT NULL,
ship_date DATE NOT NULL
);

CREATE VIEW delivery_due_list AS
SELECT ID, ship_date FROM deliveries
WHERE ship_date <= CURRENT_DATE;
```

11

Otros lenguajes procedurales

Algunos de los lenguajes procedurales que se pueden usar en PostgreSQL son los siguientes:

1. *Un lenguaje propio llamado PL/PgSQL (similar al PL/SQL de oracle).*
2. *C.*
3. *C++.*
4. *Java PL/Java web.*
5. *PL/Perl.*
6. *plPHP.*
7. *PL/Python.*
8. *PL/Ruby.*
9. *PL/sh.*
10. *PL/Tcl.*
11. *PL/Scheme.*
12. *Lenguaje para aplicaciones estadísticas R por medio de PL/R.*

Con ellos, escribimos funciones para PL/pgSQL, como escribir un código en Python dentro de una función. Comúnmente se escribe una "u", de untrusted[1] *como sufijo del lenguaje para su identificación, es el caso de PL/Perlu o PL/Pythonu.*

En este capítulo se trabaja en los lenguajes procedurales de desconfianza PL/Tcl, PL/Python y PL/R.

[1] desconfianza

11.1. lenguaje C

No deberíamos escribir código C en funciones PL/pgSQL porque tendríamos acceso al interior del PL/pgSQL y de PostgreSQL, lo que significa peligro latente.

Esta sección describe los detalles a bajo-nivel de la interfase función trigger. Esta información sólo es necesaria cuando escribimos funciones C para triggers. Si usamos un lenguaje de alto-nivel, los detalles son gestionados para él. En la mayoría de los casos debemos considerar usar un lenguaje procedural antes de usar C en un trigger.

La función trigger debe usar el gestor de interfase en la "versión 1".

Cuando una función es llamada por el manejador de trigger, no recibe argumentos normales, sino un puntero de "context" a una estructura EventTriggerData. Las funciones en C pueden discriminar si son llamadas desde el manejador de trigger y no por la macro ejecutante:

```
CALLED_AS_EVENT_TRIGGER(fcinfo)
```

la cual se expande como:

```
((fcinfo)->context != NULL && IsA((fcinfo)->context,
   EventTriggerData))
```

Si el código previo regresa verdad (true), es seguro interpretar `fcinfo->context` *en tipo* `EventTriggerData *` *y usar la estructura de puntero EventTriggerData. La función no debe cambiar dicha estructura o cualquiera de sus datos.*

Un struct EventTriggerData es definido en commands/event_trigger.h:

```
typedef struct EventTriggerData {
  NodeTag     type;
  const char *event;      /* event name */
  Node       *parsetree;  /* parse tree */
  const char *tag;        /* command tag */
} EventTriggerData;
```

En caso de duda, ver la tabla 9.1, página 120.

Un función de un trigger debe devolver un HeapTuple pointer o un NULL. Atentos al resultado de un tg_trigtuple o tg_newtuple, si no queremos modificar la fila sobre la que estamos operando.

```
\label{cod:0001}
typedef struct Trigger {
Oid         tgoid;
char       *tgname;
Oid         tgfoid;
int16       tgtype;
bool        tgenabled;
bool        tgisconstraint;
Oid         tgconstrrelid;
bool        tgdeferrable;
```

```
11 bool         tginitdeferred;
12 int16        tgnargs;
13 int16        tgnattr;
14 int16       *tgattr;
15 char       **tgargs;
16 } Trigger;
```

11.2. PL/Tcl

PL/Tcl es un lenguaje procedural para PostgreSQL, usamos Tcl para la creación de funciones y procedimientos desencadenados por eventos. Este paquete fue escrito originalmente por Jan Wieck.

11.2.1. Introducción

PL/Tcl ofrece la mayoría de las características del lenguaje C con algunas restricciones. Las cuales son buenas porque todo se ejecuta en un interprete Tcl. Además del reducido juego de órdenes de Tcl, sólo se disponen de unas pocas de ellas para acceder a bases de datos a través de SPI y para enviar mensajes mediante elog(). No hay forma de acceder al interior del proceso de gestión de la base de datos, ni obtener acceso al nivel del sistema operativo, bajo los permisos del identificador de usuario de PostgreSQL, como es posible en C. Así que un usuario de la bases de datos sin privilegios puede usar este lenguaje.

La otra restricción, interna, es que los procedimientos Tcl no pueden usarse para crear funciones de entrada / salida para nuevos tipos de datos.

Los objetos compartidos por el gestor de llamada PL/Tcl se construyen automáticamente y se instalan en el directorio de bibliotecas de PostgreSQL, sólo si el soporte de Tcl/Tk ha sido especificado durante la configuración en el procedimiento de instalación.

11.2.2. Funciones de PL/pgSQL y nombres de procedimientos Tcl

En PL/pgSQL, un mismo nombre de función puede usarse para diferentes funciones, siempre que el número de argumentos o sus tipos sean distintos. Esto puede ocasionar conflictos con los nombres de procedimientos Tcl.

Para ofrecer la misma flexibilidad en PL/Tcl, los nombres de procedimientos Tcl internos contienen el identificador de objeto de la fila de procedimientos pg_proc como parte de sus nombres. Así, diferentes versiones (por el número de argumentos) de una misma función de PL/pgSQL pueden ser diferentes también para Tcl.

11.2.3. Definiendo funciones en PL/Tcl

Para crear una función en el lenguaje PL/Tcl, usamos la sintaxis:

```
CREATE FUNCTION funcname (argumen) RETURNS
RETURNTYPE AS '
# PL/Tcl function body
' LANGUAGE 'pltcl';
```

Cuando invocamos esta función en una consulta, los argumentos se dan como variables `$1` *... * `$n` *en el cuerpo del procedimiento Tcl. Así, una función de máximo que devuelva el mayor de dos valores int4 es creada del siguiente modo:*

```
CREATE FUNCTION tcl_max (int4, int4) RETURNS int4 AS '
if {$1 > $2} {RETURN $1}
RETURN $2
' LANGUAGE 'pltcl';
```

Pasamos argumentos de tipo compuesto al procedimiento como matrices de Tcl. Los nombres de elementos en la matriz son los nombres de los atributos del tipo compuesto.

Si un atributo de la fila actual tiene el valor NULL, no aparece en la matriz.

Veamos un código que define la función overpaid_2, escrita en PL/Tcl.

```
CREATE FUNCTION overpaid_2 (EMP) RETURNS bool AS '
IF {200000.0 < $1(salary)} {
RETURN "t"
}
IF {$1(age) < 30 && 100000.0 < $1(salary)} {
RETURN "t"
}
RETURN "f"
' LANGUAGE 'pltcl';
```

11.2.4. Datos Globales en PL/Tcl

Cuando usamos las funciones SPI, es útil mantener algunos datos globales entre dos llamadas al procedimiento. Los procedimientos PL/Tcl que son ejecutados por un BACKEND comparten el mismo interprete de Tcl. Para proteger a los procedimientos PL/Tcl de efectos secundarios, disponemos de una matriz para cada uno de los procedimientos a través de la orden "upvar". El nombre global de esa variable es el nombre interno asignado por el procedimiento, y el nombre local es GD.

11.2.5. Procedimientos desencadenados en PL/Tcl

Los procedimientos desencadenados se definen como funciones sin argumento, que devuelven un tipo OPAQUE. Lo mismo ocurre en el lenguaje PL/Tcl.

La información del gestor de procedimientos desencadenados se pasa al cuerpo del procedimiento en las siguientes variables:

Tabla 11.1: Variables de Pl/tcl para PL/pgSQL

Variable	Descripción
`$TG_name`	El nombre del procedimiento trigger se toma de la sentencia CREATE TRIGGER.
`$TG_relid`	El ID de objeto de la tabla que provoca el desencadenamiento ha de ser invocado.
`$TG_relatts`	Una lista de los nombres de campos de las tablas, precedida de un elemento de lista vacío. Esto se hace para que al buscar un nombre de elemento en la lista con lsearch, se devuelva el mismo número positivo, comenzando por 1, en el que los campos están numerados en el catalogo de sistema. `'pg_attribute'`.
`$TG_when`	La cadena BEFORE o AFTER, dependiendo del suceso de la llamada desencadenante.
`$TG_level`	La cadena ROW o STATEMENT, dependiendo del suceso de la llamada desencadenante.
`$TG_op`	La cadena INSERT, UPDATE o DELETE, dependiendo del suceso de la llamada desencadenante.
`$NEW`	Una matriz que contiene los valores de la fila de la nueva tabla para acciones INSERT/UPDATE, o vacía para DELETE.
`$OLD`	Una matriz que contiene los valores de la fila de la vieja tabla para acciones UPDATE o DELETE, o vacía para INSERT.
`$GD`	La matriz de datos de estado global, como se describa más adelante.
`$args`	Es una lista de los argumentos del procedimiento como se dan en la sentencia CREATE TRIGGER. Los argumentos son también accesibles como `$1 ... $n` en el cuerpo del procedimiento.

EL valor devuelto por un procedimiento desencadenado es una de las cadenas OK o SKIP, o bien una lista devuelta por array get. Si el valor devuelto es OK, la operación normal desencadenante del procedimiento (INSERT/UPDATE/DELETE) tiene lugar. Obviamente, SKIP informa al gestor de procesos desencadenados para que suprima la operación. La lista array get notifica a PL/Tcl que devuelva una fila modificada al gestor de procedimientos desencadenados que será insertada en lugar de la actual en $NEW (sólo para INSERT/UPDATE). Esto NO tiene sentido cuando el proceso desencadenante es BEFORE y FOR EACH ROW.

Veamos un código de un procedimiento desencadenado que fuerza a un valor entero de una

tabla a seguir la traza del número de actualizaciones realizados en ella. Para cada nueva fila insertada, el valor es inicializado a 0, e incrementado por cada operación de actualización: Creamos la tabla;

```
CREATE TABLE mytab (num int4, modcnt int4, desc text);
```

Creamos la función;

```
CREATE FUNCTION trigfunc_modcount() RETURNS OPAQUE AS '
switch $TG_op {
  INSERT {
    set NEW($1) 0
  }
  UPDATE {
    set NEW($1) $OLD($1)
    incr NEW($1)
  }
  default {
    RETURN OK
  }
}
RETURN [array get NEW]
' LANGUAGE 'pltcl';
```

Creamos el trigger;

```
CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON
   mytab
FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

11.2.6. Acceso a bases de datos desde PL/Tcl

Las siguientes órdenes acceden a una base de datos desde el interior de un procedimiento PL/Tcl:

Tabla 11.2: Tabla con las variables de Pl/tcl para PostgreSQL

Variable	Descripción
`elog level msg`	Genera un mensaje de registro. Los posibles niveles son NOTICE, WARN, ERROR, FATAL, DEBUG y NOIND, como en la función elog() de C.
`quote string`	Duplica todas las apariciones de una comilla o de la barra invertida. Debe usarse cuando las variables se usen en la cadena de la consulta enviada a spi_exec o spi_prepara (no en la lista de valores de spi_execp).

Consideremos la siguiente cadena de consulta:

`"SELECT '$val'AS ret"`

Donde la variable 'val'contiene "no hace". Esto da lugar a la cadena de consulta.

`"SELECT 'no hace'AS ret"`

que produce un error del analizador durante la ejecución de spi_exec o spi_prepare. Debe contener:

`"SELECT 'no hace'AS ret"` *y escribirse de la siguiente manera:*

```
1 "SELECT '[ quote $val ]' AS ret"
2 spi_exec ?-count n? ?-array nam?que ?loop-body?
```

Llama al analizador/planificador/optimizador/ejecutor de la consulta. El valor opcional -count informa a spi_exec el máximo número de filas a procesar por la consulta.

