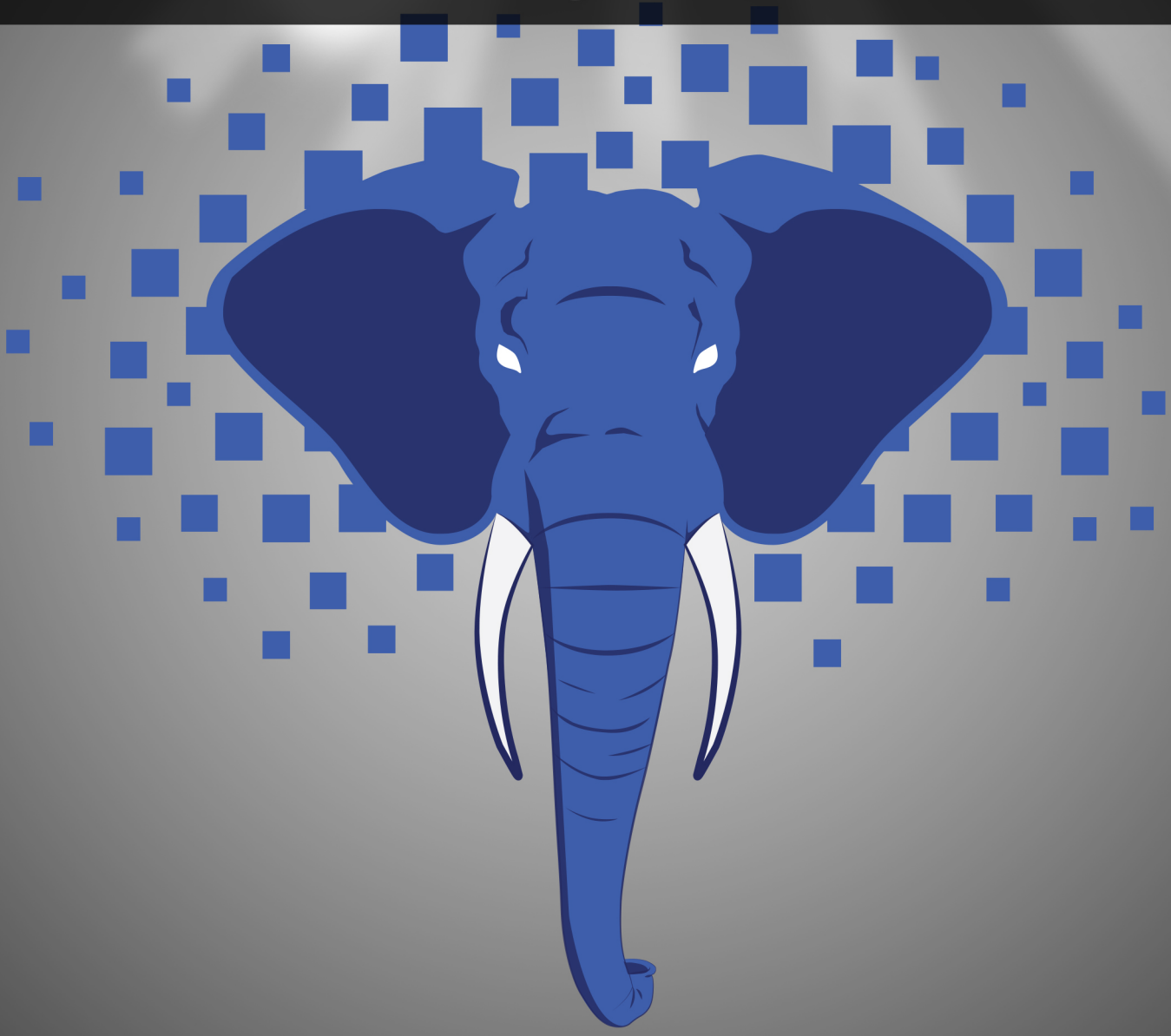
PL/pgSQL

y otros lenguajes procedurales en PostgreSQL



Anthony R. Sotolongo León
Yudisney Vazquez Ortíz

PL/pgSQL y otros lenguajes procedurales en PostgreSQL

Anthony R. Sotolongo León

Yudisney Vazquez Ortíz

Edición y corrección: Aida Elena Rodríguez Reinier
Ana Ortega Simón
Ilena Acuña Mendoza

Diseño de cubierta: Carlos Javier Hernández Rodríguez
Dirección de Comunicación Audiovisual, UCI

Diseño interior: Yudisney Vazquez Ortíz

Arbitraje: Julio Cesar Díaz Vera, UCI, Cuba
Gilberto Castillo Martínez, ETECSA, Cuba

Colaboradores: Daymel Bonne Solís
Marcos Luis Ortiz Valmaseda
Adalennis Buchillón Sorís

ISBN: 978-959-286-031-5

Impresión:

La Habana, República de Cuba
Ediciones Futuro
Universidad de las Ciencias Informáticas, Carretera a San Antonio de los Baños Km 2½. Rpto. Torrens, Boyeros, La Habana, Cuba
editorialef@uci.cu

Contenidos

Prefacio

Libro dirigido a estudiantes (de carreras afines con la Informática) y profesionales que trabajen con tecnologías de bases de datos, específicamente con el sistema de gestión de bases de datos PostgreSQL.

Surge en respuesta a la necesidad de contar con materiales orientados a la docencia, producción e investigación, con ejemplos variados, propuestas de ejercicios para aplicar los conocimientos adquiridos y en idioma español, que de forma práctica posibilite incorporar técnicas de programación haciendo uso del SQL (del inglés *Structured Query Language*) y de lenguajes procedurales para programar del lado del servidor de bases de datos. Elementos que en los libros existentes no son tratados en su conjunto, haciendo de la propuesta una opción a ser considerada.

Programar del lado del servidor de bases de datos brinda un grupo importante de opciones que pueden ser aprovechadas para ganar en rendimiento y potencia. Por lo que el propósito de este libro es que el lector pueda aplicar las opciones que brindan SQL y los lenguajes procedurales para el desarrollo de funciones, potenciando sus funcionalidades en la programación del lado del servidor de bases de datos.

Para facilitar la adquisición de los conocimientos tratados el libro brinda una serie de ejemplos basados en *Dell Store 2* (base de datos de prueba para PostgreSQL), proponiendo al final de los capítulos ejercicios en los que se deben aplicar los elementos abordados en ellos para su solución.

¿Qué cubre el libro?

El libro está dividido en 4 capítulos, donde en el:

- **Capítulo 1. Introducción a la programación del lado del servidor en PostgreSQL:** se realiza una introducción a la programación del lado del servidor, sus ventajas y cómo PostgreSQL permite dicha programación.
- **Capítulo 2. Programación de funciones en SQL:** se aborda cómo programar funciones en SQL; explicándose mediante ejemplos la sintaxis básica para su creación, empleo de parámetros y el retorno de valores.
- **Capítulo 3. Programación de funciones en PL/pgSQL:** se aborda cómo programar funciones en PL/pgSQL; explicándose su estructura, el trabajo con variables, sentencias, estructuras condicionales y de control, paso de mensajes, tratamiento de errores, depurado de código y cómo implementar cursores y disparadores.
- **Capítulo 4. Programación de funciones en lenguajes procedurales de desconfianza de PostgreSQL:** se realiza una breve introducción a los lenguajes procedurales de desconfianza que soporta PostgreSQL, abordándose en detalle PL/Python y PL/R; de los que se explica su compatibilidad con el gestor, la sintaxis básica para escribir funciones en ellos, el empleo de parámetros, la homologación de los tipos de datos de cada uno con el gestor y, cómo realizar con ellos el retorno de valores, la ejecución de consultas y la definición de disparadores.

¿Qué necesita para trabajar con el libro?

Para que el libro sea útil y puedan irse probando los ejemplos ilustrados, el lector debe:

- Tener conocimientos básicos de SQL, Python y R.
- Tener acceso a un servidor de bases de datos PostgreSQL 9.5 (versión en la que fueron ejecutadas todas las sentencias contenidas en el libro), de preferencia con privilegios administrativos; los ejecutables del gestor pueden ser descargados desde el sitio de PostgreSQL.

http://www.postgresql.org/download/

- Contar con un cliente de administración, todos los ejemplos fueron ejecutados en el psql, pero puede emplearse pgAdmin o cualquier otro.
- Contar con la base de datos *Dell Store 2*, sobre la que están basados los ejemplos de los capítulos y que puede ser accedida desde el sitio pgfoundry.org.

http://pgfoundry.org/frs/?group_id=1000150&release_id=376

¿Cómo dosificar los contenidos del libro?

La estructuración de los capítulos responde a la manera de abordar la programación del lado del servidor de bases de datos mediante el desarrollo de funciones aumentando el grado de complejidad, tanto en el uso de las potencialidades que ofrecen los lenguajes, como de los lenguajes en sí.

Se sugiere leer el capítulo introductorio para comprender las ventajas que ofrece la programación del lado del servidor y cómo PostgreSQL la permite. Aquel lector con dominio de SQL puede prescindir del estudio del capítulo 2 (aun cuando puede ser necesario revisarlo para conocer la sintaxis de creación de las funciones) y comenzar a estudiar el capítulo 3 y las potencialidades que ofrece PL/pgSQL para programar del lado del servidor de bases de datos. Y finalmente, tendrá las habilidades suficientes para comprender y poder hacer uso de los lenguajes procedurales de desconfianza PL/Python y PL/R abordados en el capítulo 4.

Convenciones

Para proveer una mayor legibilidad, a lo largo del libro se utilizan varios estilos de texto según la información que transmiten:

- Código: el código de las sintaxis básicas, funciones y consultas utilizadas se escriben en Consolas a 9 puntos, resaltándose con negritas y en mayúsculas (cuando por sintaxis no sea incorrecto) las palabras reservadas del lenguaje; el siguiente es un ejemplo de bloque de código:

```
CREATE FUNCTION eliminar_estudiantes() RETURNS void AS
$$
   DELETE FROM estudiante WHERE anno = 5;
$$ LANGUAGE sql;
```

- Los ejemplos, generalmente código de consultas o funciones implementadas, se enumeran y enmarcan de la forma:

Ejemplo 1: Función SQL que elimina los estudiantes de quinto año

```
CREATE FUNCTION eliminar_estudiantes() RETURNS void AS
$$
   DELETE FROM estudiante WHERE anno = 5;
$$ LANGUAGE sql;
```

- Notas: acotaciones sobre lo que se esté discutiendo se escribirán a 9 puntos y enmarcadas, de la forma:

Nota Ejemplos de funciones internas y funciones escritas en C pueden encontrarse en la Documentación Oficial de PostgreSQL en las secciones *Internal Functions* y *C-Language Functions*

http://www.postgresql.org/docs/9.5/static/extend.html

Sobre la presente edición

En esta segunda edición se incorporan contenidos relacionados con la implementación de cursores, el depurado de código y el manejo de errores; se añaden nuevos ejercicios a los capítulos 1 y 3 y; además, se agrega una sección con posibles soluciones a los ejercicios propuestos para que los lectores puedan autoevaluarse cuando los realicen.

Autores

Anthony R. Sotolongo León (asotolongo@gmail.com). Fue profesor de bases de datos en la Universidad de las Ciencias Informáticas y miembro del Grupo de Usuarios PostgreSQL de Cuba; tiempo en el que organizó y coordinó eventos relacionados con el gestor en el país en los que impartió charlas y postgrados; fue coordinador del Diplomado en Tecnologías de Bases de Datos PostgreSQL, impartido en la Universidad, formando parte de su Comité Académico durante 3 ediciones. Actualmente es consultor de tecnologías de bases de datos en ARKADIOS, Chile.

Yudisney Vazquez Ortíz (yvazquezo@uci.cu, yvazquezo@gmail.com). Profesora de bases de datos en la Universidad de las Ciencias Informáticas. Miembro y fundadora del Grupo de Usuarios PostgreSQL de Cuba. Organiza y coordina eventos relacionados con el gestor en el país en los que ha impartido postgrados. Diseñó el Diplomado en Tecnologías de Bases de Datos PostgreSQL, impartido en la Universidad, formando parte del Comité Académico durante sus 5 ediciones.

Errores

De encontrarse cualquier error se agradecería su comunicación a los autores.

Capítulo 1
Introducción a la programación del lado del servidor en PostgreSQL

Metas:

- Enunciar las ventajas de la programación del lado del servidor de bases de datos
- Conocer los tipos de Funciones Definidas por el Usuario soportados por PostgreSQL
- Explicar los usos de la lógica de negocio cuando se realiza del lado del servidor de bases de datos

INTRODUCCIÓN A LA PROGRAMACIÓN DEL LADO DEL SERVIDOR EN POSTGRESQL

1.1 Introducción a las Funciones Definidas por el Usuario

Las bases de datos relacionales son el estándar de almacenamiento de las aplicaciones informáticas. Las operaciones con dichas bases de datos suelen realizarse, comúnmente, mediante sentencias SQL, lenguaje por defecto para interactuar con las mismas.

Utilizar los gestores de bases de datos relacionales puramente para almacenar datos es restringir sus potencialidades de trabajo, puesto que brindan otras opciones además de ser un contenedor para el almacenamiento. Ejemplo de ellas son los procedimientos almacenados y los disparadores, parte de un grupo importante de funcionalidades que se pueden potenciar programando del lado del servidor de bases de datos.

El manejo de los datos solamente con Lenguaje de Definición y Manipulación de Datos (DDL y DML) tiene limitaciones, pues no permiten operaciones como el control de flujo o la utilización de variables para retener un dato; lo que impide realizar operaciones de lógica de negocio del lado del servidor de bases de datos, con las consecuentes ventajas que esto puede acarrear.