Si la consulta es una sentencia SELECT y se incluye el cuerpo del ciclo opcional (grupo de sentencias Tcl similar a una sentencia anticipada), se evalúa para cada fila seleccionada, y se comporta como se espera, tras continue/break. Los valores de los campos seleccionados se colocan en nombres de variables, como nombres de columnas. Así, `spi_exec "SELECT count (*)AS cnt FROM pg_proc"` *asigna a la variable cnt el número de filas en el catálogo de sistema pg_proc. Si se incluye la opción -array, los valores de las columnas son almacenados en la matriz asociativa llamada name, indexada por el nombre de la columna, en lugar de por variables individuales.*

```
1 spi_exec -array C "SELECT * FROM pg_class" {
2   elog DEBUG "tengo tabla $C(relname)"
3 }
```

El código anterior imprime un mensaje de registro DEBUG para cada una de las filas de pg_class.

El valor devuelto por spi_exec es el número de filas afectado por la consulta, y se encuentra en la variable global spi_processed. Mientras que spi_prepare prepara y almacena una consulta para una ejecución posterior.

Es un distinto del caso de C, ya que, la consulta prevista es copiada en el contexto de memoria de mayor nivel. Por lo tanto, no existe forma de planificar una consulta sin guardarla.

Si la consulta hace referencia a argumentos, debemos tener los nombres de sus tipos en forma de lista. El valor devuelto por spi_prepare es el identificador de la consulta que se usará en las siguientes llamadas a spi_execp.

Veamos a spi_execp en el siguiente código.

```
spi_exec ?-count n? ?-array nam? ?-nullsesquvalue? ?loop-body?
```

El cual ejecuta una consulta preparada en spi_prepare con sustitución de variables. El valor, opcional, -count informa a spi_execp el número de filas a procesar en la consulta.

El valor para -nulls es una cadena de espacios de longitud n, que notifica a spi_execp que

valores son NULL. Si se indica, debe tener exactamente la longitud del número de valores.

El identificador de la consulta es devuelto por la llamada a spi_prepare.

Si se pasa una lista a spi_prepare, debe ser de la misma longitud a spi_execp después de la consulta. Si la lista spi_prepare está vacía, el argumento puede omitirse.

Si la consulta es una sentencia SELECT, lo que se ha descrito para spi_exec ocurre para el cuerpo del ciclo y las variables de los campos seleccionados.

Veamos el código de una función que usa una consulta planificada:

```
1  CREATE FUNCTION t1_count(int4, int4) RETURNS int4 AS '
2  if {![ info exists GD(plan) ]} {
3  # prepare the saved plan on the first call
4  set GD(plan) [ spi_prepare \\
5  "SELECT count(*) AS cnt FROM t1 WHERE num >= \\$1 AND
6  int4 ]
7  }
8  spi_execp -count 1 $GD(plan) [ list $1 $2 ]
9  RETURN $cnt
10 ' LANGUAGE 'pltcl';
```

Notemos que cada una de las barras invertidas[2] debe ser duplicada en la consulta que crea la función, dado que el analizador principal procesa estas barras en `CREATE FUNCTION`*. En la cadena de la consulta que se pasa a spi_prepare debe haber un signo $ para marcar la posición del parámetro, y evitar que $1 sea sustituido por el valor dado en la primera llamada a la función.*

11.2.7. Módulos y la orden *desconocido*

PL/Tcl tiene una característica especial para cosas que suceden raramente.

Reconoce dos tablas "mágicas", pltcl_modules y pltcl_modfuncs. Si existen, el módulo "desconocido" es cargado por el interprete, tras su creación. Cada vez que se invoca un procedimiento desconocido, es comprobado, por si está definido en uno de esos módulos. Si ocurre, el módulo es cargado cuando sea necesario. Para habilitar este comportamiento, el gestor de llamadas de PL/Tcl debe ser compilado con la opción `DPLTCL_UNKNOWN_SUPPORT` *habilitado.*

Existen scripts[3] de soporte para mantener esas tablas en el subdirectorio de módulos del código fuente de PL/Tcl, incluyendo el código fuente del módulo "desconocido", que debió ser instalado inicialmente.

[2] código de escape
[3] guión

11.3. PL/Python

11.3.1. Introducción

PL/Python fue introducido en la versión 7.2 de PostgreSQL por Andrew Bosma en el año 2002 y mejorado en cada versión del gestor. El mismo posibilita escribir funciones en lenguaje Python para PostgreSQL.

Python es un lenguaje simple y fácil de aprender, posee múltiples bibliotecas para variadas actividades, ya sea en sistemas de gestión comercial, interacciones con el sistema operativo, etc.

Para la instalación de PL/Python a partir de 9.1 en adelante que se creó el mecanismo de extensiones, este lenguaje se instala como una extensión con `CREATE EXTENSION plpythonu`.

También, se instala desde la línea de comandos con CREATELANG plpythonu nombre_basededatos.

Para el uso correcto de PL/Python en Debian/Ubuntu el sistema debe tener instalado el paquete postgresql-plpython-9.1 o superior.

11.3.2. Funciones en PL/Python

Una función en PL/Python se crea de igual forma que el resto de las funciones en PostgreSQL, haciendo uso del comando CREATE FUNCTION. El cuerpo de la función es código Python. El retorno de variables desde Python es RETURN. También, puede ser empleado YIELD y, si no se provee una salida, devuelve NONE que se traduce como null.

El código siguiente muestra la función para devolver la cadena "Hola Mundo".

```
1 CREATE FUNCTION holamundo() RETURNS text AS
2 $$
3 RETURN "Hola Mundo"
4 $$
5 languaje 'plpythonu';
```

Invocación de la función:

```
1 debian=# SELECT holamundo();
2 holamundo
3 Hola Mundo
4 (1 fila)
```

Notemos en el código anterior que, al igual que las funciones en SQL y PL/pgSQL, a la función se le debe especificar el lenguaje en que está siendo creada, en el caso de Python plpythonu.

11.3.3. Parámetros de una función en PL/Python

Los parámetros de una función PL/Python se pasan como en cualquier otro lenguaje procedural y, una vez dentro de la función, deben ser llamados por su nombre. El código

siguiente muestra esto:

```
1 CREATE FUNCTION hola(nombre text) RETURNS text AS
2 $$
3 RETURN "Hola %s" % nombre
4 $$
5 LANGUAGE 'plpythonu';
```

Invocación de la función:

```
1 debian=# SELECT hola("Antonio");
2 hola
3 Hola Antonio
4 (1 fila)
```

Al igual que en PL/pgSQL pueden definirse parámetros de salida. El código a continuación muestra su empleo.

```
1 CREATE OR REPLACE FUNCTION saludo(
2 INOUT nombre text, OUT mayuscula text) AS
3 $$
4 mayuscula = "HOLA %s" % nombre.upper()
5 RETURN ("Hola "+ nombre, mayuscula)
6 $$
7 languaje 'plpythonu';
```

Llamado de la función:

```
1 debian=# SELECT saludo("Dianella");
2 saludo
3 Hola Dianella, HOLA DIANELLA
4 (1 fila)
```

11.3.4. Pasando tipos compuestos como parámetros

Los tipos de datos compuestos de PostgreSQL pueden pasarse como parámetro a PL/Python y son automáticamente convertidos en diccionarios dentro de Python. A continuación un código de cómo se utilizan:

```
1  -- Creación de un tipo de dato compuesto
2  CREATE TYPE mini_prod AS (nombre varchar, precio numeric);
3  -- Función que chequea que determinado producto inice o no con A
4  CREATE OR REPLACE FUNCTION parametros_mi_tipo(tabla mini_prod)
5  RETURNS character varying AS
6  $$
7  IF "A" in tabla["nombre"][0]:
8  RETURN "Comienza con A"
9  ELSE:
10 RETURN "No Comienza con A"
11 $$
```

```
12 languaje 'plpythonu';
```

Invocación de la función:

```
1 debian=# SELECT parametros_mi_tipo(ROW("Braveheart", 1.10));
2 parametros_mi_tipo
3 No comienza con A
4 (1 fila)
```

11.3.5. Pasando arreglos como parámetros

También se pueden pasar como parámetros arreglos de PostgreSQL, los cuales son convertidos a listas o tuplas. El código siguiente muestra cómo emplearlos:

```
1 -- Función que devuelve el primer elemento de un arreglo pasado
2 por parámetro
3 CREATE OR REPLACE FUNCTION arreglos(a character varying[])
4 RETURNS character
5 VARIYING AS
6 $$
7 RETURN a[0]
8 $$
9 languaje 'plpythonu';
```

– *Invocación de la función*

```
1 debian=# SELECT arreglos(array["Roberto","Chávez"]);
2 arreglos
3 Roberto
4 (1 fila)
```

11.3.6. Homologación de tipos de datos PL/Python

La homologación de tipos de datos de PostgreSQL a PL/Python se realiza como muestra la tabla siguiente.

Tabla 11.3: Homologación de tipos de PostgreSQL a PL/Python

PostgreSQL	Python 2	Python 3
text, char, varchar	str	str
boolean	bool	bool
real, numeric, double	float	float
smallint, int	int	int
bigint, oid	long	int
null	none	none

11.3.7. Retorno de valores de una función en PL/Python

En apartados anteriores se ha mostrado cómo pasar parámetros y cómo se utiliza la sentencia RETURN.

11.3.8. Devolviendo arreglos

En esta sección se analizan algunas particularidades del retorno de valores en PL/Python. Devolver una lista o tupla en Python es similar a un arreglo en PostgreSQL; el código a continuación lo muestra.

```
1 -- Función que retorna un arreglo
2 CREATE FUNCTION retorna_arreglo_texto() RETURNS text[] AS
3 $$
4 RETURN ("Hola","Mundo")
5 $$ languaje 'plpythonu';
```

Invocación de la función

```
1 debian=# SELECT retorna_arreglo_texto();
2 retorna_arreglo_texto
3 {Hola,Mundo}
4 (1 fila)
```

11.3.9. Devolviendo tipos compuestos

Devolver tipos compuestos se realiza de tres modos:

- *Tupla o lista: que tiene la misma cantidad y tipos de datos que el compuesto definido por el usurario.*
- *Diccionario: que tiene la misma la cantidad y tipos de datos que el tipo compuesto definido, además, los nombres de las columnas deben coincidir.*
- *Objeto: donde la definición de la clase tiene los atributos similares al del tipo de datos compuesto por el usuario.*

Devolviendo valores de tipo compuesto como una tupla:

```
1 -- Función que retorna un dato compuesto como tupla
2 CREATE FUNCTION retorna_tipo_lista() RETURNS mini_prod AS
3 $$
4 nombre = "Un jugute para David"
5 precio = 2.80
6 RETURN (nombre, precio) # como tupla
7 $$ languaje 'plpythonu';
```

Invocación de la función

```
debian=# SELECT retorna_tipo_lista();
retorna_tipo_lista
("Un juguete para David", 2.8)
(1 fila)
```

Notamos que en éste, y en los siguientes códigos, se emplea el tipo de dato compuesto mini_prod previamente creado.

También puede regresarse como en la forma RETURN [nombre, precio] para hacerlo una lista:

```
-- Crear tipo de dato compuesto
CREATE TYPE mini_prod AS (nombre varchar, precio numeric);
-- Función que retorna un dato compuesto como diccionario
CREATE FUNCTION retorna_tipo_dic() RETURNS mini_prod AS
$$
nombre = "Un juguete para David"
precio = 2.80
RETURN { "nombre": nombre, "precio": precio }
$$ languaje 'plpythonu';
```

Invocación de la función:

```
debian=# SELECT retorna_tipo_dic();
retorna_tipo_dic
("Un juguete para David", 2.8)
(1 fila)
```

11.3.10. Retornando conjuntos de resultados

Para devolver conjuntos de datos puede utilizarse:

- *Una lista o tupla.*
- *Un generador (yield).*
- *Un iterador.*

Devolviendo conjunto de valores como una lista o tupla:

```
-- Crear tipo de dato compuesto
CREATE TYPE mini_prod AS (nombre varchar, precio numeric);
-- Función que retorna un conjunto de datos como lista o tupla
CREATE FUNCTION mini_producto_conjunto() RETURNS SETOF mini_prod
   AS
$$
nombre = "Hola Hemingway"
precio = 4.31
RETURN ( [ nombre, precio ], [ nombre + nombre, precio + precio ]
   )
$$ languaje 'plpythonu';
```

Invocación de la función:

```
debian=# SELECT * FROM mini_producto_conjunto();
nombre | precio
Hola Hemingway | 4.31
Hola HemingwayHola Hemingway | 8.62
(2 filas)
```

Devolviendo conjunto de valores como un generador:

```
-- Crear tipo de dato compuesto
CREATE TYPE mini_prod AS (nombre varchar, precio numeric);
-- Función que retorna un conjunto de datos como un generador
CREATE FUNCTION mini_producto_conjunto_generador()
RETURNS SETOF record AS
$$
lista=[('Clandestinos', 5.00), ('Memorias del subdesarrollo',
   5.10),
("Suite Habana", 3.90)]
FOR producto IN lista:
YIELD ( producto[0], producto[1] )
$$ languaje 'plpythonu';
```

Invocación de la función:

```
debian=# SELECT * FROM mini_producto_conjunto_generador()
AS (a text, b numeric);
a | b
Clandestinos | 5.00
Memorias del subdesarrollo | 5.10
Suite Habana | 3.90
(3 filas)
```

11.3.11. Ejecutando consultas en la función PL/Python

PL/Python realiza consultas sobre la base de datos a través de un módulo llamado plpy, importado por defecto. Tiene varias funciones importantes como son:

- *plpy.execute(consulta [, max-rows]): ejecuta una consulta con una cantidad finita de tuplas en el resultado, especificado como segundo parámetro (opcional). Devuelve un objeto similar a una lista de diccionarios, al que se accede con la forma resultado[0]["micolumna"], para el primer registro y a la columna "micolumna".*
- *plpy.prepare(consulta [, argtypes]): prepara planes de ejecución para determinadas consultas para luego ser ejecutadas por la función execute(plan [, arguments [, max-rows]]).*
- *plpy.cursor(consulta): añadido a partir de la versión 9.2 del gestor.*

Devolviendo valores de la ejecución de una consulta desde PL/Python con plpy.execute:

```
-- Función que retorna los 10 primeros productos haciendo uso de plpy.execute
CREATE FUNCTION obtener_productos() RETURNS TABLE (id int, titulo text, precio
numeric) AS
$$
resultado = plpy.execute("SELECT * FROM products LIMIT 10")
FOR tupla IN resultado:
YIELD (tupla['prod_id'], tupla['title'], tupla['price'])
$$
languaje 'plpythonu';
```

Invocación de la función:

```
debian=# SELECT * FROM obtener_productos();
id | titulo | precio
1 | ACADEMY ACADEMY| 25.99
2 | ACADEMY ACE | 20.99
3 | ACADEMY ADAPTATION | 28.99
4 | ACADEMY AFFAIR | 14.99
5 | ACADEMY AFRICAN | 11.99
6 | ACADEMY AGENT | 15.99
7 | ACADEMY AIRPLANE | 25.99
8 | ACADEMY AIRPORT | 16.99
9 | ACADEMY ALABAMA | 10.99
10 | ACADEMY ALADDIN | 9.99
(10 filas)
```

Ejecución con plpy.execute de una consulta preparada con plpy.prepare:

```
CREATE FUNCTION preparada() RETURNS text AS
$$
miplan = plpy.prepare("SELECT * FROM products WHERE prod_id = $1",
    ["int"])
resultado = plpy.execute(miplan, [4])
RETURN resultado[0]['title']
$$
languaje 'plpythonu';
```

Invocación de la función:

```
debian=# SELECT * FROM preparada();
preparada
ACADEMY AFFAIR
(1 fila)
```

PL/Python cuenta con otras funciones útiles, como plpy.debug(msg), plpy.log(msg), plpy.info(msg), plpy.notice(msg), plpy.warning(msg), plpy.error(msg).

11.3.12. Mezclando

Como ha sido demostrado, los lenguajes de desconfianza son utilizados para hacer rutinas externas al servidor de bases de datos, a continuación se muestra un código de cómo guardar en un archivo XML el resultado de una consulta.

Guardar en un XML el resultado de una consulta:

```
1  CREATE OR REPLACE FUNCTION salva_tabla_xml() RETURNS character
       varying AS
2  $$
3  from xml.etree import ElementTree
4  rv = plpy.execute("SELECT * FROM categories")
5  raiz = ElementTree.Element('consulta')
6  for valor in rv:
7  datos = ElementTree.SubElement(raiz,'datos')
8  for key, value in valor.items():
9  element = ElementTree.SubElement(datos, key)
10 element.text = str(value)
11 contenido = ElementTree.tostring(raiz)
12 archivo = open('/tmp/archivo.xml','w')
13 archivo.write(contenido)
14 archivo.close()
15 RETURN contenido
16 $$
17 languaje 'plpythonu';
```

Notamos que para que esta función se ejecute la ruta especificada en archivo debe existir.

11.3.13. Realizando triggers con PL/Python

PL/Python escribe funciones activadoras, para eso cuenta con un diccionario TD, que posee varios pares clave/valor útiles para su trabajo; algunos se describen a continuación:

- *TD[.event"]: almacena como un texto el evento que se está ejecutando (INSERT, UPDATE, DELETE o TRUNCATE).*
- *TD["when"]: almacena como un texto el momento de ejecución del trigger (BEFORE, AFTER o INSTEAD OF).*
- *TD["new"], [.old"]: almacenan el registro nuevo y viejo respectivamente, en dependencia del evento que se ejecute.*
- *TD["table_name"]: almacena el nombre de la tabla que dispara el trigger.*

Se devuelve NONE u OK para dar a conocer que la operación con la fila se ejecutó correctamente o bien SKIP para terminar el evento.

El código a continuación muestra un trigger que verifica, al insertar, si el título es "HOLA

MUNDO" y si es así, no lo inserta:

Creamos la función:

```
-- Función trigger
CREATE FUNCTION trigger_plpython() RETURNS trigger AS
$$
plpy.notice('Comenzando el trigger')
IF TD["event"] == "INSERT" AND TD["new"]['title'] == "HOLA MUNDO":
plpy.notice('Se abortó la inserción por tener el título: ' + TD["
    new"]['title'])
RETURN "SKIP"
ELSE:
plpy.notice('Se registró correctamente')
RETURN "OK"
$$
languaje 'plpythonu';
```

Creamos el trigger:

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

Invocación de la función:

```
debian=# INSERT INTO
products(prod_id,category,title,actor,price,special,common_prod_id
    )
debian=# VALUES (20001, 13, 'HOLA MUNDO', 'Anthony Sotolongo',
    12.0, 0, 1000);
NOTICE: Comenzando el trigger
CONTEXT: PL/Python function "trigger_plpython"
NOTICE: Se abortó la inserción por tener el título: HOLA MUNDO
CONTEXT: PL/Python function "trigger_plpython"
```

11.4. PL/R

PL/R es un lenguaje procedural para PostgreSQL que escribe funciones en el lenguaje R para su empleo dentro del gestor. Es desarrollado por Joseph E. Conway desde el 2003 y compatible con PostgreSQL desde su versión 7.4.

Soporta casi todas las funcionalidades de R desde el gestor. Este lenguaje está orientado específicamente a operaciones estadísticas, es muy potente en esta rama, cuenta con variados paquetes (conjunto de funcionalidades) para su trabajo y se utiliza en varias áreas de la informática, minería de datos, sistemas experto, medicina, bioinformática, entre otras.

La instalación de PL/R es a partir de la versión 9.1 en adelante, cuando fue añadido en PostgreSQL el mecanismo de extensiones, se puede emplear el comando create extension plr. También, se instala desde la línea de comandos con createlang plr nombre_basedatos.

Se debe tener instalado en los sistemas Debian/Ubuntu el paquete PostgreSQL-9.1-plr o superior

11.4.1. Escribir funciones en PL/R

Una función en PL/R se define igual que las demás funciones en PostgreSQL, con la sentencia CREATE FUNCTION. El cuerpo de la función es código R y tiene la particularidad de que las funciones deben nombrarse de forma diferente aunque sus atributos no sean los mismos.

El retorno de la función de R se realiza con RETURN, pero en ocasiones no es necesario escribirla.

El código siguiente muestra cómo sumar dos valores con PL/R.

Función en PL/R que suma 2 números pasados por parámetros:

```
1 CREATE FUNCTION suma(a integer, b integer) RETURNS integer AS
2 $$
3 RETURN (a + b)
4 $$
5 languaje 'plr';
```

11.4.2. Pasando parámetros a una función PL/R

Los parámetros de una función se pasan como en cualquier lenguaje procedural y:

- *Pueden nombrarse, debiendo ser llamados por dicho nombre dentro de la función.*
- *De no ser nombrados pueden ser accedidos con argN, siendo N el orden que ocupan en la lista de parámetros.*

El código a continuación muestra la implementación una función que suma 2 números pasados por parámetros, sin nombrarlos:

```
1 CREATE OR REPLACE FUNCTION suma(integer, integer) RETURNS integer
     AS
2 $$
3 RETURN (arg1 + arg2)
4 $$
5 languaje 'plr';
```

11.4.3. Utilizando arreglos como parámetros

En una función, cuando se recibe un arreglo como parámetro, se convierte a un vector `c(...)`.

La función siguiente calcula la desviación estándar de un arreglo pasado por parámetro:

```
1 -- Cálculo de la desviación estándar desde PL/R
2 CREATE OR REPLACE FUNCTION desv_estandar(arreglo int[]) RETURNS
     real AS
3 $$
4 desv <- sd(arreglo)
5 RETURN (desv)
6 $$
7 languaje 'plr';
```

Invocación de la función:

```
1 debian=# SELECT desv_estandar(array[4,2,3,4,5,6,3]);
2 desv_estandar
3 1.34519
4 (1 fila)
```

11.4.4. Utilizando tipos de datos compuestos

En PL/R se puedan pasar tipos de datos compuestos definidos por el usuario, los cuales son convertidos en R a data.frames de una fila.

Pasando un tipo de dato compuesto como parámetro:

```
1  -- Crear tipo de dato compuesto
2  CREATE TYPE mini_prod AS (nombre varchar, precio numeric);
3  -- Función que recibe un tipo de dato compuesto como parámetro
4  CREATE OR REPLACE FUNCTION compuesto(a mini_prod) RETURNS text AS
5  $$
6  IF (a$precio == 0)
7  {RETURN (print("Precio incorrecto"))}
8  RETURN (print("Precio correcto"))
9  $$
10 languaje 'plr';
```

Invocación de la función

```
1 debian=# SELECT compuesto((ROW('Suite Habana', 0)));
2 compuesto
3 Precio incorrecto
4 (1 fila)
```

11.4.5. Homologación de tipos de datos PL/R

La homologación de tipos de datos de PostgreSQL a PL/R se realiza como muestra la tabla siguiente.

Tabla 11.4: Homologación de tipos de datos PostgreSQL a PL/R

PostgreSQL	R
boolean	logical
int8, float4, float8, cash, numeric	numeric
int, int4	integer
bytea	objetc
otro	caracter

11.4.6. Retornando valores de una función en PL/R

Los resultados de una función se devuelven con RETURN y, entre paréntesis se especifica el valor a devolver, que debe coincidir con el tipo de dato definido en la declaración de la función.