PostgreSQL, como gestor de bases de datos relacional, permite la programación del lado del servidor desde sus inicios, mejorándose considerablemente su soporte a partir de la versión 7.2 y agregándose otros lenguajes (llamados lenguajes de desconfianza), además del conocido estándar SQL. Paulatinamente se han ido incorporando mejoras a la programación del lado del servidor y hoy es una verdadera potencialidad para el desarrollo de aplicaciones que utilizan PostgreSQL como gestor de bases de datos.

PostgreSQL permite el desarrollo de la programación de lógica de negocio del lado del servidor mediante las Funciones Definidas por el Usuario, llamadas en otros gestores "procedimientos almacenados". Estas funciones se entienden como el conjunto agrupado de operaciones que se ejecutan del lado del servidor, que pueden derivar en acciones sobre los datos y, que pudieran utilizar otras características del gestor como los tipos de datos, operadores personalizados, reglas, vistas, etc. Las funciones definidas por el usuario se clasifican en cuatro tipos:

- Función en SQL: ejecuta puramente operaciones SQL.
- Función en lenguaje procedural: ejecuta operaciones empleando lenguajes de tipo procedural como PL/pgSQL, PL/Python, etc.

- Función interna: ejecuta operaciones en lenguaje C que están enlazadas directamente al servidor PostgreSQL, lo que implica que están predefinidas dentro del gestor.
- Función en lenguaje C: ejecuta operaciones, escritas en el lenguaje C o compatible, que pueden ser cargadas a PostgreSQL dinámicamente bajo demanda.

Este libro se centrará en los dos primeros tipos de funciones mencionadas, debido a que son la forma más común de programar del lado del servidor en PostgreSQL.

Nota

Ejemplos de funciones internas y funciones escritas en C pueden encontrarse en la Documentación Oficial de PostgreSQL en las secciones *Internal Functions* y *C-Language Functions*

http://www.postgresql.org/docs/9.5/static/extend.html

Los siguientes ejemplos ilustran acciones que se pueden realizar al emplear funciones definidas por el usuario.

Ejemplo 1: Función SQL que elimina los estudiantes de quinto año

```
CREATE FUNCTION eliminar_estudiantes() RETURNS void AS
$$
   DELETE FROM estudiante WHERE anno = 5;
$$ LANGUAGE sql;
```

Ejemplo 2: Función en PL/pgSQL que suma 2 valores

```
CREATE FUNCTION sumar(num1 integer, num2 integer) RETURNS integer AS
$$
BEGIN
   RETURN $1 + $2;
END;
$$ LANGUAGE plpgsql;
```

Note que entre ambos tipos de funciones hay ciertas diferencias de estructura que se analizarán, con mayor nivel de detalle, en los capítulos 2 y 3 respectivamente.

Una vez creada la función, y almacenada en el gestor de bases de datos, para su empleo se puede invocar de 2 formas:

- Como un elemento del SELECT, con la sintaxis:

  SELECT *nombre_función*([*parámetro_función1*, …]);

- Desde la cláusula FROM, con la sintaxis:

```
SELECT * FROM nombre_función( [ parámetro_función1, … ] );
```

Los siguientes ejemplos ilustran cómo realizar invocaciones de funciones.

Ejemplo 3: Invocación de la función version de PostgreSQL

```
postgres=# SELECT version();
                           version
-------------------------------------------------------------
PostgreSQL 9.5.3, compiled by Visual C++ build 1800, 64-bit
(1 fila)
```

Nota *version*() es una función interna de PostgreSQL; puede encontrarse otras funciones internas en la Documentación Oficial en el capítulo *Functions and Operators*

http://www.postgresql.org/docs/9.5/static/functions.html

Ejemplo 4: Invocación de la función sumar definida en el ejemplo 2

```
postgres=# SELECT * FROM sumar(1, 2);
 sumar
-------
     3
(1 fila)
```

En los capítulos siguientes se utilizan ambos tipos de invocaciones en variados escenarios, explicándose las diferencias entre ellas con mayor detalle en el epígrafe "Retorno de conjunto de valores" en la página 24.

Nota Si dentro de una función se realiza el llamado a otra función, basta con colocar su nombre sin necesidad de utilizar el comando SELECT

1.2 Ventajas de utilizar la programación del lado del servidor de bases de datos

Tener el código de la lógica de negocio en el servidor de bases de datos puede ir en contra de algunos modelos de desarrollo de aplicaciones, como por ejemplo el Modelo Tres Capas, donde se le asigna a cada capa una de las siguientes actividades:

- Capa de datos: base de datos.

- Capa de negocio: capa intermedia, lógica de negocio, operaciones, etc.
- Capa de presentación: interfaz al usuario.

El punto de vista anterior se cumple si se logra que las capas coincidan con un lugar físico en el desarrollo del sistema, es decir, la capa de datos con el gestor de bases de datos, la capa de negocio con el código de las aplicaciones y la capa de presentación con la interfaz que se le presenta al usuario.

Pero si se ven las capas como lógicas en el desarrollo del sistema, puede describirse como la capa de datos al gestor de bases de datos, como la capa de negocio el código del negocio (que puede estar en el gestor de bases de datos), y como la capa de presentación a la interfaz para el usuario.

Viéndolo desde este último punto de vista, la programación del lado del servidor brinda excelentes posibilidades de desarrollo y, por tanto, varias ventajas como las que se describen en el libro *PostgreSQL Server Programming* de los autores Hannu Krosing, Jim Mlodgenski y Kirk Roybal, las que se resumen en:

- Aumento del rendimiento: si la lógica de negocio se realiza del lado del servidor se impide que los datos viajen de la base de datos a la aplicación, evitando la latencia por esta operación.
- Fácil mantenimiento: si la lógica de negocio se modifica por algún motivo, los cambios se realizan en la base de datos central y pueden ser hechos fácilmente mediante el Lenguaje de Definición de Datos (DDL) para actualizar las funciones implicadas en ellos.
- Simplicidad para garantizar la seguridad en la lógica de negocio: a las funciones se les pueden definir privilegios de acceso a través del Lenguaje de Control de Datos (DCL) y se evita que los datos viajen por la red.

Otra ventaja es que evita la necesidad de reescribir código de negocio; por ejemplo, si se tuviera una base de datos de la que consumen varias aplicaciones escritas en diferentes lenguajes como Pascal, Java, PHP y Python, se tuviera que realizar una operación para la inserción y actualización de datos y, dicha operación se encuentra del lado del servidor de bases de datos, solamente se debe solicitar su ejecución desde las distintas aplicaciones.

1.3 Usos de la lógica de negocio del lado del servidor

Las ventajas de programar del lado del servidor son aprovechadas, fundamentalmente, en operaciones que de ser realizadas mediante consultas SQL resultarían engorrosas sino imposibles, destacando las mencionadas en las secciones siguientes.

Transacciones que incluyen varias operaciones

Es común ejecutar varias sentencias interrelacionadas que realizan operaciones sobre los datos, sobre todo para evitar el tráfico en la red. Por ejemplo, si se desea actualizar el género de una canción de la cual se conoce su nombre y, además, devolver su autor, se pudiera implementar una función como la mostrada en el ejemplo 5.

Ejemplo 5: Función que actualiza el género de una canción y muestra su autor

```
CREATE FUNCTION actualizar_genero(nomb_c varchar, genero_c varchar) RETURNS
 varchar AS
$$
DECLARE
   id integer;
   autor_c varchar;
BEGIN
   UPDATE cancion SET genero = $2 WHERE nombre_cancion = $1;
   SELECT idcancion INTO id FROM cancion WHERE nombre_cancion = $1;
   SELECT autor INTO autor_c FROM cancion WHERE idcancion = id;
   RETURN autor_c;
END;
$$ LANGUAGE plpgsql;
```

Note que se realizan varias operaciones para obtener el resultado deseado que, de realizarse a nivel de aplicación, implicaría un mayor tráfico de datos entre esta y el servidor, incidiendo en el rendimiento de la misma.

Auditoría de datos

La auditoría de datos permite chequear las operaciones realizadas sobre los datos con el objetivo de detectar anomalías o acciones no deseadas o incorrectas. Uno de los usos que más se le da a la lógica de negocio del lado del servidor es para realizar estas auditorías. Para efectuar una operación de este tipo, donde se lleve el registro de los datos modificados o eliminados, se pueden utilizar disparadores; ejemplos de ello son los módulos *Pgaudit* y *Audit-triggers*, que empaquetan un conjunto de funciones y disparadores, posibilitan la detección de cambios realizados sobre los datos y, son configurables para trabajar en las tablas deseadas.

https://github.com/jcasanov/pg_audit

https://github.com/2ndQuadrant/audit-trigger

Consumo de características de otros lenguajes de programación

Existen operaciones que los lenguajes nativos brindados por el gestor no realizan o su posibilidad de hacer alguna actividad es compleja. Una de las potencialidades que más se puede explotar con la programación del lado del servidor es el consumo o ejecución de funciones de otros lenguajes.

Por ejemplo, si se necesitara generar un gráfico de pastel, pudiera emplearse el lenguaje R (especializado en actividades estadísticas) como muestra la función del ejemplo 6 que, una vez invocada, genera un gráfico como el de la figura 1.

Ejemplo 6: Empleo de PL/R para generar un gráfico de pastel

```
CREATE FUNCTION generar_pastel(nombre text, vector integer[], texto text)
 RETURNS integer AS
$$
   png(paste(nombre,"png",sep="."))
   pie(vector,header=TRUE,col=rainbow(length(vector)),main=texto,labels=
    vector)
   dev.off()
$$ LANGUAGE plr;

-- Invocación de la función
postgres=# SELECT generar_pastel('migraficapie',array[3,6,7,9],'Ejemplo de
           pie');
 generar_pastel
----------------
(1 fila)
```

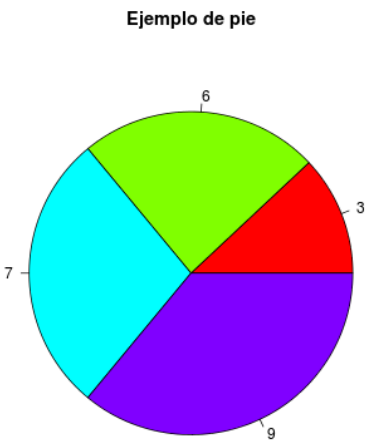


Figura 1: Gráfica generada con PL/R

Exportación de datos

PostgreSQL permite exportar datos, por ejemplo, si se necesitara exportar el resultado de una consulta a un formato CSV, se puede implementar una función con PL/pgSQL que realice dicha actividad como muestra el ejemplo siguiente.

Ejemplo 7: Función que exporta el resultado de una consulta a un fichero CSV

```
CREATE FUNCTION salvar_csv() RETURNS boolean AS
$$
BEGIN
   COPY (SELECT * FROM empleado) TO '/tmp/archivo.csv' WITH CSV;
   RETURN true;
END;
$$ LANGUAGE plpgsql;
```

Los ejemplos anteriores evidencian que, sin lugar a dudas, programar del lado del servidor de bases de datos empleando funciones definidas por el usuario brinda un grupo importante de opciones que pueden ser aprovechadas para ganar en rendimiento y potencia. De ahí que el propósito de este libro sea analizar, con mayor nivel de detalle, la forma de crear funciones utilizando SQL y lenguajes procedurales para potenciar sus funcionalidades desde PostgreSQL.

1.4 Modelo de datos para el trabajo con el libro

El modelo de datos mostrado en la figura 2 se corresponde con la base de datos *Dell Store 2*, disponible en el proyecto "Colección de bases de datos de ejemplos para PostgreSQL" y que puede ser accedida desde el sitio pgfoundry.org. Dicho modelo será el empleado, como base de los ejemplos, en los siguientes capítulos.

http://pgfoundry.org/frs/?group_id=1000150&release_id=376

1.5 Resumen

La programación del lado del servidor de bases de datos es una opción para el desarrollo de los sistemas informáticos. Su atractivo radica en que ofrece un grupo de ventajas entre las que destacan el ganar en rendimiento, seguridad y facilidad de mantenimiento, así como el evitar la reescritura de código.

PostgreSQL, como sistema de gestión de bases de datos relacional, permite la programación del lado del servidor, principalmente mediante el empleo de Funciones Definidas por el Usuario.

En el capítulo se mostraron algunos ejemplos básicos del uso de las funciones definidas por el usuario que, a medida que se avance en la lectura de este libro podrán comprenderse mejor y utilizarse para potenciar las funcionalidades de este gestor.

1.6 Para hacer

1. Mencionar los tipos de funciones definidas por el usuario que soporta PostgreSQL.

2. Mencionar las ventajas que ofrece programar del lado del servidor de bases de datos.

3. Enumerar los principales usos que se le dan a la lógica del negocio cuando se realiza del lado del servidor de bases de datos.