11.4.7. Devolviendo arreglos

Para devolver arreglos, éstos se devuelven como arreglos de una dimensión, la función a continuación devuelve el resumen estadístico de un vector como un arreglo de PostgreSQL.

Devolución de valores, con arreglos, desde PL/R:

```
1 -- Función que realiza un resumen estadístico de un arreglo pasado
      por parámetro
2 CREATE OR REPLACE FUNCTION resumen_estadistico(a integer[])
     RETURNS real[] AS
3 $$
4 resumen <- summary(a)
5 RETURN (resumen)
6 $$
7 languaje 'plr';
```

Invocación de la función:

```
1 debian=# SELECT resumen_estadistico(array[2,5,3,2,2,7,8,0]);
2 resumen_estadistico
3 {0,2,2.5,3.625,5.5,8}
4 (1 fila)
```

11.4.8. Devolviendo tipos compuestos

Para devolver tipos compuestos se utiliza un data.frame, con los respectivos atributos del tipo de dato compuesto definido por el usuario, con los atributos en orden respecto a los tipos

de datos.

Devolviendo valores con tipos de datos compuestos:

```
1 CREATE OR REPLACE FUNCTION devolvercompuesto() RETURNS SETOF
     mini_prod AS
2 $$
3 RETURN (data.frame(nombre="Los pájaros tirándole a la escopeta",
     precio=2.50))
4 $$
5 language 'plr';
```

Invocación de la función:

```
1 debian=# SELECT devolvercompuesto();
2 nombre | precio
3 Los pájaros tirándole a la escopeta | 2.5
4 (1 fila)
```

11.4.9. Devolviendo conjuntos

Se pueden devolver conjuntos de datos de algún tipo compuesto, TABLE o RECORD. A continuación se muestran dos ejemplos haciendo uso de ambos tipos.

Devolución de conjuntos:

```
1  -- Devolviendo conjuntos haciendo uso de TABLE
2  CREATE OR REPLACE FUNCTION devolver_varios_table() RETURNS TABLE(a
       text, b
3  numeric) AS
4  $$
5  nombres <- c("Fresa y Chocolate", "La película de Ana", "Conducta"
       )
6  precios <- c(25.01, 10.65, 60)
7  dataframe <- data.frame(nombre = nombres, precio = precios)
8  RETURN (as.matrix(dataframe))
9  $$
10 language 'plr';
```

Llamado de la función:

```
1 debian=# SELECT * FROM devolver_varios_table();
2 a | b
3 Fresa y Chocolate | 25.01
4 La película de Ana | 10.65
5 Conducta | 60.00
6 (3 filas)
```

Devolviendo conjuntos usando RECORD:

```
-- Devolviendo conjuntos haciendo uso de RECORD
CREATE OR REPLACE FUNCTION devolver_varios_record() RETURNS SETOF
    RECORD AS
$$
nombres <- c("Fresa y Chocolate", "La película de Ana", "Conducta"
    )
precios <- c(25.01, 10.65, 60)
dataframe <- data.frame(nombre = nombres, precio = precios)
RETURN (dataframe)
$$
languaje 'plr';
```

Invocación de la función:

```
debian=# SELECT * FROM devolver_varios_record() AS (a text, b
    numeric);
a | b
Fresa y Chocolate | 25.01
La película de Ana | 10.65
Conducta | 60.00
(3 filas)
```

11.4.10. Ejecutando consultas en la función PL/R

Para ejecutar consultas se pueden utilizar varias funciones como:

- *pg.spi.exec(consulta): donde consulta es una cadena de caracteres y la función devuelve los datos de un SELECT en un data.frame de R; si es una consulta de modificación entonces devuelve el número de filas afectadas.*
- *pg.spi.prepare: permite preparar consultas y salvar el plan generado para una ejecución posterior de las mismas; el plan soólo se graba durante la conexión o transacción en curso.*
- *pg.spi.execp: ejecuta una consulta previamente preparada con pg.spi.prepare y permite los argumentos para la consulta preparada.*
- *pg.spi.cursor_open y pg.spi.cursor_fetch: empleados para el trabajo con cursores.*

Devolución del resultado de una consulta usando pg.spi.exec:

```
CREATE OR REPLACE FUNCTION devolver\_consulta() RETURNS SETOF mini
    \_prod AS
$$
resultado <- pg.spi.exec( " SELECT title, price FROM products
    WHERE prod_id <
100 " )
RETURN (resultado)
```

```
$$
languaje 'plr';
```

Invocación de la función:

```
debian=# SELECT * FROM devolver_consulta();
nombre | precio
ACADEMY ACADEMY | 25.99
ACADEMY ACE | 20.99
ACADEMY ADAPTATION | 28.99
ACADEMY AFFAIR | 14.99
-- More --
```

11.4.11. Mezclando

Estos lenguajes de desconfianza son utilizados para hacer rutinas externas al servidor de bases de datos, a continuación se muestra un código para generar un gráfico de barras en un archivo .png con el resultado de una consulta.

Función que genera una gráfica de barras con el resultado de una consulta en PL/R:

```
CREATE FUNCTION barras_simple(nombre text, consulta text, texto
    text, ejex text[])
RETURNS integer AS
$$
png(paste(nombre,"png", sep="."))
resultado <- pg.spi.exec(consulta)
barplot(as.matrix(resultado),beside=TRUE,main=texto,col=rainbow(
    length(as.
matrix(resultado))), names.arg=c(ejex))
dev.off()
$$
languaje 'plr';
```

Invocación de la función:

```
debian=# SELECT * FROM barras_simple('grafica_barras',
debian=# 'SELECT count(products.prod_id) AS cantidad FROM products
     JOIN categories ON
categories.category=products.category
GROUP BY categories.categoryname,
categories.category ORDER BY categories.category LIMIT 4',
debian=# 'Cantidad producto x Categoría',
debian=# array(SELECT categoryname FROM categories ORDER BY
    categoryname LIMIT 4
)::text[]);
```

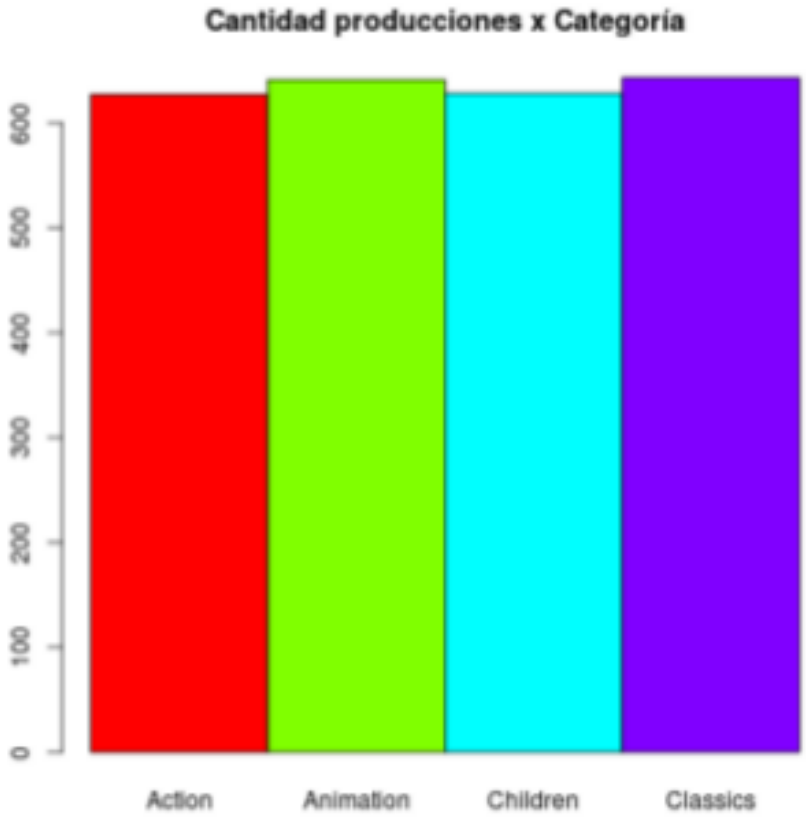


Figura 11.1: Gráfica de barras generada con el resultado de una consulta en PL/R

11.4.12. Realizando triggers con PL/R

En PL/R se escriben funciones activadoras, para eso cuenta con un diccionario TD, con varios pares clave/valor útiles; algunos se describen a continuación:

- *pg.tg.relname: devuelve el nombre de la tabla que invocó al trigger.*
- *pg.tg.when: devuelve una cadena, en mayúscula, especificando cuándo se ejecutó el trigger (BEFORE o AFTER).*
- *pg.tg.op: devuelve una cadena, en mayúscula, del evento que ejecutó el trigger (INSERT, UPDATE o DELETE).*
- *pg.tg.new: data.frame con los valores nuevos de la fila nueva insertada o modificada, puede llamarse utilizando el nombre de la columna de la tabla, por ejemplo pg.tg.new$columna.*
- *pg.tg.old: data.frame con los valores viejos que tiene la fila actualizada o eliminada, puede llamarse utilizando el nombre de la columna de la tabla, por ejemplo pg.tg.new$columna.*

Existen otras variables como pg.tg.level, pg.tg.relid, pg.tg.args, pg.tg.name.

El retorno del trigger puede ser NULL, una fila en forma de data.frame(pg.tg.new, pg.tg.old) o alguno que contenga las mismas columnas de la tabla que lo invocó. NULL significa que el resultado de la operación realizada será ignorado.

El código a continuación muestra el empleo de un trigger desde PL/R:

```
-- Función trigger
CREATE FUNCTION trigplrfunc() RETURNS trigger AS
$$
pg.thrownotice ("Comenzado el trigger")
IF (pg.tg.op == "INSERT" & pg.tg.new$price == 0) {
pg.thrownotice("Registro no insertado")
RETURN (NULL)
}
RETURN (pg.tg.new)
$$
languaje 'plr';
```

El trigger es:

```
CREATE TRIGGER testplr_trigger
BEFORE INSERT OR UPDATE
ON products
FOR EACH ROW
EXECUTE PROCEDURE trigplrfunc();
```

Invocación de la función:

```
debian=# INSERT INTO products (prod_id, category, title, actor,
   price, special,
common_prod_id) VALUES (20003, 13, "Don Quijote de la Mancha", '
   Miguel de Cervantes', 30, 0,
1000);
NOTICE: Comenzado el trigger
NOTICE: Registro no insertado
```

Consulta exitosa: 0 filas afectadas, tiempo de ejecución 33 ms con PL/Python y PL/R.

1. *Desarrolle una función en PL/Python que devuelva el monto total(price* quantity) de las canciones donde el compositor es "JUAN GABRIEL".*

2. *Construya una función que devuelva los 100 automóviles más caros.*

3. *Elabore una función en PL/Python que exporte los datos del ejercicio 2 a un archivo CSV.*

4. *Realice una función en PL/Python para devolver los laptops que su marca comience con un carácter pasado por parámetro, prepare dicha consulta antes de ejecutarla.*

5. *Conciba una función en PL/Python que permita exportar los datos del ejercicio 4 a un archivo Excel.*

6. *Implemente un mecanismo basado en triggers en PL/Python para:*

   a) *Llevar un registro de las categorías eliminadas.*

   b) *Controlar que si se agrega o modifica el precio de una producción y este precio es 0, le envíe un correo al administrador del sistema (suponga un correo* `admin@dellstore.com`*) notificando la situación.*

7. *Logre una función en PL/R que devuelva los productos que su precio sea mayor a uno pasado por parámetro, prepare dicha consulta antes de ejecutarla.*

8. *Confeccione funciones en PL/R que permitan obtener gráficos de:*

   a) *Pastel con el resultado de la consulta del ejercicio 7.*

   b) *Histograma de las películas del actor "SEAN CONNERY".*

9. *Realice una función en PL/R para calcular la mediana de los libros del autor "DAN BROWN".*

11.5. Resumen

Los llamados lenguajes de "desconfianza" escriben lógica de negocio en el servidor de bases de datos PostgreSQL, escritas en sus propios lenguajes. En este capítulo se analizaron las características de los lenguajes PL/Python y PL/R, los cuales pueden ser útiles para determinadas operaciones con los datos. Para implementar una función en dichos lenguajes se emplea el comando CREATE FUNCTION, en el que se define el nombre de la función, los parámetros que recibirá y el tipo de retorno de la función y; su cuerpo estará compuesto por las características del lenguaje especificado. Se realizó una homologación con los tipos de datos de PostgreSQL, así como la ejemplificación de la implementación de triggers.