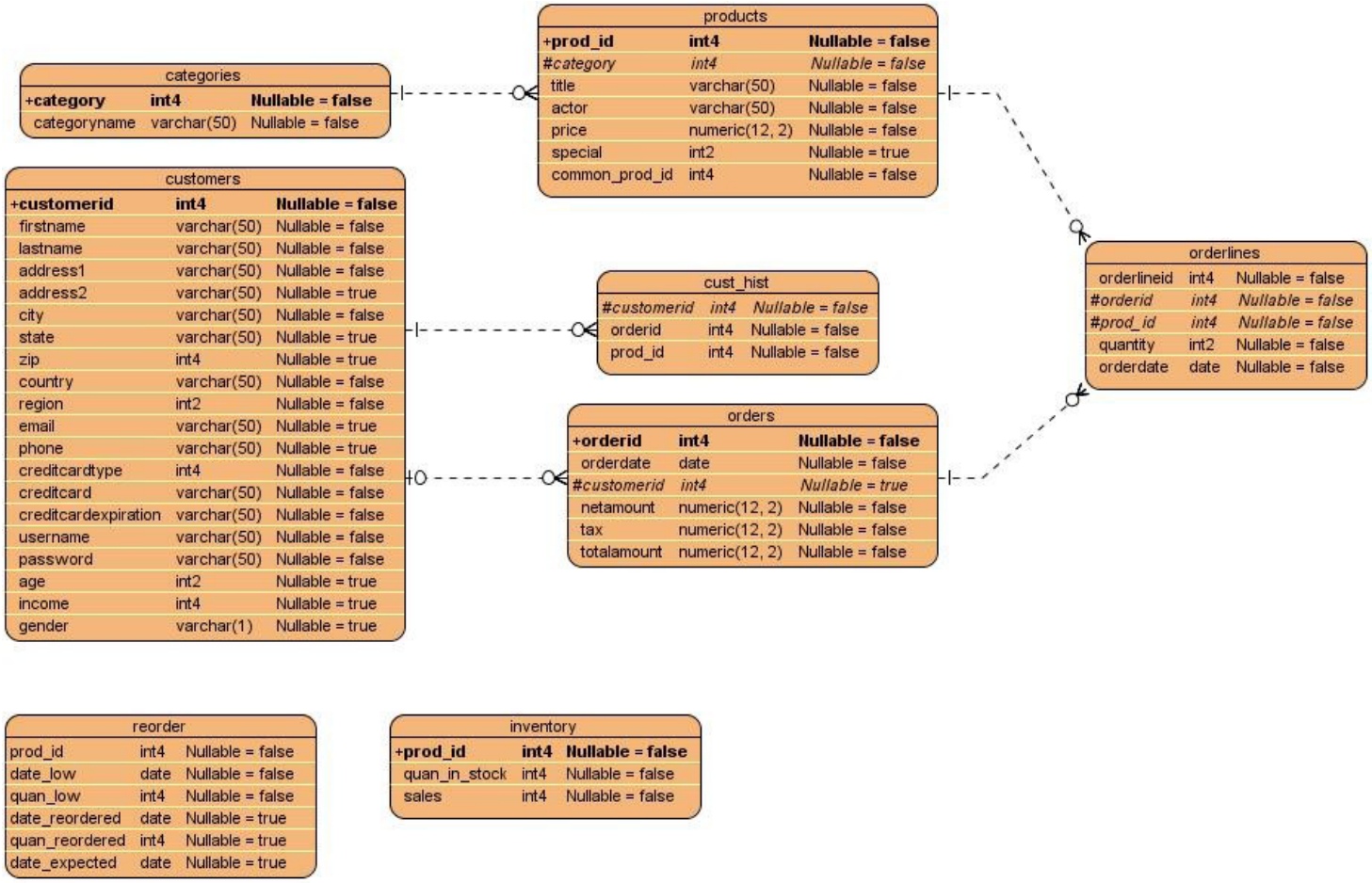


Figura 2: Modelo de datos de la base de datos Dell Store 2

Capítulo 2
Programación de funciones en SQL

Metas:

- Dominar la sintaxis para la definición de funciones SQL
- Emplear las cláusulas necesarias para la creación de funciones SQL
- Utilizar parámetros para el trabajo con funciones SQL
- Retornar valores básicos, compuestos o conjuntos, como resultado de la ejecución de funciones SQL

PROGRAMACIÓN DE FUNCIONES EN SQL

2.1 Introducción a las funciones SQL

En el capítulo se realiza una breve introducción a cómo programar funciones en SQL; explicándose, mediante ejemplos, la sintaxis básica para la creación de funciones, el empleo de parámetros y los tipos existentes para el retorno de valores.

Las funciones son parte de la extensibilidad que provee PostgreSQL a los usuarios; siendo el código escrito en SQL uno de los más sencillos de añadir al gestor. Las funciones SQL ejecutan una lista arbitraria de sentencias SQL separadas por punto y coma (;) que retornan el resultado de la última consulta de dicha lista; por lo que cualquier colección de consultas SQL puede ser empaquetada y definida como una función.

Además de consultas de tipo SELECT se pueden incluir consultas de modificación de datos (INSERT, UPDATE, DELETE), así como cualquier otro comando SQL, excepto aquellos de control de transacciones (COMMIT, SAVEPOINT, etc.) y algunos de utilidad (VACUUM, etc.).

Nota La función SQL es analizada antes de su ejecución, por lo que si contiene comandos que alteran el catálogo del sistema (como CREATE) los efectos no serán visibles durante el análisis del resto de los comandos empaquetados, pudiendo generarse errores en tiempo de ejecución de existir consultas que dependan de dichas creaciones; en tales escenarios se sugiere utilizar un lenguaje procedural, como los que se analizarán en capítulos posteriores

2.2 Sintaxis para la definición de funciones SQL

Para la definición de funciones SQL se emplea el comando CREATE FUNCTION de la forma:

```
CREATE [ OR REPLACE ] FUNCTION nombre( [ parámetro1 ] [ ,... ] ) [ RETURNS
 tipo_retorno | RETURNS TABLE( nombre_columna tipo_columna [ ,... ] ) ] AS
$$
   cuerpo de la función…
$$ LANGUAGE sql;
```

Nota Para analizar la sintaxis ampliada de la definición de una función, ver en la Documentación Oficial en el capítulo *Reference*, el epígrafe *SQL Commands*

http://www.postgresql.org/docs/9.5/static/sql-commands.html

El empleo del comando CREATE FUNCTION requiere tener en cuenta los siguientes elementos:

- Para definir una nueva función el usuario debe tener privilegio de uso (USAGE) sobre el lenguaje, SQL en este caso.
- Si el esquema es incluido, entonces la función es creada en el esquema especificado, de otra forma, es creada en el esquema actual.
- El nombre de la nueva función no debe coincidir con ninguna existente con los mismos parámetros y en el mismo esquema, en ese caso la existente se reemplaza por la nueva siempre que sea incluida la cláusula OR REPLACE durante la creación de la función.
- Funciones con parámetros diferentes pueden compartir el mismo nombre.

CREATE FUNCTION requiere que el cuerpo de la función sea escrito como una cadena constante. Note que en la sintaxis detallada en la Documentación Oficial de PostgreSQL para la definición de la función se emplean comillas simples ('), pero es más conveniente usar el doble dólar ($$) o cualquier otro tipo de acotado, ya que, de usarse las primeras, si estas o las barras invertidas (\) son requeridas en el cuerpo de la función deben entrecomillarse también, haciendo engorroso el código.

Nota

Para analizar en detalle el acotado a emplear en el cuerpo de una función remitirse a las secciones *String Constants* y *Dollar-quoted String Constants* del capítulo *SQL Syntax* en la Documentación Oficial; en este libro se empleará el $$ para acotar el cuerpo de las funciones

http://www.postgresql.org/docs/9.5/static/sql-syntax-lexical.html#SQL-SYNTAX-CONSTANTS

La función mostrada en el ejemplo 8 elimina aquellos clientes que tienen 18 años o menos; note que no se necesita que la función retorne ningún valor, por tanto, el tipo de retorno definido es VOID; para mayor detalle sobre los tipos de retorno ver el epígrafe "2.4 Retorno de valores" en la página 21.

Ejemplo 8: Función que elimina clientes de 18 años o menos

```
CREATE FUNCTION eliminar_clientesmenores() RETURNS void AS
$$
   DELETE FROM customers WHERE age <= 18;
$$ LANGUAGE sql;
```

Para reemplazar la definición actual de una función existente se especifica la cláusula OR REPLACE, con la que:

- Se reemplaza la definición de una función existente con el mismo nombre, parámetros y en el mismo esquema especificado.
- No es posible cambiar el nombre o los tipos de los parámetros de la definición de la función, si se intenta, lo que realmente se está haciendo es crear una función distinta.
- No es posible cambiar el tipo de retorno de la función existente; para hacerlo se debe eliminar y volver a crear la función.
- No se cambian el propietario y los permisos de la función.
- Si se elimina y se vuelve a crear la función, esta nueva función no es el mismo objeto que la anterior, por tanto, tendrán que eliminarse las reglas, vistas y disparadores existentes que hacían referencia a la antigua función.

Por ejemplo, la función mostrada en el ejemplo 9 reemplaza la función anterior *eliminar_clientesmenores*() pues tiene el mismo nombre y parámetros (en este caso ninguno), especificando ahora que se borrarán aquellos clientes menores de 20 en lugar de 18 años. Note que de no especificarse la cláusula OR REPLACE, al intentar definirse la función PostgreSQL arroja un error enunciando que ya existe una con el mismo nombre y parámetros especificados.

Ejemplo 9: Reemplazo de función que elimina clientes menores de 20 en lugar de 18 años

```
CREATE OR REPLACE FUNCTION eliminar_clientesmenores() RETURNS void AS
$$
   DELETE FROM customers WHERE age < 20;
$$ LANGUAGE sql;
```

Pero, si además de eliminar los clientes menores de 20 años se quisiera mostrar la cantidad de clientes restantes, como a la función existente no se le puede cambiar el tipo de retorno, se debe eliminar e implementar una nueva con las especificaciones requeridas, como se muestra en el ejemplo 10.

Ejemplo 10: Reemplazo de la función eliminar_clientesmenores por otra que elimina los clientes menores de 20 años y retorna la cantidad de clientes que quedan registrados en la base de datos

```
-- Eliminar la función anterior
DROP FUNCTION eliminar_clientesmenores();
```

```
-- Definir la nueva función
CREATE FUNCTION eliminar_clientesmenores() RETURNS bigint AS
$$
   DELETE FROM customers WHERE age < 20;
   SELECT count(*) FROM customers;
$$ LANGUAGE sql;

-- Invocación de la función
dell=# SELECT eliminar_clientesmenores();
 eliminar_clientesmenores
--------------------------
                   19480
(1 fila)
```

Note que la diferencia de esta función con la implementada en el ejemplo 9 radica en que se agrega una sentencia SELECT (para retornar la cantidad de clientes restantes) después de la eliminación, con el consecuente cambio del tipo de dato de retorno de la función.

2.3 Parámetros de funciones SQL

Para que las funciones sean realmente útiles, generalmente requieren de datos de entrada, conocidos como "parámetros". Los parámetros de una función especifican aquellos valores que el usuario define sean empleados para su procesamiento posterior en el cuerpo de la misma, con el fin de obtener el resultado esperado. Se definen dentro de los paréntesis colocados detrás del nombre de la función con la forma:

```
[ modo_parámetro ] [ nombre_parámetro ] tipo_dato_parámetro
```

Donde:

- *modo_parámetro* puede ser de 4 tipos:
  - IN: modo por defecto, especifica que el parámetro es de entrada, o sea, forma parte de la lista de parámetros con que se invoca a la función y que son necesarios para el procesamiento definido en ella.
  - OUT: parámetro de salida, forma parte del resultado de la función y no se incluye en su invocación.
  - INOUT: parámetro de entrada/salida, puede ser empleado indistintamente para que forme parte de la lista de parámetros de entrada y, luego, del resultado.
  - VARIADIC: parámetro de entrada con un tratamiento especial, que permite definir un arreglo para especificar que la función acepta un conjunto variable de parámetros, los que lógicamente deben ser del mismo tipo.

- *nombre_parámetro* es una cadena de caracteres que identifica al parámetro en el cuerpo de la función.
- *tipo_dato_parámetro* puede ser uno de los tipos de datos definidos en el estándar SQL o por el usuario.

Nota

Para mayor detalle en el empleo de parámetros de tipo VARIADIC remitirse a la Documentación Oficial en la sección *SQL Functions with Variable Numbers of Arguments*, del capítulo *Extending SQL*

http://www.postgresql.org/docs/9.5/static/xfunc-sql.html#XFUNC-SQL-VARIADIC-FUNCTIONS

Los parámetros pueden ser referenciados en el cuerpo de la función usando sus nombres (a partir de la versión 9.2 del gestor) o números. Para ello se debe tener en cuenta que:

- Para usar el nombre, se debe previamente haber especificado este en la declaración de los parámetros cuando se definió la función.
- Si el parámetro es nombrado igual que alguna columna empleada en el cuerpo de la función puede causar problemas de precedencia. No obstante, se recomienda con el fin de evitar esta ambigüedad: (1) emplear parámetros con nombres distintos a las columnas de las tablas que puedan utilizarse en la función, (2) definir un alias distinto para la columna de la tabla que entra en conflicto con el parámetro o, (3) calificar los parámetros con el nombre de la función.
- Los parámetros pueden ser referenciados con números de la forma $*número*, refiriéndose $1 al primer parámetro definido, $2 al segundo, y así sucesivamente; lo cual funciona se haya, o no, nombrado el parámetro.
- Si el parámetro es de tipo compuesto se puede emplear la notación *parámetro.campo* para acceder a sus atributos.

Los ejemplos 11, 12, 13, 14 y 15 ilustran los elementos mencionados.

Note, en el ejemplo 11, que en la sentencia de inserción se hace referencia a los parámetros mediante sus nombres, especificados en la definición de la función.

Ejemplo 11: Empleo de parámetros usando sus nombres en una sentencia INSERT

```
CREATE FUNCTION insertar_categoria(categoria integer, nombre varchar) RETURNS
 void AS
$$
   INSERT INTO categories VALUES (categoria, nombre);
$$ LANGUAGE sql;
```

Note, en el ejemplo 12, que puede omitirse el nombre de los parámetros y solamente especificar el tipo de dato, en cuyo caso se referencian mediante el signo dólar ($) y el número que ocupan en el listado de parámetros definidos. Esta forma, aunque válida, se evita porque hace ilegible el código en funciones más complejas. Para evitar errores con los nombres de los parámetros se debe borrar antes la función creada en el ejemplo 11.

Ejemplo 12: Empleo de parámetros utilizando su numeración en una sentencia INSERT

```
CREATE FUNCTION insertar_categoria(integer, varchar) RETURNS void AS
$$
   INSERT INTO categories VALUES ($1, $2);
$$ LANGUAGE sql;
```

Observe, en el ejemplo 13, que se definen los parámetros *category* y *categoryname* con los mismos nombres que los atributos de la tabla *categories* y que, para evitar la ambigüedad, se califican con el nombre de la función y que, al igual que en el ejemplo anterior, se debe borrar la función existente para evitar errores con los nombres.

Ejemplo 13: Empleo de parámetros con iguales nombres que columnas de tabla usada en la función

```
CREATE FUNCTION insertar_categoria(category integer, categoryname varchar)
 RETURNS void AS
$$
   INSERT INTO categories VALUES (insertar_categoria.category,
    insertar_categoria.categoryname);
$$ LANGUAGE sql;
```

En el ejemplo 14 aprecie que se define el parámetro *categoria* que es del tipo compuesto *categories* (PostgreSQL crea para cada tabla un tipo compuesto asociado), y que para utilizarlo en la inserción de un nuevo producto se califica accediendo a su elemento *category*. Note que la llamada debe realizarse como parte de una consulta a *categories*, de la misma forma que se realiza en el "Ejemplo 19" en la página 23.

Ejemplo 14: Empleo de parámetros de tipo compuesto

```
CREATE FUNCTION insertar_producto(id integer, categoria categories, titulo
 varchar, actor varchar, precio float, especial integer, id_comun integer)
 RETURNS void AS
$$
   INSERT INTO products VALUES (id, categoria.category, titulo, actor, precio,
    especial, id_comun);
$$ LANGUAGE sql;
```

En el ejemplo 15 se puede apreciar el empleo de los parámetros de salida *first* y *last* para guardar el nombre y apellidos, respectivamente, del cliente pasado por parámetro. De

no haberse declarado, dichos parámetros, se debería especificar el tipo de retorno de la función haciendo uso de la cláusula RETURNS.

Ejemplo 15: Empleo de parámetros de salida

```
CREATE FUNCTION mostrar_nombrecompleto(id integer, OUT first varchar, OUT last
 varchar) AS
$$
   SELECT firstname, lastname FROM customers WHERE customerid = id;
$$ LANGUAGE sql;

-- Invocación de la función desde la lista del SELECT
dell=# SELECT mostrar_nombrecompleto(20);
 mostrar_nombrecompleto
------------------------
(IAYPUX,YELMUQZEHW)
(1 fila)

-- Invocación de la función desde la cláusula FROM
dell=# SELECT first, last FROM mostrar_nombrecompleto(20);
 firstname |  lastname
-----------+------------
IAYPUX     | YELMUQZEHW
(1 fila)
```

2.4 Retorno de valores

La cláusula RETURNS indica el retorno de la función, especificando el tipo de dato que debe devolver una vez que se ejecute; que puede ser:

- Uno de los tipos de datos definidos en el estándar SQL o por el usuario, por ejemplo, VARCHAR para devolver la ciudad en que vive el cliente "Brian Daniel Vázquez López", INTEGER para devolver la edad del cliente "Llilian Pozo Ortiz", VOID para aquellas funciones que no retornen un valor usable, un tipo de dato compuesto (ejemplo PRODUCTS para devolver una tupla de la tabla *products* de la base de datos), RECORD para retornar una fila resultante de un subconjunto de las columnas de una tabla o de concatenaciones entre tablas, etc.
- Un conjunto de tuplas: mediante la especificación del tipo de retorno "SETOF *algún_tipo*" o, de forma equivalente, declarando el tipo de retorno "TABLE (*columnas*)", en cuyo caso todas las tuplas de la última consulta son retornadas.
- El valor nulo: en caso de que la consulta no retorne ninguna fila.

A menos que la función sea declarada con el tipo de retorno VOID, la última sentencia de su cuerpo debe ser un SELECT, o un INSERT, UPDATE o DELETE que incluya la cláusula RETURNING con la que devuelvan lo que sea especificado en el tipo de retorno de la función; de lo contrario se arrojará un error en tiempo de ejecución.

Retorno de tipos de datos básicos

Las funciones SQL más simples no tienen parámetros o devuelven un tipo de dato básico, retorno que se puede realizar (como se muestra en los ejemplos 16 y 17):

- Utilizando una consulta SELECT como la última del bloque de sentencias SQL de la función.
- Empleando la cláusula RETURNING como parte de las consultas INSERT, UPDATE o DELETE.

Ejemplo 16: Función que dado el identificador de la orden devuelve su monto total

```
CREATE FUNCTION monto_total(id integer) RETURNS numeric AS
$$
   SELECT totalamount FROM orders WHERE orderid = id;
$$ LANGUAGE sql;

-- El mismo ejemplo empleando enumeración de parámetros
CREATE FUNCTION monto_total_enum_param(integer) RETURNS numeric AS
$$
   SELECT totalamount FROM orders WHERE orderid = $1;
$$ LANGUAGE sql;
```

Note, en el ejemplo 16, que ambas funciones están compuestas por una sola sentencia SELECT, retornando un tipo de dato básico.

La primera función del ejemplo 17 está conformada por dos sentencias, siendo la última el SELECT necesario para el retorno de la función. En la segunda, que da respuesta a la misma necesidad, se garantiza el retorno de la función mediante la cláusula RETURNING en el UPDATE.

Ejemplo 17: Función que incrementa el precio en un 5% a un producto y lo muestra

```
CREATE FUNCTION incrementar_precio(id integer) RETURNS numeric AS
$$
   UPDATE products SET price = price + 0.05 * price WHERE prod_id = id;
   SELECT price FROM products WHERE prod_id = id;
$$ LANGUAGE sql;

-- Empleando la cláusula RETURNING en el UPDATE
CREATE OR REPLACE FUNCTION incrementar_precio(id integer) RETURNS numeric AS
$$
   UPDATE products SET price = price + 0.05 * price WHERE prod_id = id
    RETURNING price;
$$ LANGUAGE sql;
```

Empleo y retorno de tipos de datos compuestos

Cuando se escriben funciones que utilizan tipos de datos compuestos no basta con emplear el parámetro, sino que hay que especificar qué campo de dicho tipo de dato se utilizará.

Por ejemplo, si se deseara incrementar y devolver el precio de un producto determinado, en un 5%, pudiera implementarse una función como la mostrada en el ejemplo 18 que, a diferencia del ejemplo anterior, recibe como parámetro un producto en lugar de su identificador.

Ejemplo 18: Función que incrementa, en un 5%, y muestra el precio de un producto pasado como parámetro

```
CREATE FUNCTION aumentar_precio(prod products) RETURNS numeric AS
$$
   UPDATE products SET price = price + 0.05 * price WHERE prod_id =
    prod.prod_id RETURNING price;
$$ LANGUAGE sql;
```

Note que en el ejemplo anterior se hace referencia a *prod_id* mediante la calificación del atributo con el producto pasado por parámetro.

Si se quisiera utilizar la función *aumentar_precio* del ejemplo anterior para subirle el precio a un producto determinado, pudiera utilizarse la consulta mostrada en el ejemplo 19. Constate que dos de las ventajas de pasar por parámetro un dato compuesto son que no se necesita conocer previamente un atributo en particular y, se puede hacer lo que se desee a más de un registro, siempre que en el WHERE de la consulta que realiza la llamada a la función se especifiquen las condiciones necesarias para ello.

Ejemplo 19: Empleo de la función aumentar_precio para incrementar el precio del producto ACADEMY ADAPTATION

```
dell=# SELECT common_prod_id, aumentar_precio(products.*)
dell=# FROM products
dell=# WHERE title= 'ACADEMY ADAPTATION';
 common_prod_id | aumentar_precio
----------------+-----------------
           7173 |           30.44
(1 fila)
```

Note aquí que el empleo del asterisco (*) en el SELECT se utiliza para seleccionar la tupla actual de la tabla como un valor compuesto, aunque también puede ser referenciada usando solamente el nombre de la tabla, pero este uso es poco empleado por acarrear confusión.

Una función puede, además, retornar un tipo de dato compuesto. Por ejemplo, si se quisieran obtener todos los datos de un cliente pasado por parámetro, pudiera implementarse una función como la mostrada en el ejemplo 20.

Ejemplo 20: Función que devuelve todos los datos de un cliente pasado por parámetro

```
CREATE FUNCTION mostrar_cliente(id integer) RETURNS customers AS
$$
   SELECT * FROM customers WHERE customerid = id;
$$ LANGUAGE sql;
```

Más aun, si se quisiera, siguiendo con el ejemplo anterior, solamente devolver el nombre de un cliente pasado por parámetro, la llamada a la función quedaría de la forma que se observa en el ejemplo 21; lo que se realiza accediendo al atributo *firstname* de la tabla *customers* sobre la que se realiza la selección en la función *mostrar_cliente*.

Ejemplo 21: Empleo de la función mostrar_cliente para devolver el nombre del cliente con id 31

```
dell=# SELECT (mostrar_cliente(31)).firstname;
 firstname
----------
 XSKFVE
(1 fila)
```

Retorno de conjunto de valores

Muchas veces se necesita que una función retorne un conjunto de valores, por ejemplo, aquellos clientes menores de 30 años, o los que viven en determinada ciudad, etc. Con lo visto hasta ahora esto no puede hacerse, pero su implementación es posible mediante la especificación de SETOF en la cláusula RETURNS cuando se define la función.

Cuando una función SQL es declarada para que retorne SETOF *algún_tipo*, al invocarse lo que se hace es retornar cada fila de la consulta resultante como un elemento del conjunto de resultado. Esta funcionalidad puede ser usada:

- Al llamarse a la función en la cláusula FROM, siendo cada elemento del resultado una fila de la tabla resultante de la consulta.
- Al invocarse la función como parte de la lista del SELECT (que devuelve el resultado enumerando en cada tupla los valores de las columnas separados por coma).

La segunda opción tiene como problema que cuando se pone como parte de la lista del SELECT más de una función que retorna un conjunto de valores, el resultado puede no tener mucho sentido; este es uno de los motivos por los que en la Documentación Oficial se dice que esta funcionalidad pudiera desaparecer en versiones posteriores del

gestor, no obstante, sigue siendo actualmente una funcionalidad válida y ampliamente utilizada.

Por ejemplo, si se quisiera obtener un listado de los productos existentes ordenados por el identificador, pudiera implementarse una función como la mostrada en el ejemplo 22.

Ejemplo 22: Función que devuelve el listado de los productos existentes

```
CREATE FUNCTION listar_productos() RETURNS SETOF products AS
$$
   SELECT * FROM products ORDER BY prod_id;
$$ LANGUAGE sql;

-- Invocación de la función desde la cláusula FROM
dell=# SELECT * FROM listar_productos();
 prod_id | category |          title           | ... | common_prod_id
---------+----------+--------------------------+-----+----------------
       1 |       14 | ACADEMY ACADEMY          |     |           1976
       2 |        6 | ACADEMY ACE              |     |           6289
-- More --

-- Invocación de la función como un elemento del SELECT
dell=# SELECT listar_productos();
       listar_productos
--------------------------------
 (1,14,"ACADEMY ACADEMY",…,1976)
 (2,6,"ACADEMY ACE",…,6289)
-- More --
```

Pudiera, además, implementarse la función empleando parámetros de salida, por ejemplo, si se quisiera obtener el nombre y precio de los productos existentes, la función del ejemplo 23 serviría.

Ejemplo 23: Función que devuelve nombre y precio de los productos existentes empleando parámetros de salida

```
CREATE FUNCTION mostrar_productos(OUT nombre varchar, OUT precio numeric)
 RETURNS SETOF record AS
$$
   SELECT title, price FROM products;
$$ LANGUAGE sql;

--Invocación de la función desde la cláusula FROM
dell=# SELECT * FROM mostrar_productos();
     nombre      | precio
-----------------+--------
 ACADEMY ACADEMY | 25.99
 ACADEMY ACE     | 20.99
 -- More --
```

Note en el ejemplo anterior que el tipo definido en SETOF es RECORD, que indica que la función debe retornar un conjunto de filas con una estructura que se define al ser revisada la función por el analizador sintáctico durante su invocación. Este tipo se emplea cuando el resultado:

- No es la estructura completa de una tabla, por ejemplo, cuando se quiere obtener solamente un conjunto de los atributos existentes, como en el caso anterior.
- Se obtiene de concatenaciones entre tablas, por ejemplo, cuando se desea mostrar los productos existentes y el nombre de la categoría a que pertenecen.

Retorno de tipo tabla

Otra de las formas para retornar un conjunto de valores es empleando la cláusula RETURNS TABLE cuando se define la función. La misma posee la ventaja de que fue añadida recientemente al estándar y, por ende, puede ser más portable que el uso de SETOF.

Por ejemplo, la función implementada en el ejemplo 23 con parámetros de salida pudiera reescribirse de la forma mostrada en el ejemplo 24 empleándose RETURNS TABLE.

Ejemplo 24: Función que devuelve nombre y precio de los productos existentes empleando RETURNS TABLE

```
CREATE OR REPLACE FUNCTION mostrar_productos() RETURNS TABLE(nombre varchar,
 precio numeric) AS
$$
   SELECT title, price FROM products;
$$ LANGUAGE sql;
```

Note que para emplear esta forma debe especificarse cada columna de la salida que se desee como una columna de la tabla del resultado en la cláusula RETURNS TABLE y que, entonces, el tipo de dato de la función se sustituye por una tabla con las columnas y tipos de datos especificados.

2.5 Resumen

Las funciones SQL son la forma más sencilla de agregar código a PostgreSQL para su extensión. Una función SQL es un conjunto de sentencias SQL empaquetadas, con el objetivo de realizar un grupo de acciones para obtener un resultado.

Para definir una función SQL se emplea el comando CREATE FUNCTION, en el que se especifica el nombre de la función, los parámetros que recibirá, su tipo de retorno y el cuerpo de la misma, en el que se listan las sentencias SQL que la conformarán.