Se debe tener cuidado en el uso de estos lenguajes pues desde ellos se pueden acceder a recursos del servidor más allá de los datos almacenados. Se recomiendan utilizar en entornos controlados y para actividades que no se puedan realizar desde los lenguajes nativos como lo son el SQL y el PL/pgSQL. El rendimiento de los mismos no suele ser el mejor.

12

Migración desde Oracle

En este capítulo explicamos algunas diferencias entre el PL/SQL de Oracle y el PL/pgSQL de PostgreSQL con el propósito de ayudar a los desarrolladores a migrar aplicaciones de Oracle a PostgreSQL. La mayoría del código expuesto aquí es del módulo ArsDigita Clicks-tream 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 más características que su homónimo de PostgreSQL, pero PL/pgSQL permite una gran funcionalidad y es mejorado constantemente.

12.1. Principales Diferencias

Algunas cosas a recordar cuando migre 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 comando FOR (vea el código más adelante).*
- *En PostgreSQL necesita escapar las comillas simples.*

12.1.1. Escapando comillas simples

En PostgreSQL necesitamos escapar las comillas simples dentro de la definición de una función. Esto es importante recordarlo para el caso de crear funciones que generan otras

funciones. Vea la tabla 2.3.13, página 17.

Aquí tenemos una función de Oracle:

```
1  CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name IN
       varchar, v_version IN varchar)
2  RETURN varchar IS
3  BEGIN
4  IF v_version IS NULL THEN
5  RETURN v_name;
6  END IF;
7  RETURN v_name || '/' || v_version;
8  END;
9  /
10 SHOW ERRORS;
```

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

1. *PostgreSQL no tiene parámetros nominados. Tenemos que especificarlos como ALIAS dentro de su función.*

2. *Oracle puede tener parámetros IN, OUT, y INOUT pasados a funciones. El INOUT, significa que el parámetro recibe un valor y regresa otro. PostgreSQL sólo tiene parámetros IN y las funciones sólo pueden regresan un valor simple.*

3. *La palabra RETURN en la función prototype (no en la función body) regresa RETURNS en PostgreSQL.*

4. *Las funciones PostgreSQL son creadas usando comillas simples y delimitadores, así que tiene que escapar comillas simples dentro de sus funciones.*

5. *El comando /show errors no existe en PostgreSQL.*

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

```
1  CREATE OR REPLACE FUNCTION cs_fmt_browser_version(VARCHAR, VARCHAR
       )
2  RETURNS VARCHAR AS '
3  DECLARE
4  v_name ALIAS FOR $1;
5  v_version ALIAS FOR $2;
6  BEGIN
7  IF v_version IS NULL THEN
8  RETURN v_name;
9  END IF;
10 RETURN v_name || ''/'' || v_version;
11 END;
12 ' LANGUAGE 'plpgsql';
```

Una Función que crea otra Función.

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

```
1  CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
2  CURSOR referrer_keys IS
3  SELECT * FROM cs_referrer_keys
4  ORDER BY try_order;
5  a_output VARCHAR(4000);
6  BEGIN
7  a_output := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(
       v_host IN VARCHAR, v_domain IN VARCHAR,
8  v_url IN VARCHAR) RETURN VARCHAR IS BEGIN';
9  FOR referrer_key IN referrer_keys LOOP
10 a_output := a_output || ' IF v_' || referrer_key.kind || ' LIKE ''
       ' ||
11 referrer_key.key_string || ''' THEN RETURN ''' || referrer_key.
       referrer_type ||
12 '''; END IF;';
13 END LOOP;
14 a_output := a_output || ' RETURN NULL; END;';
15 EXECUTE IMMEDIATE a_output; END;
16 /
17 show errors
```

Aquí tenemos la misma función en PostgreSQL:

```
1  CREATE FUNCTION cs_update_referrer_type_proc() RETURNS INTEGER AS
       '
2  DECLARE
3  referrer_keys RECORD; -- Declare a generic record to be used in a
       FOR
4  a_output varchar(4000);
5  BEGIN
6  a_output := ''CREATE FUNCTION cs_find_referrer_type(VARCHAR,
       VARCHAR,VARCHAR)
7  RETURNS VARCHAR AS ''''
8  DECLARE
9  v_host ALIAS FOR $1;
10 v_domain ALIAS FOR $2;
11 v_url ALIAS FOR $3;
12 BEGIN '';
13 --
14 -- Advierta cómo escaneamos a través de los resultados de una
       consulta en un bucle
```

```
-- 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';
```

Un Procedimiento con manipulación de cadenas y parámetros OUT.

El siguiente código de Oracle PL/SQL es usado para interpretar una URL y regresar varios elementos (máquina, ruta y consulta). Es un procedimiento porque en las funciones PL/pgSQL sólo puede ser devuelto un valor.

En PostgreSQL, una forma de grabar 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
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;
instr
```

Aquí tenemos cómo éste procedimiento es migrado 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 tenemos que solventarlo usando una combinación de otras funciones.He creado mis propias funciones instr que hacen exactamente lo mismo que las de Oracle.*

12.2. Procedimientos

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

Un código:

```
1  CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
2  a_running_job_count INTEGER;
3  PRAGMA AUTONOMOUS_TRANSACTION;(1)
4  BEGIN
5  LOCK TABLE cs_jobs IN EXCLUSIVE MODE;(2)
6  SELECT count(*) INTO a_running_job_count
7  FROM cs_jobs
8  WHERE end_stamp IS NULL;
9  IF a_running_job_count > 0 THEN
10 COMMIT; -- free lock(3)
11 raise_application_error(-20000, 'Unable to create a new job: a job
       is currently running.');
12 END IF;
13 DELETE FROM cs_active_job;
14 INSERT INTO cs_active_job(job_id) VALUES (v_job_id);
15 BEGIN
16 INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id,
       sysdate);
17 EXCEPTION WHEN dup_val_on_index THEN NULL; -- don't worry if it
       already exists(4)
18 END;
19 COMMIT;
20 END;
21 /
22 show errors
```

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

1. *No hay ningún comando pragmático en PostgreSQL.*
2. *Si realizamos 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 comando BEGIN.*
4. *La excepción a esto es cuando tuviera que ser reemplazado por un comando IF.*

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

```
1  CREATE OR REPLACE FUNCTION cs_create_job(INTEGER) RETURNS
       INTEGER AS '
2  DECLARE
3  v_job_id ALIAS FOR $1;
4  a_running_job_count INTEGER;
5  a_num INTEGER;
6  -- PRAGMA AUTONOMOUS_TRANSACTION;
7  BEGIN
8  LOCK TABLE cs_jobs IN EXCLUSIVE MODE;
9  SELECT count(*) INTO a_running_job_count
10 FROM cs_jobs
11 WHERE end_stamp IS NULL;
12 IF a_running_job_count > 0 THEN
13 -- COMMIT; -- free lock
14 RAISE EXCEPTION ''Unable to create a new job: a job is
       currently running.'';
15 END IF;
16 DELETE FROM cs_active_job;
17 INSERT INTO cs_active_job(job_id) VALUES (v_job_id);
18 SELECT count(*) into a_num
19 FROM cs_jobs
20 WHERE job_id=v_job_id;
21 IF NOT FOUND THEN
22 -- If nothing was returned in the last query
23 -- This job is not in the table so lets insert it.
24 INSERT INTO cs_jobs(job_id, start_stamp) VALUES (v_job_id,
       sysdate());
25 RETURN 1;
26 ELSE
27 RAISE NOTICE ''Job already running.'';(1)
28 END IF;
29 RETURN 0;
30 END;
31 ' LANGUAGE 'plpgsql';
```

12.3. Tópico selecto

Cómo migrar un esquema de Oracle hacia PostgreSQL, y necesito convertir ciertas funciones de la BD origen a sus análogas en el destino.

La forma más parecida de simular un bulk collect en PostgreSQL es con arrays y de igual forma con tipos de datos personalizados.

Código SQL:

```
CREATE TYPE type_tu_tipo AS
(fecha TIMESTAMP WITHOUT TIME zone,
edad INTEGER,
nombre VARCHAR(100));
```

Debemos declarar los tipos antes de ser usados, a diferencia de ORACLE que pueden ser volatiles en el tiempo de vida del bloque, función o procedimiento.

Posterior, se declara:

```
v_mi_variable type_tu_tipo[];
v_mi_variable := array(SELECT (fecha,edad,nombre)::type_tu_tipo
   FROM tabla);
```

Después de tener el array cargado, procedemos a recorrerlo.

13

Ejercicios y casos de estudio

Este capítulo sustenta los conceptos y la experiencia necesarios para conocer detalladamente el sistema, tratando de estimular el esfuerzo del lector en la preparación del ambiente de trabajo y en la solución de los ejercicios y casos de estudio. Los conceptos básicos se aplican en torno al mismo proyecto que usaremos en esta serie: "Universidad DASD", en lo sucesivo DASD, la cual es producto de la imaginación del autor.

Los libros y artículos que se ofrecen en la sección de referencias, apoyan algunos de los conceptos que se aplican en la solución práctica de problemas de administración de base de datos.

Estos ejercicios se han preparado para generar experiencia práctica a los estudiantes o profesionales de la "Administración de Base de Datos". El autor tiene por lo menos 14 años de experiencia en el uso de PostgreSQL en las aulas, proyectos de investigación y en sistemas que se han implementado para la automatización de las actividades cotidianas de empresas publicas y privadas. Como producto de esa experiencia académica e industrial se han obtenido estos ejercicios para capacitar a nuestros lectores. También puede ser una fuente de consulta para los profesionales que laboran en el sector empresarial.

Como se ha mencionado previamente PL/pgSQL tiene características y programación estándar que ofrecen sistemas propietarios, por lo que los ejemplos fácilmente pueden ser aplicados en otros sistemas de bases de datos del mercado, o pueden ser referencia para aplicarlos en proyectos industriales.

13.1. Objetivo

El lector aprenderá a usar los conceptos básicos del lenguaje PL/pgSQL y resolverá ejercicios acordes a los disparadores en la Base de Datos PostgreSQL.

13.2. Ejercicios

1. *Escriba una función que reciba dos parámetros y devuelva el resultado de la operación suma realizada con ellos.*

2. *Escriba una función que utilice datos provenientes de una consulta SQL.*
3. *Escriba una función que reciba un parámetro y realice una consulta SQL con ese valor.*
4. *Escriba una función que haga uso de la sentencia IF...THEN...ELSE...END*
5. *Se dispone de una tabla llamada resultados de partidos de voley-ball. Esta tabla tiene la siguiente estructura:*

   num_partido int PK

   jornada int

   eq_local char(3)

   set_local int

   eq_visitante char(3)

   set_visitante int

   Como ejemplo, 1-1-NUM-3-ALM-2, significa que el partido número 1, perteneciente a la primera jornada el equipo Pumas ganó por 3 a 2 a América.

   Sabiendo que el sistema de puntuación de esta competición hace que se den 3 puntos a quien gana por 3-0 o 3-1, 2 puntos a quien gana por 3-2, 1 punto a quien pierde por 2-3 y 0 puntos a quien pierda por 0-3 o 1-3. Escriba la función puntos() a la que se pasa el código de un equipo y devuelve los puntos conseguidos de acuerdo a sus resultados.
6. *Escriba una función que recorra una matriz.*

Funciones que recorren un resultset.

La sintaxis para un resultset que recorre un registro es:

```
FOR reg IN sql LOOP
// Acciones con el registro
END LOOP;
```

Se evalúa la consulta SQL que genera un resultset.

La variable reg (del tipo RECORD) recoge los campos del SQL de forma que se pueden manejar con el formato reg.campo1, reg.campo2, ...

Se repite el proceso para cada registro del resultset

Supongamos que se dispone de una tabla que almacena los movimientos diarios de caja de una empresa y se desea que el campo saldo tenga actualizado su valor:

```
Tabla: diario
id integer PK
fecha date\s
ingreso double
gasto double
saldo double
```

```
DECLARE
reg RECORD;
valor double precision;
BEGIN
valor=0;
FOR reg IN SELECT id,ingreso,gasto FROM diario ORDER BY id LOOP
if(reg.ingreso is not null) then
valor=valor+reg.ingreso;
end if;
if(reg.gasto is not null) then
valor=valor-reg.gasto;
end if;
UPDATE diario SET saldo=valor WHERE id=reg.id;
END LOOP;
return 1;
END;
```

13.3. Casos de estudio

El caso de estudio consiste en un proyecto que describe el problema de una empresa dedicada a la prestación de servicios educativos: después de leer el texto se genera el diagrama E-R con la solución a este problema, se continúa con la creación de las tablas y su población, para finalmente trabajar con los permisos de grupos y usuarios.

13.3.1. Proyecto BANCO

Siguiendo el modelo entidad-relación de la figura 13.1, creamos la base de datos BANCO y sus respectivas tablas:

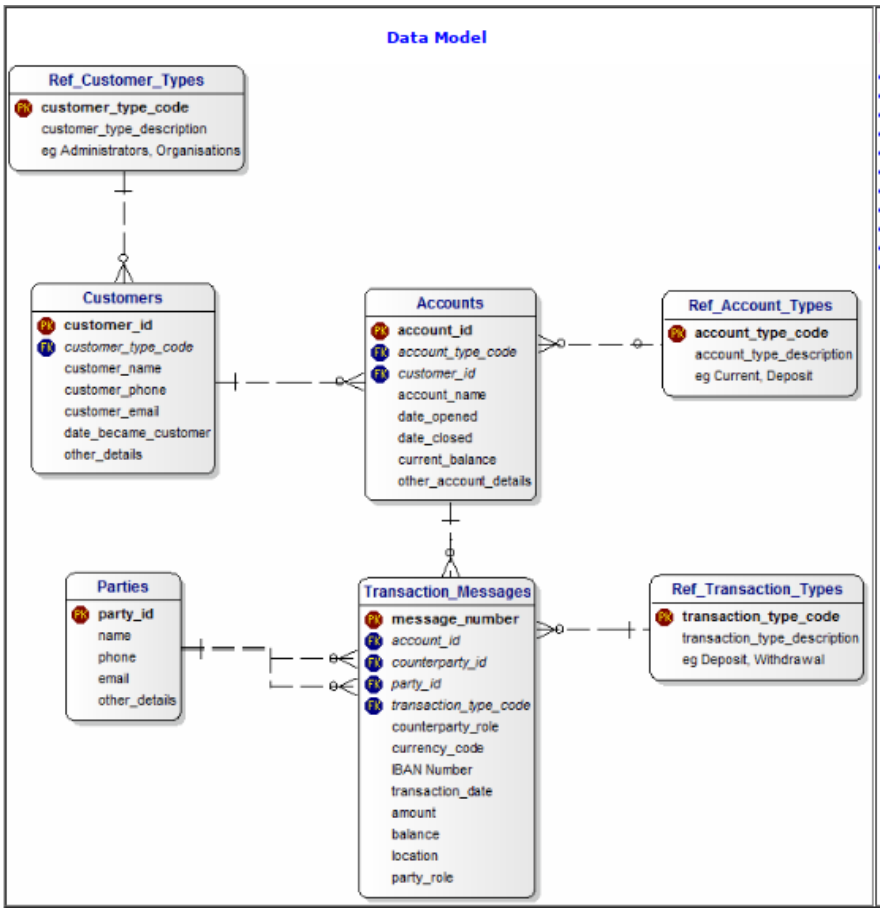


Figura 13.1: Modelo entidad-relación de la base de datos BANCO.

El modelo E-R tiene las siguientes restricciones primarias:

- *El campo costumer_phone tiene la siguiente restricción, debe estar entre 6400000 y 650000.*
- *El campo phone de parties tiene la siguiente restricción, debe estar entre 6400000 y 650000.*
- *El campo DepositWithdrawl tiene la siguiente restricción: ser positivo.*
- *El campo balance tiene la siguiente restricción: ser positivo.*
- *El campo account_type_code tiene la siguiente restricción: 1 ahorro, 2 corriente, 3 inversiones.*
- *El campo current_balance tiene la siguiente restricción: ser positivo.*

13.3.1.1. Trabajo adicional

Un cliente debe ser único en el banco, aunque puede tener más de una cuenta. Dos o más clientes pueden tener una misma cuenta. Las cuentas pueden ser de ahorro y corriente.

13.3.1.2. Problemática a resolver

Escriba las consultas y/o funciones para realizar las siguientes tareas, creando:

1. *el esquema privado donde debe estar incluida la tabla cliente y gerente.*
2. *el dominio teléfono que restrinja los datos entre 6400000 y 6499999.*
3. *las siguientes tablas según las especificaciones indicadas.*
4. *Un (1) gerente, seis (6) sucursales, veinticinco (25) clientes, quince (15) cuentas y diez (10) préstamos.*
5. *un proceso para verificar el resultado de insertar una cuenta con saldo negativo, un interés de 5 %.*
6. *la tabla temporal cliente-cuenta y mostrar los datos personales del cliente y su saldo.*
7. *una función que devuelva el interés ganado de una cuenta.* `Saldo*interes\_mesual`.
8. *una función que devuelva el mayor préstamo realizado a un cliente.*
9. *una función para devolver la cantidad total de prestamos de una sucursal.*
10. *un cursor que contenga los préstamos realizados en el último mes y mostrar los 2 últimos movimientos.*
11. *un cursor con los clientes de Sucre y mostrar los 3 primeros.*
12. *una función en PL/pgSQL que devuelva cuantos clientes tienen una cuenta superior a 5000.*
13. *una función que en PL/pgSQL que devuelva cuantos clientes tienen una cuenta creada por lo menos hace un año.*
14. *una vista para los clientes por ciudad.*
15. *una vista con todos los datos de las sucursales.*
16. *una vista con los clientes y sus cuentas.*
17. *una vista con los clientes y sus préstamos.*
18. *reglas para que no se puedan insertar datos en las vistas.*

19. *un trigger que controle que el capital no sea menor a 1000.*

20. *un trigger que controle que el saldo de un cliente sea menor a 100.*

21. *un trigger que controle que ningún cliente saque un préstamo mayor al capital de la sucursal.*

22. *un Trigger para calcular el interés de un préstamo al concluir el periodo de cálculo de interés.*

13.3.2. Proyecto Universidad DASD

En DASD se ofrecen dos tipos de cursos en el periodo de vacaciones, agosto-septiembre en 6 semanas, en el cual se imparten cursos de verano y extracurriculares. Los primeros son materias que un estudiante regular que estudia una carrera reprueba, por lo que se le permite reparar hasta dos materias; mientras que los segundos son cursos especiales para adelantar que se ofrecen a estudiantes regulares como estudiantes y/o a profesionales externos.

Los docentes de la DASD, son los únicos que imparten estos cursos, por los cuales recibe un pago adicional, se les paga según un tabulador que indica el costo de la hora de estos cursos de acuerdo al nivel académico del docente. El pago se genera a partir del alta del curso y sólo se expide un cheque por cada curso.

Además los estudiantes deben acudir a pagar adicionalmente al costo del curso por asistir a ellos.

DASD tiene dos departamentos que intervienen en la gestión de los cursos:

1. *Departamento de Administración (DA). Le corresponde efectuar el pago a los docentes y los cobros a los estudiantes; es dirigido por el C.P. López y es auxiliado por el Sr. Jurado.*

2. *Departamento de Control Estudios (DCE). Es quien decide que cursos se imparten en el periodo, quién los imparte, y acepta las solicitudes de los estudiantes. Un caso especial, es el de los docentes, ya que DA es quién puede modificar el sueldo quincenal, mientras que DCE no puede visualizar éste. Lo curioso radica en que, es DCE quién acepta a los docentes y a los registra en el sistema, pero es DA donde se captura el sueldo.*

   Importante es para la administración de DASD que esta política se aplique al pie de la letra, y que sea implementada directamente sobre la DB.

A continuación se describen las tablas a las cuales tiene acceso el personal de DA: CuentaCheques, Cheque, Tabulador, docentes, Concepto, Recibo, y DetalleRecibo. En casos especiales, DA podrá acceder a consultar las tablas de Cursos Especiales, Cursos Especiales Verano, Cursos Especiales Extracurriculares, Cursos Extracurriculares y Materias. Explícitamente, no se le permite modificar ningún campo o registro.

El personal de DA tiene acceso a las tablas: CursosEspeciales, CursosExtracurricular, Materias, CEVerano, CEExtracurricula, estudiantes, Bimestre, Faltas, CalendarioEscolar.

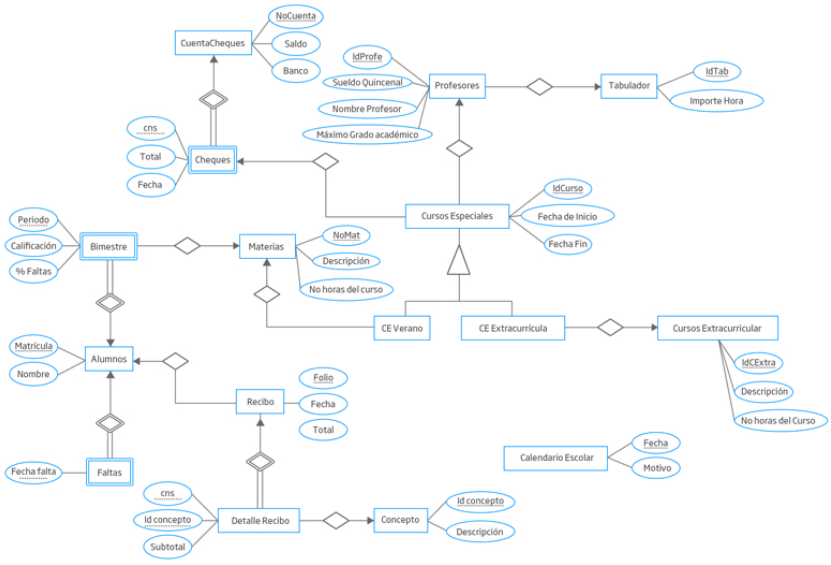


Figura 13.2: Modelo Entidad-Relación de la base de datos DASD.

Para aplicar los disparadores a nuestra base de datos haremos un cambio en la base de datos. El cual consiste en que al momento de pagar un estudiante su inscripción a un curso extracurricular se le pregunta el idcurso a inscribir, debemos relacionarlo con el folio y el consecutivo del detalle del recibo en el que ha pagado el curso y automáticamente incrementar el campo Total Recabado en la tabla Cursos Especiales, esto con el fin de que DCE vea el ingreso económico de este curso. La figura 13.3.2, página 166 muestra la modificación al diagrama. Las entidades y relaciones, en rojo, son las modificaciones de este diagrama, las demás entidades permanecen intactas y se han incluido para referencia.

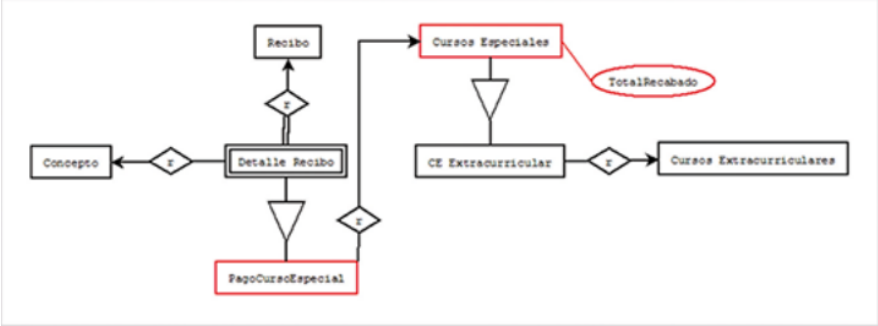


Figura 13.3: Modelo Entidad-Relación según cambio propuesto.

El problema consiste que cuando se detecte que el estudiante Rodolfo ha pagado el concepto 50 (pago de cursos especiales) por el Curso 75, el cual es un curso extracurricular de Paradigmas de programación, lenguaje C, y que será impartido del primer lunes de agosto al jueves de la segunda semana de septiembre del año actual y por el cual paga Bs.S. 450. En el momento en el que el recibo y el detalle del recibo se actualicen, también se actualiza

la tabla PagoCursoEspecial, entonces automáticamente se debe de incrementar el importe recabado para el curso que se ha pagado en la tabla CursosEspeciales. Asimismo, si el recibo es cancelado, entonces el importe recabado en la tabla CursosEspeciales debe ser disminuido. Los cambios efectuados se ven reflejados en las siguientes definiciones de tablas:

```
1 -- creando la nueva tabla
2 CREATE TABLE PagoCursoEspecial(
3 folio int,
4 cns int,
5 idcurso int REFERENCES CursosEspeciales,
6 PRIMARY KEY(folio, cns),
7 FOREIGN KEY(folio, cns) REFERENCES DetalleRecibo
8 );
```

> ***NOTA***: *borrando la tabla de CursosEspeciales para después redefinirla, cuidado con el cascade ya que si tiene datos relacionados con ella puede llegar a borrar varias tablas y puede tener que volver a crearlas a partir del laboratorio 1.*
>
> ```
> drop table CursosEspeciales cascade;
> ```

```
1 -- redefiniendo la tabla de CursosEspeciales
2 -- con los cambios requeridos
3 CREATE TABLE CursosEspeciales(
4 idcurso int,
5 idprofe int,
6 fini varchar,
7 ffin varchar,
8 ncuenta int,
9 cns int,
10 TotalRecabado numeric(10,2),
11 CostoCurso numeric(10,2),
12 FOREIGN KEY(idprofe) REFERENCES docentees,
13 FOREIGN KEY(ncuenta, cns) REFERENCES Cheque,
14 primary key (idcurso)
15 );
```

Ahora, preparamos el entorno para trabajar con los elementos adecuados, ejecutamos los siguientes comandos con el usuario postgres: – Insertando datos de estudiantes `INSERT INTO estudiantes VALUES (100, 'Juan Perez');`

– Insertando datos de concepto

`INSERT INTO concepto VALUES (50, 'Pago de Cursos Especiales');`

– Insertando datos del CursosEspeciales `INSERT INTO CursosEspeciales VALUES ( 75, 5, '2008-01-10', '2008-01-30', 3, 10, 0.0, 4500 );`

– Insertando datos del CEExtracurricula

```
INSERT INTO CEExtracurricula VALUES ( 75, 4 );
```

– *Insertando datos de Recibo*

```
INSERT INTO Recibo VALUES(200, 100, '2008-02-10', 4500 );
```

– *Insertando datos de Detalle de Recibo*

```
INSERT INTO DetalleRecibo VALUES ( 1, 50, 200, 4500 );
```

Ésta es la función denominada ActualizadorTotal, cuya tarea es la de hacer el incremento del total recabado. Está escrita con el lenguaje denominado PLPGSQL, y debe copiarse usando la cuenta del usuario postgres.

Ésta es la función DecrementadorTotal(), cuya tarea es la de decrementar el total recabado. Está escrita con el lenguaje denominado PLPGSQL, y debe copiarse usando la cuenta del usuario postgres.

Finalmente, construimos un disparador para cuando alguien que había pagado un curso y decide retirarlo, ante esta situación el total recaudado debe de disminuir.

13.3.2.1. Trabajo adicional

Escriba los disparadores para que cumpla con la siguiente condición: El estudiante Rodolfo ha decidido de última hora que va a asistir a dos cursos extracurriculares, "Paradigmas de programación" que es el curso 75 y "Programación Avanzada de Procedimientos Almacenados" que es el curso 100. Así que va a efectuar el pago y la cajera en el mismo recibo (el folio 600) le cobra los dos cursos. ¿Qué modificaciones debe hacer en el programa que incrementa el total recabado? Y ¿En el qué lo disminuye?

En apariencia las dos funciones hacen casi lo mismo, ¿Puede optimizar el trabajo? Es decir, hacer una sola función que sea capaz de controlar tanto el incremento como el decremento del total recabado.

13.3.2.2. Problemática a resolver

Para nuestro problema, debemos hacer algunas suposiciones. Resulta que los dos empleados del DA están generando cheques para pagar cursos de la cuenta 2 (con un saldo de Bs.S. 9000), uno para el docente Julio por Bs.S. 3000 y otro para el docente Anyerg por Bs.S. 1000, lo curioso es que al momento de generarlos y debido a que el sistema está funcionando en red con una base de datos centralizada, lo hacen al mismo tiempo, como consecuencia el saldo quedó en Bs.S. 6000 (o podría quedar en 8000 dependiendo de cuál cheque afecta el saldo primero). Así que efectuaremos primero las transacciones con un procedimiento almacenado pero sin usar transacciones y después incorporamos su uso para demostrar la utilidad de las mismas.

1. *1er. Caso:*

   Efectuaremos la expedición de dos cheques de manera simultánea desde los programas escritos en php, uno para el docente Julio y el otro para el docente Anyerg. Ejecute

el programa index.php en el IDE de su elección, en cada equipo. Estos programas se deben ejecutar en equipos distintos para obligar a la base de datos a tratar datos en forma concurrente. La figura 13.4, página 169, muestra los datos que se deben capturar en el programa index.php, desmarque el CheckBox de Transacciones de la pantalla. Se intenta forzar al PostgreSQL a cometer un error, por lo que después de capturar los datos se debe dar clic sobre el botón Enviar de manera simultánea en los dos equipos que forman parte de la red. Codifique el detalle del programa Index.php que invoca la función AltaDeCheque cuando no se selecciona el CheckBox indicado, debe notar que no se ha incorporado el uso de transacciones.

Figura 13.4: Datos a capturar en cada equipo de la red.

Finalmente consulte el saldo de la Cuenta de Cheques 2, ¿Cuál es el saldo? ¿Es correcto? De ser así, devuelva los valores a su estado previo y vuelva a intentarlo hasta que encuentre un resultado erróneo.

Explique la razón por la cual fallan los procesos o la razón por la cual falla si la invocamos desde php.

13.3.2.3. PL/pgSQL en redes informáticas

13.3.3. Configuración de la red

Instale la red de área local usando el mecanismo de su elección. Asigne a cada equipo una dirección IP estática.

13.3.3.1. Configuración de PostgreSQL para aceptar conexiones remotas

En cualquier sistema operativo que esté usando busque los archivos Postgresql.conf y Pg_hba.conf y efectúe en ellos los siguientes cambios.

```
1 Postgresql.conf
```

Busque el renglón con contiene el siguiente comando:

```
listen_address = 'localhost'
```

Y modifique con la siguiente configuración

```
listen_address = '*'
Pg_hba.conf
```

Busque el renglón donde se encuentra la configuración de IPv4:

```
# IPv4 local connections:
```

Elimine el renglón de abajo y ponga la siguiente configuración:

```
# IPv4 local connections:
host all all 127.0.0.1/32 md5
```

Es importante reiniciar el equipo que ejecuta el proceso servidor de PostgreSQL, para que desde el arranque este proceso adquiera los valores recién configurados.

13.3.3.2. Para usuarios de sistemas operativos Windows y Linux

a) *Windows*

Los archivos de configuración se encuentran en la ruta:

''C:\Program Files\PostgreSQL\9.3\data''.

b) *Linux*

Regularmente se encuentra en el directorio /usr/local/pgsql/data/, pero seguramente cada versión tiene su propia ruta de ubicación.

13.3.3.3. Preparación del ambiente de trabajo

Ejecute los programas de php en cada equipo y capture los mismos datos del caso 1 (Figura 13.3.2), página 166, sólo que ahora seleccione el CheckBox de Transacciones de la pantalla. Estos programas se deben ejecutar en equipos distintos para obligar a la base de datos a tratar datos en forma concurrente. La figura 4 muestra el detalle del programa Index.php que invoca la función AltaDeCheque cuando es seleccionado el CheckBox indicado, debe notar que se ha incorporado el uso de transacciones. Se intenta forzar al PostgreSQL a cometer un error, por lo que después de capturar los datos se debe dar clic sobre el botón Aceptar de manera simultánea en los dos equipos que forman parte de la red. Note que se ha agregado el comando "FOR UPDATE" al cuerpo de la llamada a la AltaDeCheque, y que previamente se ha invocado el comando "BEGIN TRANSACTION" posteriormente se ha agregado el comando "COMMIT TRANSACTION".

13.3.3.4. Trabajo adicional

- *Modifique la función AltaDeCheque para que en caso de que no se tenga saldo en la cuenta de cheques no se permita ejecutar la inserción del cheque y devuelva un valor de 0. Además modifique la invocación desde php para que en caso de que la función AltaDeCheque devuelva un cero se ejecute un Rollback Transaction y en caso de un 1 ejecutar un Commit Transaction;*
- *El mismo problema aparece con el total recabado por cada curso al momento de que dos cajeros cobren el mismo curso al mismo tiempo a dos estudiantes distintos. Construya las funciones pertinentes en PL y adecúe el programa en php que desarrolló. Haga los cambios que considere necesarios.*
- *Para el sistema de inventarios, actualice el inventario en un procedimiento de transacciones.*

Apéndice

A

Apéndice A

Ciertas variables especiales son pasadas a una función almacenada PL/pgSQL por defecto:

Tabla A.1: Variables especiales para los triggers

Variable	Descripción
NEW	Tipo de dato RECORD; es una variable que mantienen la nueva fila de la base de datos en las operaciones INSERT o UPDATE, en los desencadenados ROW.
OLD	Tipo de dato RECORD; es una variable que mantiene la fila actual de la base de datos en operaciones UPDATE o DELETE, en los desencadenados ROW.
TG_NAME	Nombre de tipo de dato; es una variable que contiene el nombre del procedimiento desencadenado (trigger) que se ha activado.
TG_LEVEL	Tipo de dato texto; una cadena de 'ROWó 'STATEMENT', dependiendo de la definición del procedimiento desencadenado.
TG_RELID	Tipo de dato oid; el ID del objeto de la tabla que ha provocado la invocación del procedimiento desencadenado.
TG_RELNAME	Tipo de dato nombre; el nombre de la tabla que ha provocado la activación del procedimiento desencadenado.
TG_NARGS	Tipo de dato entero; el numero de argumentos dado al procedimiento desencadenado en la sentencia CREATE TRIGGER.
TG_ARGV[..]	Tipo de dato matriz de texto; los argumentos de la sentencia CREATE TRIGGER. El índice comienza por cero, y puede ser dado en forma de expresión. Índices no validos dan lugar a un valor NULL.
TG_OP	operación a realizar, ésta puede ser UPDATE, INSERT o DELETE.
TG_WHEN	es cualquiera de BEFORE, AFTER o INSTEAD OF dependiendo de la definición del trigger
TG_TABLE_NAME	la segunda es la segunda variable representa el nombre de la tabla sobre la que se disparó el trigger

Hay más argumentos que se pasan a las funciones almacenadas por defecto.

B

Apéndice B

Comandos especiales a ser utilizados en los cursores:

Tabla B.1: Comandos especiales para los cursores

Variable	Descripción
FETCH	devuelve la siguiente fila desde el cursor a un destino, el cual puede ser una variable fila, registro o una lista de variables simples, separadas por comas.
CLOSE	cierra un cursor abierto, liberando recursos.
MOVE	posiciona un cursor sin devolver datosa. Como el comando FETCH, sin datos.
NEXT	regresa siguiente valor RECORD o ROW de un conjunto de datos.
PRIOR	regresa justo el valor anterior RECORD o ROW de un conjunto de datos.
FIRST	regresa primer valor RECORD o ROW de un conjunto de datos.
LAST	regresa último valor RECORD o ROW de un conjunto de datos.
ABSOLUTE count	regresa posición absoluta de un valor RECORD o ROW de un conjunto de datos.
RELATIVE count	regresa posición relativa de un valor RECORD o ROW de un conjunto de datos.
ALL	regresa todos los valores RECORD o ROW de un conjunto de datos.
FORWARD [count | ALL]	avanza puntero, un número determinado de valores o todos, en un valor RECORD o ROW de un conjunto de datos.
BACKWARD [count | ALL]	retrocede puntero, un número determinado de valores o todos, en un valor RECORD o ROW de un conjunto de datos.
SCROLL	el cursor es capaz de desplazarse hacia adelante o hacia atrás, en un valor RECORD o ROW de un conjunto de datos.

C

Apéndice C

Las tablas C.1 muestran los código de posibles errores definidos para PostgreSQL. (Algunos no son usados en la actualidad, pero están definidos en el SQL estándar).

Las clases error también son mostradas, por cada clase hay un código "estándar" que tiene al menos tres caracteres 000. Es usado para condiciones de error que no tienen código especifico asignado.

La columna "Condition Name" es el nombre de condición usado en PL/pgSQL. Los nombres son escritos en mayúscula o minúscula. PL/pgSQL no reconoce las advertencias (warning), como opuesto a error.

La tabla muestra las clases: 23, 24, 26, 27, 2D, 2F, 34, 38, 39, 3D, 3F, 40, P0 para SQLSTATE.

Tabla C.1: Listado de código de violación de restricción de integridad.

Código	Nombre de Condición
Class 23	Integrity Constraint Violation
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
Class 24	Invalid Cursor State
24000	invalid_cursor_state
Class 26	Invalid SQL Statement Name
26000	invalid_sql_statement_name
Class 27	Triggered Data Change Violation
27000	triggered_data_change_violation
Class 2D	Invalid Transaction Termination
2D000	invalid_transaction_termination
Class 2F	SQL Routine Exception
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted
2F004	reading_sql_data_not_permitted
Class 34	Invalid Cursor Name
34000	invalid_cursor_name
Class 38	External Routine Exception
38000	external_routine_exception
38001	containing_sql_not_permitted
38002	modifying_sql_data_not_permitted
38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
Class 39	External Routine Invocation Exception
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned
39004	null_value_not_allowed
39P01	trigger_protocol_violated
39P02	srf_protocol_violated
39P03	event_trigger_protocol_violated
Class 3D	Invalid Catalog Name
3D000	invalid_catalog_name
Class 3F	Invalid Schema Name
3F000	invalid_schema_name

Tabla C.2: Listado de código de violación de restricción de integridad II

Código	Nombre de Condición
Class 40	Transaction Rollback
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
Class P0	PL/pgSQL Error
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure

D

Apéndice D

La interfaz de programación del servidor (SPI) ofrece funciones en lenguaje C definidas por el usuario el ejecutar comandos SQL dentro de ellas. SPI es un conjunto de funciones de interfaz para simplificar el acceso al analizador, planificador y ejecutor en PostgreSQL; también se encarga de la gestión de la memoria.

> ***Nota:*** *Los lenguajes procedimentales disponibles proporcionan varios mecanismos para ejecutar comandos SQL desde los procedimientos. La mayoría de estas facilidades están basadas en SPI.*

Para evitar malentendidos, utilizamos el término "función" cuando hablemos de funciones de interfaz SPI y "procedimiento" para una función C de usuario que esté utilizando SPI.

Tenga en cuenta que si un comando llamado a través de SPI falla, el control no devuelve su procedimiento. Más bien, la transacción o sustracción en la que se ejecuta el procedimiento será revertida. (Recordar que estas convenciones sólo se aplican a los errores detectados dentro de las propias funciones SPI.) Es posible recuperar el control después de un error estableciendo su propia transacción en torno a las llamadas SPI que podrían fallar.

Las funciones SPI devuelven un resultado no negativo sobre el éxito (bien sea a través de un valor entero devuelto o en la variable global SPI_result, como se describe a continuación). En caso de error, devolverá un resultado negativo o NULL.

El código fuente que utiliza SPI debe incluir el archivo de cabecera executor/spi.h.

La función $SPI\_prepare$, prepara una sentencia, sin ejecutarla

Mientras que $SPI\_saveplan$, graba una sentencia preparada.

E

Apéndice E

Tabla E.1: Archivo utils/rel.h

Variable	Descripción
tg_relation->rd_att	descriptor de la relación entre tuplas.
tg_relation->rd_rel->relname	nombre de relación; el type no es char* sino NameData.
spi_getrelname(tg_relation)	obtiene un char* si necesita copiar el nombre.
tgname	nombre del trigger.
tgnargs	número de argumentos en tgargs.

Bibliografía

[1] Conway, Joseph E. (2009), *PL/R User's Guide - R Procedural Language. Boston.*

[2] Date, C. J. y Darwen, Hugh. (1996), *A Guide to the SQL Standard . California : Addison-Wesley Professional, 1996. ISBN: 978-0201964264.*

[3] Dominguez Ch. J. (2015), *MySQL: Triggers, Funciones y Procedimientos, IEASS, Editores, Venezuela.*

[4] Dominguez Ch. J. (2003), *Aplicaciones WEB con PHP 4 y PosgreSQL 7.1, IEASS, Editores, Venezuela.*

[5] Douglas, K. Douglas, S. (2005), *PostgreSQL A comprehensive guide to building, programming and administering Postgre SQL databases. (2nd. Edition).*

[6] Elmasri, R.; Navathe, S.B. (2002), *Fundamentos de Sistemas de Bases de Datos. 3ª Edición. Addison-Wesley.*

[7] Garcia-Molina, Jeffrey Ullman y Jennifer Widom. (2008), *Database systems: the complete book. Prentice-Hall.*

[8] López, José (2017), *Programación Funcional (html). Medium Com. Archivado desde el original el 1 de abril de 2018. Consultado el 1 de abril de 2018.*

[9] Mangones, E. C. (sf), *Auditoría en un ambiente de base de datos.*

[10] Mannino, M. (sf), *Administración de Base de Datos. México.*

[11] Manual de usuario de postgreSQL. (sf), *http://es.tldp.org/Postgresql-es/web/navegable/user/sql-createtable.html.*

[12] Pastor,O; García,R. (1994), *El Modelo Orientado a Objetos aplicado al Diseño e Implementación de entornos ORACLE; in the Proceedings of CUORE-94, Benalmádena Málaga (España).*

[13] Documentación oficial PostgreSQL. (s.f.), *http://www.postgresql.org.es.*

[14] Krosing, Hannu, Mlodgenski, Jim y Roybal, Kirk. (2013), *PostgreSQL Server Programming. Birmingham: Packt Publishing, ISBN 978-1-84951-698-3.*

[15] Laborda, Javier; Josep Galimany, Rosa María Pena, Antoni Gual (1985), *Software. Biblioteca práctica de la computación Barcelona: Ediciones Océano-Éxito, S.A..*

[16] Matthew, Neil y Stones, Richard. (2005), *Beginning databases with PostgreSQL, from novice to professional. 2nd edition. 2005.*

[17] Melton, Jim y Simon, Alan R. (1993), *Understanding the New SQL. s.l. : Morgan Kaufmann Publishers, ISBN: 9781558602458.*

[18] Momjian, Bruce. (2001), *PostgreSQL Introduction and Concepts. 2001.*

[19] Sánchez Andrés, María Ángeles (1996) (en español), *Programación estructurada y fundamentos de programación (1 edición). McGraw-Hill / Interamericana de España, S.A..*

[20] Silberschatz, A.; Korth, H.; Sudarshan,S. (2002), *Fundamentos de bases de datos (4.a ed.). Madrid: McGraw Hill.*

[21] The PostgreSQL Global Development Group. (2013), *PostgreSQL 9.3.0 Documentation. California, USA..*

[22] Pachón A. (2003), *PL/pgSQL - SQL Procedural Language The PostgreSQL Global Development Group, USA.*

[23] PostgreSQL 9.0 High Performance. Birmingham-Mumbai : Packt Publishing, (2010), *ISBN 978-1-849510-30-1.*

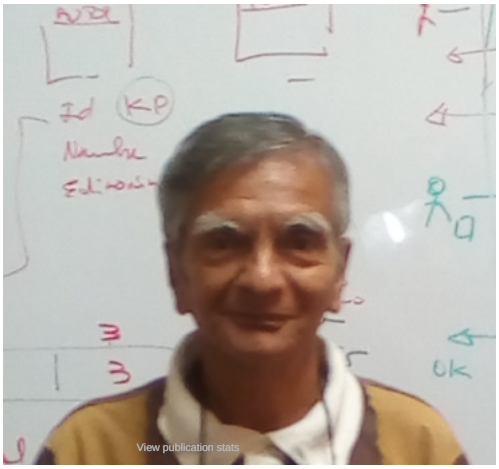
[24] Worsley, John C.; Drake, Joshua D. (2002), *Practical PostgreSQL. O'Reilly.*

[25] Zea Ordóñez, M. P. et Al (2017), *Administración de bases de datos con PostgreESQL, Editorial Área de Innovación y Desarrollo, S.L., España.*

Acerca del autor

Graduado en Física, Facultad de Ciencias, Universidad Nacional Autónoma de México (UNAM), Doctor en Ciencias de la Computación, Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas (IIMAS) UNAM. Especialista en Economía Matemática, Centro de Investigación y Docencia Económica, CIDE, México. Cursante del Doctorado en Minería de Datos, Modelos y Sistemas Expertos por la Universidad de Illinois en Urbana-Champaings (USA).

Ha sido profesor en la Facultad de Química y en la Facultad de Ingeniería, UNAM. Fue profesor en el Departamento de Ciencias Básicas, Universidad de Las Américas en Cholula, (UDLAP), Puebla, México y, durante seis años, profesor visitante en la Universidade Federal de Rio Grande do Sul (Brasil), profesor y jefe de sistemas postgrados de agronomía y veterinaria, Universidad Central de Venezuela (UCV), actualmente es profesor del Departamento de informática y del Departamento de Postgrado, Universidad Politécnica Territorial de Aragua (UPT Aragua), Venezuela.



Expositor y conferencista a nivel nacional e internacional.

Es asesor en mejora de procesos, gestión de proyectos, desarrollo de software corporativo en los sectores de servicios, banca, industria y gobierno.

El Dr. Domínguez es un especialista reconocido en base de datos, desarrollo de software y servidores en el área del software libre, así como un experto en LINUX DEBIAN.

En la actualidad orienta su trabajo a la creación y desarrollo de equipos de software de alto desempeño. Autor de múltiples artículos y libros sobre la materia.