Los parámetros de las funciones, que pueden ser de entrada, salida, entrada/salida o variables, especifican aquellos valores que el usuario define sean empleados para su procesamiento posterior en el cuerpo de la función y; pueden ser accesibles mediante su nombre o de la forma $*número*, siendo *número* la posición que ocupa en el listado de parámetros comenzando por el 1.

Las funciones SQL pueden retornar uno de los tipos básicos definidos en el estándar SQL o por el usuario, el valor nulo o un conjunto de valores (utilizando SETOF o RETURNS TABLE).

2.6 Para hacer con SQL

1. Implementar funciones que permitan la inserción de datos en cada una de las tablas siguientes y retorne la tupla insertada:
   a. *categories*
   b. *products*
   c. *customers*
   d. *orderlines*
   e. *orders*

2. Desarrollar una función que dado el identificador de un producto devuelva toda la información referente a él.

3. Crear una función que dado el identificador de un cliente devuelva su nombre y apellidos, edad, género, correo electrónico, teléfono, ciudad y país.
   a. Emplear parámetros de salida.
   b. ¿Qué tipo de dato pudiera emplearse para no tener que declarar tantos parámetros de salida?

4. Obtener una función que dado el identificador de un producto actualice su precio, pasado también por parámetro, y muestre finalmente toda la información del producto.

5. Elaborar una función que dado el identificador del producto devuelva toda la información referente a él, así como el total de pedidos realizados de él.

6. Implementar una función que devuelva todos los clientes de sexo femenino menores de 35 años.
   a. ¿De qué formas puede definirse el tipo de retorno de esta función? ¿Cuál es la diferencia entre ellas?

7. Desarrollar una función que dado el identificador de un cliente elimine las órdenes realizadas por él anteriores al 24 de abril de 2004 y muestre luego, de las resultantes, su identificador, fecha, monto neto y monto total.

8. Crear una función que inserte un nuevo producto suministrado por el usuario y, además, muestre nombre, precio y categoría, de los existentes en la base de datos.

9. Obtener una función que muestre, por categoría, el total de productos existentes.

Capítulo 3
Programación de funciones en PL/pgSQL

Metas:

- Dominar la estructura y sintaxis para la definición de funciones PL/pgSQL
- Utilizar parámetros y variables en funciones PL/pgSQL
- Emplear estructuras iterativas y condicionales soportadas por PL/pgSQL
- Utilizar los comandos implementados por PL/pgSQL para el retorno de valores
- Conocer las opciones para el paso de mensajes y el tratamiento de errores
- Dominar las opciones implementadas por PL/pgSQL para la definición de disparadores y cursores

PROGRAMACIÓN DE FUNCIONES EN PL/PGSQL

3.1 Introducción a las funciones PL/pgSQL

En el capítulo se realiza una introducción a cómo programar funciones en PL/pgSQL, lenguaje procedural para PostgreSQL empleado para implementar funciones, disparadores y cursores, que ha sido incluido por defecto en todas las liberaciones del gestor a partir de su versión 9.0.

PL/pgSQL es un lenguaje influenciado directamente del PL/SQL de Oracle, que al igual que las funciones SQL, permite la agrupación de consultas SQL evitando la saturación de la red entre el cliente y el servidor de bases de datos. Brinda, además, un grupo de ventajas adicionales entre las que destacan que puede 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 y cursores. Elementos que le otorgan mayor potencialidad al combinar las ventajas de un lenguaje procedural y la facilidad de SQL.

3.2 Estructura de las funciones PL/pgSQL

Al ser PL/pgSQL un lenguaje estructurado por bloques, la definición de una función en él debe ser un bloque de la forma:

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

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.

En este lenguaje procedural:

- Las sentencias (y declaraciones) terminan con punto y coma (;) al final de cada línea, al igual que en las funciones SQL.
- Para retornar el resultado se emplea la cláusula RETURN (para más detalles ver el epígrafe "3.6 Retorno de valores" en la página 46).

- 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 puede ser un sub-bloque, empleado para realizar agrupaciones lógicas o crear variables para un grupo de sentencias; pudiendo ser accedidas las variables de bloques externos en un sub-bloque al calificarlas con la etiqueta del bloque al que pertenecen.
- Cuando la función es creada se chequea su estructura no las sentencias SQL, por lo que puede crearse sin errores y al invocarse generarse errores en tiempo de ejecución derivados de los comandos SQL.

La estructura en forma de bloque puede constatarse en la función en PL/pgSQL mostrada en el ejemplo 25.

Ejemplo 25: Función que retorna la suma 2 de números enteros

```
CREATE FUNCTION sumar(num1 integer, num2 integer) RETURNS integer AS
$$
BEGIN
   RETURN $1 + $2;
END;
$$ LANGUAGE plpgsql;

-- Invocación de la función
dell=# SELECT sumar(2, 3);
sumar
-----
    5
(1 fila)
```

El ejemplo 26 muestra el tratamiento de variables en bloques anidados y el acceso a variables externas mediante su calificación.

Ejemplo 26: Empleo de variables en bloques anidados

```
CREATE FUNCTION incrementar_precio_porciento(id integer) RETURNS numeric AS
$$
<<principal>>
DECLARE
  incremento numeric := (SELECT price FROM products WHERE prod_id = $1) * 0.3;
BEGIN
```

```
   RAISE NOTICE 'El precio con el incremento será de % pesos', incremento;
    -- Incremento en un 30%
   DECLARE
      incremento numeric := (SELECT price FROM products WHERE prod_id = $1) *
       0.5;
   BEGIN
      RAISE NOTICE 'El precio con el incremento excepcional será de % pesos',
       incremento; -- Incremento en un 50%
      RAISE NOTICE 'El precio con el incremento será de % pesos',
       principal.incremento; -- Incremento en un 30%
   END;
   RAISE NOTICE 'El precio con el incremento será de % pesos', incremento;
    -- Incremento en un 30%
   RETURN incremento;
END;
$$ LANGUAGE plpgsql;

--Invocación de la función
dell=# SELECT * FROM incrementar_precio_porciento(1);
NOTICE: El precio con el incremento será de 7.797 pesos
NOTICE: El precio con el incremento excepcional será de 12.995 pesos
NOTICE: El precio con el incremento será de 7.797 pesos
NOTICE: El precio con el incremento será de 7.797 pesos
 incrementar_precio_porciento
------------------------------
                       7.797
(1 fila)
```

3.3 Variables en PL/pgSQL

Las variables pueden ser de cualquier tipo de dato SQL o definido por el usuario y deben ser declaradas en la sección DECLARE.

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

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

Donde:

- La cláusula CONSTANT especifica que la variable es constante, por lo que el valor inicial asignado es el valor con que se mantendrá durante la ejecución de la función; de no ser especificado, la variable es inicializada con el valor nulo.
- La cláusula NOT NULL especifica que la variable no puede tener asignado un valor

nulo (generándose un error en tiempo de ejecución en caso de que ocurra); todas las variables declaradas como no nulas deben tener especificado un valor no nulo.

- La cláusula DEFAULT (o :=) asigna un valor a la variable.

Ejemplos de declaraciones de variables son los siguientes:

```
impuesto CONSTANT numeric := 0.3;     -- variable numérica con valor constante
                                      -- 0.3
incremento integer DEFAULT 31;        -- variable entera con valor 31
region varchar := 'Occidente';        -- variable de texto con el valor
                                      -- Occidente
fecha date NOT NULL := '2007-11-24'; -- variable fecha, no nula y con valor
                                      -- 2007-11-24
```

Copiado de tipos de datos

Para el trabajo con variables una de las utilidades de PL/pgSQL es el copiado de tipos de datos, útil para que: (1) no sea necesario conocer el tipo de dato de la estructura que se esté referenciando y, (2) en caso de cambiar dicho tipo de dato no sea necesario cambiar la definición de la función.

Nota Los tipos de datos de PostgreSQL pueden verse en la Documentación Oficial, en la sección *Data Types*

http://www.postgresql.org/docs/9.5/static/datatype.html

Esta funcionalidad se utiliza de la forma:

```
variable%TYPE
```

Permitiendo %TYPE capturar el tipo de dato de la variable o columna de la tabla asociada, como se muestra en el ejemplo siguiente, en el que la variable declarada impuesto tomará el tipo de dato del campo *tax* de la tabla *orders*:

```
impuesto orders.tax%TYPE
```

Tipo de dato fila

PL/pgSQL soporta, además, el tipo de dato fila (ROWTYPE), un tipo compuesto que puede almacenar toda la fila de una consulta SELECT o de un FOR y del que sus campos

pueden ser accesibles calificándolos, como cualquier otro tipo de dato compuesto (ver el epígrafe en el capítulo 2 "Empleo y retorno de tipos de datos compuestos" en la página 23).

Para emplearlo se debe tener en cuenta que:

- En esta estructura solo 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 la misma estructura de las filas de una tabla o vista mediante la forma:

```
nombre_tabla%ROWTYPE
```

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

```
variable nombre_tabla
```

Nota En PostgreSQL cada tabla tiene asociado un tipo de dato compuesto con su mismo nombre

Ejemplos equivalentes empleando ambas formas son los siguientes:

```
cliente customers%ROWTYPE;
cliente customers;
```

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 permite definir tipos de datos propios para 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 en 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 del mismo es de la forma:

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

Ejemplo de su empleo es el siguiente:

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

Tipo de dato RECORD

PL/pgSQL soporta el tipo de dato RECORD, similar a ROWTYPE pero sin estructura predefinida, la que toma de la fila actual asignada durante la ejecución de los comandos SELECT o FOR.

RECORD no es un tipo de dato verdadero sino un contenedor. Este no es el mismo concepto que cuando se declara una función para que retorne 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 actual es determinada cuando la llamada es revisada por el analizador sintáctico (ver el "Ejemplo 37" en la página 49 para su uso), 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 el 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.

Parámetros de funciones PL/pgSQL

En PL/pgSQL, al igual que en SQL (ver el epígrafe "2.3 Parámetros de funciones SQL" en la página 18), 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:

- Al nombrar el parámetro en el comando CREATE FUNCTION (forma preferida).
- En la sección de declaraciones (única forma antes de la versión 8.0 de PostgreSQL).

El ejemplo 27 muestra estas 2 formas de hacer referencia a los parámetros de las funciones en PL/pgSQL.

Ejemplo 27: 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;

-- Función que define 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 se emplea no solo para definir el alias de un parámetro, sino que puede ser utilizado para definir el alias de cualquier variable.

3.4 Sentencias en PL/pgSQL

Una de las partes más significativas de una función en PL/pgSQL es la especificación de las sentencias, que permitirán detallar las acciones necesarias para obtener el resultado deseado con la ejecución de la función. En este epígrafe se mostrarán algunas de las sentencias más comunes utilizadas en PL/pgSQL.

Asignación de valores a variables

La asignación de valores a variables en PL/pgSQL se realiza en la sección DECLARE de la forma:

```
variable := expresión;
```

Donde:

- *variable* (opcionalmente calificada con el nombre de un bloque): puede ser una variable simple, un campo de una fila, RECORD o algún elemento de un arreglo.
- *expresión*: debe devolver un único valor.

Ejemplos de asignaciones son los siguientes:

```
pais := 'Cuba';
impuesto := price * 2.4;
nombre := (SELECT firstname FROM customers WHERE customerid = $1);
```

Ejecución de consultas que arrojan resultados con una única fila

Para almacenar el resultado de un comando SQL que retorne una fila puede utilizarse una variable de tipo RECORD, ROWTYPE o una lista de variables escalares, mediante la adición de la cláusula INTO al comando (SELECT, INSERT, UPDATE o DELETE con cláusula RETURNING y comandos de utilidad que retornan 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);
```

Donde:

- De emplearse más de una variable deben estar separadas por coma.
- Si una fila o lista de variables es usada, las columnas resultantes de la consulta deben coincidir exactamente con la misma estructura de las variables y sus tipos de datos o se generará un error en tiempo de ejecución.
- Si la opción STRICT no es especificada, la variable almacenará la primera fila retornada por la consulta (no bien definida a menos que se emplee ORDER BY) o nulo si no arroja resultados, siendo descartadas el resto de las filas.
- Si la opción STRICT es especificada y la consulta devuelve más de una fila se genera un error en tiempo de ejecución.
- En consultas INSERT, UPDATE o DELETE, aun cuando no sea especificado el STRICT se genera el error en tiempo de ejecución si el resultado tiene más de una fila ya que, al estas no tener la opción ORDER BY, no se puede determinar cuál de las filas del resultado se debería devolver.

Las consultas siguientes emplean estos comandos para capturar una fila del resultado.

Ejemplo 28: Captura de un campo del resultado de consultas SELECT y UPDATE

```
-- Guardar el resultado del SELECT en la variable fecha
SELECT orderdate INTO fecha FROM orders WHERE orderid = 1;

-- Guardar el resultado de firstname 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 = 'Clandestinos' WHERE prod_id = 1 RETURNING *
 INTO pd;
```

Ejecución de comandos dinámicos

Existen escenarios donde se hace inevitable utilizar 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 se utiliza 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 el resultado de la consulta, puede(n) ser de tipo RECORD, fila o lista de variables escalares separadas por coma, con las especificaciones explicadas previamente para el empleo de la cláusula INTO (ver el epígrafe en el presente capítulo "Ejecución de consultas que arrojan resultados con una única fila" en la página 38) que al ser detalladas descarta las filas resultantes.
- *USING expresión*: suministra valores a ser insertados en el comando.

La sentencia ejecutada con este comando es planeada cada vez que es llamado el EXECUTE, por lo que la cadena que la contiene puede ser creada dinámicamente dentro de la función.

Este comando es especialmente útil cuando se necesita usar valores de parámetros en la cadena a ser ejecutada que involucren tablas o tipos de datos dinámicos. Para su empleo se deben tener 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 constante (como en la primera sentencia del ejemplo 29) es equivalente a escribir la consulta directamente en PL/pgSQL, la diferencia radica en que EXECUTE replanifica el comando para cada ejecución, generando un plan específico para los valores de los parámetros empleados, mientras que PL/pgSQL crea un plan genérico y lo reutiliza, por lo que en situaciones donde el mejor plan dependa de los valores de los parámetros se recomienda el empleo de EXECUTE.
- La ejecución de consultas dinámicas requiere un manejo cuidadoso ya que pueden contener caracteres de acotado que, de no tratarse adecuadamente, pueden generar errores en tiempo de ejecución. Para ello se pueden emplear 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 *quote_literal* pero es empleada cuando puede haber parámetros nulos, retornando una cadena nula y no derivando en un error al convertir todo el comando dinámico en nulo.

• Las consultas dinámicas pueden ser escritas, además, de forma segura mediante el empleo de la función *format*, que resulta ser una manera eficiente al no ser convertidos a texto los parámetros.

Las sentencias del ejemplo 29 demuestran los elementos previamente analizados, consultas que reciben por parámetro el nombre de la tabla a eliminar (segunda consulta) y, la columna a actualizar, el nuevo valor y su identificador (tercera consulta).

Ejemplo 29: Empleo del comando EXECUTE en consultas constantes y dinámicas

```
-- Empleo del comando EXECUTE en consultas constantes
EXECUTE 'SELECT * FROM customers WHERE customerid = $1';

-- Consulta dinámica que recibe el nombre de la tabla a eliminar
EXECUTE 'DROP TABLE IF EXISTS ' || $1 || ' CASCADE';

-- Consulta dinámica que actualiza un campo de la tabla products
EXECUTE 'UPDATE products SET ' || quote_ident($1) || ' = ' ||
 quote_nullable($2) || ' WHERE prod_id = ' || quote_literal($3);
```

Note que para la ejecución de consultas dinámicas debe dejarse un espacio en blanco entre las cadenas a unir de la consulta preparada, de no hacerse se genera un error ya que al convertir toda la cadena se hace sin espacios entre los elementos concatenados.

3.5 Estructuras de control

PL/pgSQL incorpora estructuras de control (condicionales e iterativas) para imprimirle mayor flexibilidad y poder al lenguaje mediante variadas opciones para la manipulación de los datos.

Estructuras condicionales

PL/pgSQL implementa 5 formas de los condicionales IF y CASE (ver la tabla 1) que permiten la ejecución de sentencias basadas en condiciones, como muestran los ejemplos del 30 al 34.

Tabla 1: Estructuras condicionales implementadas por PL/pgSQL

Tipo	Sintaxis	Uso
IF-THEN	**IF** *condición* **THEN** *sentencias...;* **END IF;**	Forma más simple del IF, las sentencias son ejecutadas si la condición es verdadera.
IF-THEN-ELSE	**IF** *condición* **THEN** *sentencias...;* [**ELSE** *sentencias...;*] **END IF;**	Añade al tipo anterior la cláusula ELSE para especificar sentencias alternativas a ejecutar cuando no sea evaluada de verdadera la condición definida.
IF-THEN-ELSIF	**IF** *condición* **THEN** *sentencias...;* [**ELSIF** *condición* **THEN** *sentencias...;* …] [**ELSE** *sentencias...;*] **END IF;**	Empleada cuando hay más de 2 alternativas a valorar, evalúa cada condición IF hasta que encuentre una verdadera, en ese caso ejecuta las sentencias asociadas y no evalúa ningún IF restante; en caso de que no sea evaluada de verdadera ninguna condición y exista la cláusula ELSE, las sentencias asociadas a esta son ejecutadas.
CASE	**CASE** *expresión_búsqueda* **WHEN** *expresión1* [, *expresión2* [...]] **THEN** *sentencias...;* [**WHEN** *expresión1 [, expresión2* [...]] **THEN** *sentencias...;* ...] [**ELSE** *sentencias...;*] **END CASE;**	Forma más simple del CASE que permite la ejecución de sentencias basadas en condicionales de operadores de igualdad. La expresión de búsqueda es evaluada una vez y posteriormente comparada con cada expresión en la cláusula WHEN, de coincidir son ejecutadas las sentencias asociadas a dicha cláusula ignorando el resto de las cláusulas WHEN o ELSE existentes; de no coincidir es ejecutada la cláusula ELSE, si existe.
CASE buscado	**CASE** **WHEN** *condición1* **THEN** *sentencias...;* [**WHEN** *condición2* **THEN** *sentencias...;* ...] [**ELSE** *sentencias...;*] **END CASE;**	Permite la ejecución de sentencias basadas en condicionales. Cada expresión o condición de la cláusula WHEN es evaluada en cada turno hasta que una es verdadera, ejecutándose las sentencias asociadas e ignorándose el resto de las cláusulas WHEN o ELSE existentes en la estructura; al igual que en el CASE, de no haber una expresión evaluada de verdadera se ejecuta la cláusula ELSE, de existir.

En el ejemplo 30 se chequea que se inserte un nuevo producto si tiene un precio superior a los 25 pesos.

Ejemplo 30: Empleo de la estructura condicional IF-THEN implementada en PL/pgSQL

```
IF price > 25.0 THEN
   INSERT INTO products VALUES ($1, $2, $3, $4, $5, $6, $7);
END IF;
```

En el ejemplo 31 se inserta un nuevo producto si tiene un precio superior a los 25 pesos, si es menor se muestra una notificación especificando que el producto es muy barato.

Ejemplo 31: Empleo de la estructura condicional IF-THEN-ELSE implementada en PL/pgSQL

```
IF price > 25.0 THEN
   INSERT INTO products VALUES ($1, $2, $3, $4, $5, $6, $7);
ELSE
   RAISE NOTICE 'El producto es muy barato';
END IF;
```

En el ejemplo 32 se inserta un nuevo producto si tiene un precio entre los 25 y los 50 pesos, si es menor se muestra una notificación indicando que el producto es muy barato y en caso contrario, se muestra una notificación indicando que el producto es muy caro.

Ejemplo 32: Empleo de la estructura condicional IF-THEN-ELSIF implementada en PL/pgSQL

```
IF price BETWEEN 25.0 AND 50.0 THEN
   INSERT INTO products VALUES ($1, $2, $3, $4, $5, $6, $7);
ELSIF price < 25.00 THEN
   RAISE NOTICE 'El producto es muy barato';
ELSE
   RAISE NOTICE 'El producto es muy caro';
END IF;
```

En el ejemplo 33 se evalúa si el producto es de categoría infantil (2, 3) o no y, en el ejemplo 34 se evalúa si el producto es de categoría infantil (2, 3), drama (5, 7, 8) u otra.

Ejemplo 33: Empleo de la estructura condicional CASE implementada en PL/pgSQL

```
CASE category
   WHEN 2, 3 THEN
      RAISE NOTICE 'El producto es de categoría Infantil';
   ELSE
      RAISE NOTICE 'El producto no es de categoría Infantil';
END CASE;
```

Ejemplo 34: Empleo de la estructura condicional CASE buscado implementada en PL/pgSQL

```
CASE
   WHEN (category = 2 OR category = 3) THEN
      RAISE NOTICE 'El producto es de categoría Infantil';
   WHEN (category = 5 OR category = 7 OR category = 8) THEN
      RAISE NOTICE 'El producto es de categoría Drama';
   ELSE
      RAISE NOTICE 'El producto tiene otra categoría';
END CASE;
```

Como evidencian los ejemplos anteriores, pueden utilizarse operadores lógicos como AND y OR en las sentencias condicionales. De modo general, estas sentencias son útiles, sobre todo para el control de flujo, donde puede ejecutarse una acción u otra en función de las condiciones que se cumplan.

El uso de IF o CASE depende de las preferencias del programador, al lograse con ambas el mismo resultado. En ocasiones una otorga más legibilidad al código, lo que puede facilitar el mantenimiento del mismo; por ejemplo, cuando son muchas condiciones a chequear en los valores de las variables el CASE puede ser más legible, pero si son pocas las condiciones a chequear el IF suele ser la opción más empleada.

Estructuras iterativas

PL/pgSQL implementa varias formas iterativas, la tabla siguiente las muestra.

Tabla 2: Estructuras iterativas implementadas por PL/pgSQL

Tipo	Sintaxis	Uso
LOOP simple	[<<*etiqueta*>>] **LOOP** *sentencias...;* **END LOOP** [*etiqueta*];	Define un ciclo incondicional que es repetido indefinidamente hasta que encuentra una sentencia EXIT o RETURN. La etiqueta puede ser usada para sentencias EXIT o CONTINUE en LOOP anidados.
EXIT	**EXIT** [*etiqueta*] [**WHEN** *condición*];	Termina la ejecución de un LOOP o bloque; si se especifica la etiqueta termina el elemento etiquetado, pasando a la siguiente sentencia después del END asociado a la estructura terminada; de no hacerse termina el LOOP más cercano. Si se especifica el WHEN el EXIT ocurre al cumplirse la condición.

Tipo	Sintaxis	Uso
CONTINUE	**CONTINUE** [*etiqueta*] [**WHEN** *condición*];	Continúa la ejecución del LOOP especificado (mediante la etiqueta o el propio LOOP en ejecución); de especificarse la cláusula WHEN la próxima iteración del LOOP inicia solo si la condición es verdadera, en caso contrario el control pasa a la sentencia siguiente después del CONTINUE.
WHILE	[<<*etiqueta*>>] **WHILE** *condición* **LOOP** *sentencias...;* **END LOOP** [*etiqueta*];	Repite una secuencia de sentencias mientras la condición evaluada sea verdadera, la condición es chequeada antes de cada entrada en el ciclo LOOP.
FOR	[<<*etiqueta*>>] **FOR** *nombre* **IN** [**REVERSE**] *expresión1* .. *expresión2* [**BY** *expresión*] **LOOP** *sentencias...;* **END LOOP** [*etiqueta*];	Crea un LOOP que itera sobre un rango de valores enteros. La variable nombre es automáticamente definida de tipo entero y existe solamente dentro del ciclo; las 2 expresiones delimitan los límites inferior y superior del rango; la cláusula BY especifica el incremento en cada iteración (por defecto 1); la cláusula REVERSE especifica que en lugar de incrementarse el valor a iterar se decrementa; las expresiones son evaluadas en cada entrada al ciclo LOOP.
FOR recorriendo el resultado de una consulta	[<<*etiqueta*>>] **FOR** *variable* **IN** *consulta* **LOOP** *sentencias...;* **END LOOP** [*etiqueta*];	Permite iterar por el resultado de una consulta y manipular los datos de cada tupla. La variable es de tipo RECORD, fila o un listado de variables escalares separadas por coma.

En el ejemplo 35 se implementan 2 funciones, la primera, haciendo uso del LOOP cambia el precio de un producto pasado por parámetro (duplicándolo el total de veces definido como segundo parámetro) y retorna el precio final; la segunda realiza lo mismo, pero para el retorno del precio final hace uso del WHILE.

Ejemplo 35: Empleo de las estructuras iterativas LOOP y WHILE implementadas por PL/pgSQL

```
--Función que usa la estructura iterativa LOOP
CREATE FUNCTION cambiar_precio(id integer, cant integer) RETURNS numeric AS
$$
DECLARE
   contador integer := 0;
   precio numeric;
BEGIN
   LOOP
      UPDATE products SET price = price * 2 WHERE prod_id = id RETURNING price
       INTO precio;
      contador := contador + 1;
      IF contador = cant THEN
         EXIT;
      END IF;
   END LOOP;
   RETURN precio;
END;
$$ LANGUAGE plpgsql;
dell=# SELECT cambiar_precio(131, 2);
 cambiar_precio
----------------
          39.96
(1 fila)

--Función que usa la estructura iterativa WHILE
CREATE OR REPLACE FUNCTION cambiar_precio(id integer, cant integer) RETURNS
 numeric AS
$$
DECLARE
   contador integer := 0;
   precio numeric;
BEGIN
   WHILE contador < cant LOOP
      UPDATE products SET price = price * 2 WHERE prod_id = id RETURNING price
       INTO precio;
      contador := contador + 1;
   END LOOP;
   RETURN precio;
END;
$$ LANGUAGE plpgsql;
```

Como se muestra en la tabla y ejemplo anteriores, existen varias estructuras iterativas para el control de los ciclos. La elección de un caso u otro depende del escenario y de las preferencias del programador; en ocasiones una otorga mayor legibilidad y elegancia al código, lo cual contribuye al entendimiento y mantenimiento del mismo.

El uso del FOR puede analizarse en el "Ejemplo 37" en la página 49.

3.6 Retorno de valores

PL/pgSQL implementa 3 comandos que permiten devolver datos de una función, ellos son RETURN, RETURN NEXT y RETURN QUERY.

La cláusula RETURN, empleada cuando la función no devuelve un conjunto de datos, tiene la forma:

```
RETURN expresión;
```

Para su empleo se debe tener en cuenta que esta cláusula:

- Devuelve el valor de evaluar la expresión terminando la ejecución de la función.
- De haberse definido una función con parámetros de salida no se hace necesario especificar ninguna expresión en ella.
- En funciones que retornen el tipo de dato VOID puede emplearse sin especificar ninguna expresión para terminar la función tempranamente.
- No debe dejar de especificarse 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 ejemplo 36 muestra varios escenarios en los que se emplea el RETURN para devolver datos escalares, compuestos, elementos de un dato compuesto y parámetros de salida.

Ejemplo 36: Empleo del RETURN para devolver valores y culminar la ejecución de la función

```
-- Empleo de RETURN para devolver un dato escalar
CREATE FUNCTION devolver_producto(id integer) RETURNS varchar AS
$$
DECLARE
   prod varchar;
BEGIN
   SELECT title INTO prod FROM products WHERE prod_id = $1;
   RETURN prod;
END;
$$ LANGUAGE plpgsql;
```

```
-- Empleo de parámetros de salida
CREATE OR REPLACE FUNCTION devolver_producto(id integer, OUT titulo varchar)
 RETURNS varchar AS
$$
BEGIN
   SELECT title INTO $2 FROM products WHERE prod_id = $1;
   RETURN;
END;
$$ LANGUAGE plpgsql;

-- Empleo de RETURN para devolver un dato compuesto
CREATE FUNCTION datos_producto(id integer) RETURNS products AS
$$
DECLARE
   prod products;
BEGIN
   SELECT * INTO prod FROM products WHERE prod_id = $1;
   RETURN prod;
END;
$$ LANGUAGE plpgsql;

-- Empleo de RETURN para devolver un elemento de un dato compuesto
CREATE FUNCTION datos_producto_titulo(id integer) RETURNS varchar AS
$$
DECLARE
   prod record;
BEGIN
   SELECT * INTO prod FROM products WHERE prod_id = $1;
   RETURN prod.title;
END;
$$ LANGUAGE plpgsql;

-- Dato compuesto por el nombre y precio del producto
CREATE TYPE mini_prod AS (
   nombre varchar, precio numeric);

-- Empleo de RETURN para devolver un dato compuesto
CREATE FUNCTION datos_producto_mini_prod(id integer) RETURNS mini_prod AS
$$
DECLARE
   prod mini_prod;
BEGIN
   SELECT title, price INTO prod FROM products WHERE prod_id = $1;
   RETURN prod;
END;
$$ LANGUAGE plpgsql;
```

Note en el ejemplo anterior que:

- En la función donde se emplea un parámetro de salida la cláusula RETURN no necesita tener una expresión asociada.
- En la función donde se emplea RETURN para el retorno de un dato compuesto, en este caso todos los datos de un producto dado su identificador, se debe garantizar que el resultado devuelva una sola tupla, en caso contrario la función devuelve la primera del resultado.
- En la función donde se emplea RETURN para el retorno de un elemento de un dato compuesto, en este caso el nombre de un producto del que se pasa por parámetro su identificador, se debe calificar el producto mediante la forma *elemento.dato_compuesto*.
- En la función donde el tipo de dato compuesto tiene solamente el nombre y precio, se emplea un tipo de dato creado por el usuario, en este caso *mini_prod*.

Las cláusulas RETURN NEXT y RETURN QUERY, empleadas cuando la función devuelve un conjunto de datos, tienen la forma:

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

Para su empleo se debe tener en cuenta que estas cláusulas (ver su uso en el ejemplo 37):

- Son empleadas cuando la función se declara para que devuelva SETOF *algún_tipo*.
- RETURN NEXT se emplea con tipos de datos compuestos, RECORD o fila.
- RETURN QUERY añade el resultado de ejecutar una consulta al conjunto resultante de la función.
- Ambas pueden ser usadas en una misma función, concatenándose sus resultados.
- No culminan la ejecución de la función, simplemente añaden cero o más filas al resultado, pudiendo emplearse un RETURN sin argumento para salir de la función.
- De declararse parámetros de salida se puede especificar el RETURN NEXT sin una expresión, en cuyo caso la función debe declararse para que retorne SETOF RECORD.

En el ejemplo 37 se implementan funciones que devuelven los productos registrados en que su identificador sea mayor que el pasado por parámetro.

Note que durante la invocación de la función en que se emplea RETURN QUERY y RECORD debe especificarse qué estructura tendrá el resultado, para ello se

hace uso de la cláusula AS especificando nombre y tipo de dato de cada elemento de la salida; de lo contrario se arrojará un error en tiempo de ejecución.

Ejemplo 37: Empleo de RETURN NEXT y RETURN QUERY para devolver conjuntos de valores

```
-- Empleando RETURN NEXT
CREATE FUNCTION datos_producto_setof(id 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;

-- Empleando RETURN QUERY
CREATE OR REPLACE FUNCTION datos_producto_setof(id integer) RETURNS SETOF
 products AS
$$
BEGIN
   RETURN QUERY SELECT * FROM products WHERE prod_id > $1;
   RETURN; -- Opcional
END;
$$ LANGUAGE plpgsql;

-- Empleando RETURN QUERY y devolviendo un RECORD
CREATE FUNCTION datos_producto_record(id integer) RETURNS SETOF record AS
$$
BEGIN
   RETURN QUERY SELECT * FROM products WHERE prod_id > $1;
   RETURN; -- Opcional
END;
$$ LANGUAGE plpgsql;

-- Invocación de la función datos_producto_record
dell=# SELECT * FROM datos_producto_record(1000) AS (prod_id integer, category
       integer, title varchar, actor varchar, price numeric, special smallint,
       common_prod_id integer);
prod_id|category|   title   |    actor    | price |special|common_prod_id
-------+--------+-----------+-------------+-------+-------+---------------
   1001|       9|ACE ACADEMY| MIRA HUDSON | 28.99 |     0 |          1926
   1002|       1|ACE ACE    | DORIS BENING| 10.99 |     0 |          4673
-- More --
```

3.7 Mensajes

En ejemplos analizados previamente se ha hecho uso de una de las funcionalidades de los lenguajes procedurales que le otorgan flexibilidad a la hora de interactuar con los usuarios, el paso de mensajes. En PL/pgSQL los mensajes se muestran utilizando la cláusula RAISE, de la cual se ha empleado en los ejemplos anteriores la opción NOTICE; además de esta opción la cláusula RAISE soporta las opciones DEBUG, LOG, INFO, WARNING y EXCEPTION, esta última utilizada por defecto:

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

Nota

Para mayor detalle sobre las opciones de RAISE puede analizar en la Documentación Oficial la sección *Errors and Messages* del capítulo *PL/pgSQL - SQL Procedural Language*

http://www.postgresql.org/docs/9.5/static/plpgsql-errors-and-messages.html

El ejemplo 38 muestra casos donde es utilizado RAISE con varias de las opciones de mensajes, se implementa una función que devuelve, dado el identificador de un producto, su título. En la primera se emplean FOUND y EXCEPTION para determinar si lo encontró o no. En la segunda se utiliza LOG (para en caso de no encontrarlo guardarlo en los registros) y WARNING (para advertir que el producto no está registrado).

Ejemplo 38: Mensajes utilizando las opciones EXCEPTION, LOG y WARNING de RAISE

```
CREATE FUNCTION producto(titulo integer) RETURNS varchar AS
$$
DECLARE
   prod varchar;
BEGIN
   SELECT title INTO prod FROM products WHERE prod_id = $1;
   IF not found THEN
      RAISE EXCEPTION 'Producto % no encontrado', $1;
   END IF;
   RETURN prod;
END;
$$ LANGUAGE plpgsql;
```

```
-- Invocación de la función
dell=# SELECT producto(1000000);
ERROR: Producto 1000000 no encontrado

-- Empleo de FOUND, LOG y WARNING
CREATE OR REPLACE FUNCTION producto(titulo integer) RETURNS varchar AS
$$
DECLARE
   prod varchar;
BEGIN
   SELECT title INTO prod FROM products WHERE prod_id = $1;
   IF not found THEN
   RAISE LOG 'Producto % no encontrado', $1;
      RAISE WARNING 'Producto % no encontrado, es posible que no se haya
       insertado aun', $1;
   END IF;
   RETURN prod;
END;
$$ LANGUAGE plpgsql;

-- Invocación de la función
dell=# SELECT producto(25000);
WARNING: Producto 25000 no encontrado, es posible que no se haya insertado aun
 producto
----------
(1 fila)
```

Nótese que se ha utilizado la variable especial FOUND, que es de tipo lógico (verdadero o falso), que por defecto en PL/pgSQL es FALSE y se activa luego de:

- Una asignación de un SELECT INTO que haya arrojado un resultado efectivo almacenado en la variable.
- Las operaciones UPDATE, INSERT o DELETE que hayan realizado alguna acción efectiva sobre la base de datos.
- Las operaciones RETURN QUERY y RETURN QUERY EXECUTE que hayan retornado algún valor.
- Las operaciones FETCH y MOVE que hayan realizado un retorno o reposición del cursor efectiva, respectivamente (ver el epígrafe "3.8 Cursores" en la página 54).

Nota

Existen otros casos donde se activa la variable FOUND, ver la sección *Obtaining the Result Status* del epígrafe *Basic Statements*, en la Documentación Oficial

https://www.postgresql.org/docs/9.5/static/plpgsql-statements.html#PLPGSQL-STATEMENTS-DIAGNOSTICS

Tratamiento de errores

Otra de las opciones que ofrece PostgreSQL para el paso de mensajes a los usuarios finales es la captura y tratamiento de errores, que se puede realizar empleando un bloque BEGIN con la cláusula EXCEPTION de la forma:

```
BEGIN
   sentencias…
EXCEPTION
   WHEN código_error THEN
      sentencias_tratamiento…
   [ WHEN código_error THEN
      sentencias_tratamiento… ]
END;
```

Donde:

- De ocurrir un error durante la ejecución de las sentencias, el control pasa a la lista EXCEPTION recorriendo todos los WHEN buscando uno coincidente.
- En caso de encontrar una coincidencia se ejecutan las *sentencias_tratamiento* asociadas.
- De no encontrar coincidencias se muestra el mensaje definido por el sistema para el error encontrado o se aborta la ejecución de la función.

Nota

código_error puede ser cualquiera de los descritos en el apéndice *PostgreSQL Error Codes* de la Documentación Oficial

https://www.postgresql.org/docs/9.5/static/errcodes-appendix.html

En el ejemplo siguiente se implementa una función que valida si una consulta pasada por parámetro puede ser ejecutada, arrojando excepciones en caso de tener errores de sintaxis, columnas indefinidas o tablas indefinidas.

Ejemplo 39: Función para el tratamiento de excepciones

```
CREATE FUNCTION valida_consulta(consulta varchar) RETURNS void AS
$$
BEGIN
   EXECUTE $1;
   EXCEPTION
      WHEN syntax_error THEN
```

```
        RAISE EXCEPTION 'Consulta con problemas de sintaxis';
    WHEN undefined_column OR undefined_table THEN
        RAISE EXCEPTION 'Columna o tabla no válida';
END;
$$ LANGUAGE plpgsql;

-- Invocación de la función
dell=# SELECT * FROM valida_consulta('SELECT * FROM catego');
ERROR: Columna o tabla no válida
```

Nótese que en el bloque de excepciones se han definido 2 condiciones a chequear para determinar si el error es de sintaxis o está relacionado con la invalidez de columnas o tablas, empleándose los códigos *syntax_error*, *undefined_column* y *undefined_table* definidos por el gestor para estos casos.

Uso de GET STACKED DIAGNOSTICS

Para obtener información acerca de la excepción arrojada en una función PL/pgSQL se pueden emplear las variables especiales de diagnóstico de errores y el comando GET STACKED DIAGNOSTICS, de la forma:

```
GET STACKED DIAGNOSTICS variable { = | := } elemento [ ,… ];
```

Donde cada elemento es una palabra clave que identifica un valor de estado asociado al error arrojado.

Nota

Para mayor detalle sobre las opciones de GET STACKED DIAGNOSTICS y las variables especiales analizar en la Documentación Oficial la sección *Trapping Errors* del capítulo *PL/pgSQL - SQL Procedural Language*

https://www.postgresql.org/docs/9.5/static/plpgsql-control-structures.html#PLPGSQL-ERROR-TRAPPING

En el ejemplo siguiente se reimplementa la función del ejemplo 39 ajustada al manejo genérico de errores utilizando el código de error OTHERS.

Nótese que, a diferencia del ejemplo 39, si el error no es de sintaxis, invalidez de columnas o tablas, no se muestra el mensaje definido por el sistema para el error encontrado o se aborta la ejecución de la función, sino que entra en la condición OTHERS, capturándose con GET STACKED DIAGNOSTICS el error y pudiendo el programador definir su propio mensaje.

Ejemplo 40: Función para el tratamiento genérico de excepciones

```
CREATE FUNCTION valida_consulta_others(consulta varchar) RETURNS void AS
$$
DECLARE
   mensaje text;
   mensaje_detalle text;
   sqlerror text;
BEGIN
   RAISE NOTICE 'Validando consulta...';
   BEGIN
   EXECUTE $1;
   EXCEPTION
      WHEN syntax_error THEN
         RAISE EXCEPTION 'Consulta con problemas de sintaxis';
      WHEN undefined_column OR undefined_table THEN
         RAISE EXCEPTION 'Columna o tabla no válida';
      WHEN OTHERS THEN
         GET STACKED DIAGNOSTICS mensaje = message_text, mensaje_detalle =
          pg_exception_detail, sqlerror = returned_sqlstate;
      RAISE EXCEPTION 'Otro error: %, %, %', sqlerror, mensaje,
       mensaje_detalle;
   END;
END;
$$ LANGUAGE plpgsql;

-- Invocación de la función
dell=# SELECT * FROM valida_consulta_others('SELECT category::date FROM
       categories');
NOTICE: Validando consulta...
ERROR: Otro error: 42846, no se puede convertir el tipo integer a date, SQL
state: P0001
```

3.8 Cursores

Para iterar por el resultado de una consulta y realizar operaciones sobre dicho resultado uno de los mecanismos implementados en el gestor ha sido los cursores. Un cursor es una variable que:

- Apunta al resultado de una consulta, permitiendo leer las tuplas resultantes.
- Permite situarse en filas específicas del resultado.
- Permite manipular una fila o conjunto de filas a partir de su posición actual.

Y aunque emplearlos implica reserva de recursos del sistema, tiempo en recuperar los datos y trabajar con ellos de forma independiente, son útiles para encapsular consultas

en lugar de ejecutarlas completas de una vez (pudiendo dividir el resultado en subconjuntos de pocas filas) y devolver una referencia a un cursor en una función permitiendo leer las filas en "lotes" (proveyendo una forma eficiente de devolver grandes conjuntos), esencialmente para evitar el desbordamiento de memoria.

Nota PL/pgSQL maneja internamente los ciclos FOR con cursores para evitar el desbordamiento de memoria

Pasos para el trabajo con cursores

Para el trabajo con cursores se requiere realizar una secuencia de pasos que incluye la declaración del cursor, su apertura, el procesamiento de los datos y su cierre.

Declarar el cursor

Declarar un cursor es similar a declarar cualquier otra variable en PL/pgSQL, existiendo 2 tipos de cursores según la forma en que se declaren, elección que depende del objetivo buscado al trabajar con el cursor, ver el ejemplo 41:

- Cursor ilimitado, puede abrirse para cualquier consulta, declarándose de la forma:

  ```
  nombre refcursor;
  ```

- Cursor limitado, puede abrirse solamente para una consulta específica, declarándose de la forma:

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

Ejemplo 41: Declaración de cursores

```
-- Cursor ilimitado
cursor1 refcursor;

-- Cursor para el trabajo exclusivo con todos los datos de la tabla categories
cursor2 CURSOR FOR SELECT * FROM categories;

-- Cursor para el trabajo con todos los datos de productos donde special tenga
-- un valor pasado por parámetro
cursor3 CURSOR (esp integer) FOR SELECT * FROM products WHERE special = esp;
```

Abrir el cursor

Para trabajar con el cursor, una vez definido, debe abrirse; operación que se realiza mediante el comando OPEN, con el que se reserva memoria suficiente para el cursor, se ejecuta la consulta asociada (que debe ser un SELECT o alguna que retorne tuplas) y se posiciona el puntero antes de la primera fila del resultado de la consulta.

El comando OPEN puede emplearse en tres escenarios, ver el ejemplo 42:

- El cursor fue declarado como ilimitado y se abre para una consulta específica, en cuyo caso se emplea la sintaxis:
  ```
  OPEN cursor_ilimitado FOR consulta;
  ```

- El cursor fue declarado como ilimitado y se abre para una consulta, generalmente dinámica, especificada entre comillas simples, en cuyo caso se emplea la sintaxis:
  ```
  OPEN cursor_ilimitado FOR EXECUTE 'consulta';
  ```

- El cursor fue declarado limitado para ser empleado con una única consulta, en cuyo caso se emplea la sintaxis:
  ```
  OPEN cursor_limitado;
  ```

Ejemplo 42: Apertura de cursores

```
-- Abrir cursor1 para la consulta SELECT * FROM products
OPEN cursor1 FOR SELECT * FROM products;
-- Abrir cursor1 para mostrar todos los datos de una tabla pasada por parámetro
OPEN cursor1 FOR EXECUTE 'SELECT * FROM ' || quote_ident($1)';
-- Abrir cursor2 que es limitado
OPEN cursor2;
-- Abrir cursor3 con el parámetro esp con valor 1, de una de las 2 formas
-- siguientes
OPEN cursor3(1) | OPEN cursor3(esp := 1);
```

Procesar el cursor

Una vez abierto el cursor se debe trabajar con el resultado de la consulta, para lo que se pueden emplear 3 comandos:

- FETCH: guarda el valor de la tupla a la que apunta el cursor en un contenedor (variables de tipo fila, RECORD o una lista separada por comas) moviendo el puntero hacia la siguiente posición definida en *dirección*, con la sintaxis:
  ```
  FETCH [ dirección { FROM | IN } ] variable_cursor INTO contenedor;
  ```

- MOVE: reposiciona el cursor según lo definido en *dirección* sin recuperar ningún dato, con la sintaxis:
  ```
  MOVE [ dirección { FROM | IN } ] variable_cursor;
  ```

- CURRENT OF: identifica la tupla en consultas de actualización o eliminación, con la sintaxis:
  ```
  UPDATE | DELETE … WHERE CURRENT OF variable_cursor;
  ```

En los comandos FETCH y MOVE *dirección* toma por defecto el valor NEXT (con el que se almacena o mueve a la siguiente tupla del resultado), pero puede tomar otros valores como PRIOR, FIRST, LAST, ALL, ABSOLUTE *cantidad*, etc., a emplear en función del procesamiento del resultado de la consulta que se requiera realizar.

Nota

Para mayor detalle sobre las opciones para el establecimiento de la dirección en los comandos FETCH y MOVE analizar en la Documentación Oficial la sintaxis de ambos comandos

https://www.postgresql.org/docs/9.5/static/sql-commands.html

El ejemplo 43 muestra el empleo de los comandos FETCH, MOVE Y CURRENT OF para el procesamiento de los datos resultantes de la consulta para la que fue abierto el cursor.

Ejemplo 43: Procesamiento de cursores

```
-- Guardar en var_producto la tupla a que apunta cursor1
FETCH cursor1 INTO var_producto;
-- Guardar en id_categoria y nombre_categoria los atributos de la tupla a que
-- apunta cursor2
FETCH cursor2 INTO id_categoria, nombre_categoria;
-- Guardar en var_producto la última tupla de la consulta
FETCH LAST FROM cursor3 INTO var_producto;
-- Guardar en id_categoria y nombre_categoria, a partir de la posición actual
-- del cursor, la tupla anterior a la actual
FETCH RELATIVE -1 FROM cursor2 INTO id_categoria, nombre_categoria;
-- Mover el puntero de cursor2 2 tuplas anteriores a la posición actual
MOVE RELATIVE -2 FROM cursor2;
-- Mover el puntero de cursor2 a la tupla siguiente
MOVE FORWARD 1 FROM cursor2;
-- Actualizar el precio del producto al que apunta cursor1
UPDATE products SET price = price * 2 WHERE CURRENT OF cursor1;
```

Cerrar el cursor

Procesado el resultado de la consulta asociada al cursor debe cerrarse con el comando CLOSE, con el que se liberan, sin necesidad de esperar a que termine la transacción, los recursos asignados cuando se abrió y la variable para que pueda ser utilizada nuevamente (útil en el caso de las variables ilimitadas). La sintaxis para cerrar el cursor es la siguiente:

```
CLOSE variable_cursor;
```

Uso de cursores para iterar por el resultado de una consulta

Uno de los usos que se les puede dar a un cursor es para iterar para el resultado de una consulta y realizar cierto procesamiento sobre el resultado. En el ejemplo siguiente se implementa una función que, a aquellos pedidos realizados en la fecha pasada por parámetro, se rebaje a la cantidad el valor también pasado por parámetro, mostrando, además, los datos de dichos pedidos.

Ejemplo 44: Empleo de un cursor para iterar por el resultado de una consulta

```
CREATE FUNCTION aumentar_cantidad(p_orderdate date, p_quantity smallint)
 RETURNS SETOF orderlines AS
$$
DECLARE
   c1 CURSOR FOR SELECT * FROM orderlines WHERE orderdate=$1;
   res orderlines;
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO res;
      EXIT WHEN not found;
      UPDATE orderlines SET quantity=quantity-$2 WHERE orderid=res.orderid AND
      prod_id=res.prod_id;
      RETURN NEXT res;
   END LOOP;
   CLOSE c1;
   RETURN;
END;
$$ LANGUAGE plpgsql;
```

Note que en el ejemplo 44 se utiliza la estructura LOOP para iterar por el resultado de la consulta, con la particularidad de que se utiliza un cursor que apunta a una lista de todos los datos de los pedidos realizados en la fecha pasada por parámetro, guardándose cada tupla, en cada iteración del LOOP, en una variable de tipo *orderlines*, para luego actualizar la cantidad y agregar al resultado la tupla modificada.

Retorno de cursores desde una función

Probablemente, el retorno de cursores desde una función sea el mejor uso que se le pueda dar a los mismos aprovechando la potencialidad que ofrecen cuando se trabaja con grandes volúmenes de datos. La idea es declararlo, abrirlo y retornarlo dentro de la función para luego procesarlo en una transacción.

La función implementada en el ejemplo 45 retorna un cursor con el listado de las categorías. Nótese que el procesamiento de los datos del cursor no se realiza en la función sino en el bloque de transacciones una vez invocada y mediante la opción ALL de FETCH.

Ejemplo 45: Empleo de un cursor para retornar datos desde una función

```
CREATE FUNCTION mostrar_categorias() RETURNS refcursor AS
$$
DECLARE
   ref refcursor;
BEGIN
   OPEN ref FOR SELECT category, categoryname FROM categories;
   RETURN ref;
END;
$$ LANGUAGE plpgsql;

-- Invocar la función dentro de una transacción
dell=# BEGIN;

-- Invocar la función otorgándosele un nombre por defecto, en este caso
-- <unnamed portal 1>
dell=# SELECT mostrar_categorias();
 mostrar_categorias
--------------------
 <unnamed portal 1>
(1 fila)

-- Sacar la información del cursor
dell=# FETCH ALL IN "<unnamed portal 1>";
 category | categoryname
----------+--------------
        1 | Action
        2 | Animation
        3 | Children
        4 | Classics
        5 | Comedy
        6 | Documentary
        7 | Drama
        8 | Family
        9 | Foreign
       10 | Games
       11 | Horror
       12 | Music
       13 | New
       14 | Sci-Fi
```

```
       15 | Sports
       16 | Travel
(16 filas)

-- Cerrar el cursor
dell=# CLOSE "<unnamed portal 1>";

-- Cerrar la transacción
dell=# COMMIT;
```

El ejemplo 46 reimplementa la función anterior para, en este caso, definir un nombre para el cursor de forma que sea más cómodo su manejo. Note que al invocar la función en la transacción se retornan los datos de 8 en 8 tuplas utilizando FETCH *cantidad*, método útil para paginar el resultado desde la aplicación que solicite el resultado del cursor.

Ejemplo 46: Empleo de un cursor para retornar datos desde una función, usando un nombre para el cursor y dividiendo el resultado en lotes

```
CREATE FUNCTION mostrar_cat_nombre(ref refcursor) RETURNS refcursor AS
$$
BEGIN
   OPEN ref FOR SELECT category,categoryname FROM categories;
   RETURN ref;
END;
$$ LANGUAGE plpgsql;

-- Invocar la función dentro de una transacción
dell=# BEGIN;

-- Invocar la función
dell=# SELECT mostrar_cat_nombre('categorias');
 mostrar_categorias
--------------------
 categorias
(1 fila)

-- Sacar la información del cursor
dell=# FETCH 8 IN "categorias";
 category | categoryname
----------+--------------
        1 | Action
        2 | Animation
        3 | Children
        4 | Classics
        5 | Comedy
```

```
        6 | Documentary
        7 | Drama
        8 | Family
(8 filas)

dell=# FETCH 8 IN "categorias";
 category | categoryname
----------+--------------
        9 | Foreign
       10 | Games
       11 | Horror
       12 | Music
       13 | New
       14 | Sci-Fi
       15 | Sports
       16 | Travel
(8 filas)

dell=# CLOSE "categorias";
dell=# COMMIT;
```

3.9 Disparadores

Para ejecutar alguna actividad o acción en la base de datos es necesario que se invoque una función que contendrá el código que se desea ejecutar, como se ha visto hasta ahora, pero ¿cómo ejecutar, automáticamente, una acción en la base de datos cuando ocurra algún evento específico en las tablas de la misma? Para dar solución a este problema existen los desencadenadores o disparadores, conocidos mayormente como *triggers* (del inglés).

Un disparador es una acción que se ejecuta de forma automática cuando se cumple una condición establecida al realizar una operación sobre la base de datos. Por ejemplo, puede ser la realización de una actividad determinada antes de insertar un nuevo registro en la tabla *categories*.

Algunas ventajas de la utilización de disparadores son las siguientes:

- Se ejecutan automáticamente cuando ocurre determinada actividad en la base de datos.
- Permiten la implementación de requisitos complejos mientras se manejan los datos.

- Pueden exigir restricciones más complejas que las definidas con CHECK.
- Son un buen modo de realizar auditorías en las bases de datos.

Disparadores en PostgreSQL

PostgreSQL permite la implementación de disparadores mediante la definición de una función disparadora y, luego, del disparador que invoque a dicha función; esto posibilita a varios disparadores utilizar la misma función sin necesidad de reescribir código.

La definición de la función disparadora es similar a la de una función normal de PL/pgSQL, con la peculiaridad de que no se especifican parámetros y que debe retornar un tipo de dato especial llamado TRIGGER. Dentro de la función se puede escribir código en PL/pgSQL y se debe garantizar que se retorne algún valor, ya sea NULL, RECORD o una fila con la misma estructura de la tabla que lo invoca.

El ejemplo 47 muestra una función disparadora simple que notifica que ha sido invocado un disparador.

Ejemplo 47: Implementación de una función disparadora

```
CREATE FUNCTION funcion_disparadora() RETURNS trigger AS
$$
BEGIN
   RAISE NOTICE 'Un disparador ha sido invocado';
   RETURN null;
END;
$$ LANGUAGE plpgsql;
```

Para que esta función se ejecute debe definirse un disparador, que la invoque cuando ocurra determinado evento sobre determinado objeto de la base de datos y en determinado momento.

La sintaxis básica para la definición de un disparador se muestra a continuación:

```
CREATE TRIGGER nombre
{ BEFORE | AFTER | INSTEAD OF } { evento [ OR… ] }
ON tabla
[ FOR EACH { ROW | STATEMENT } ]
[ WHEN ( condición ) ]
EXECUTE PROCEDURE función_disparadora();
```

Donde *evento* puede ser INSERT, UPDATE, DELETE o TRUNCATE.

Nota La sintaxis ampliada puede analizarse en la Documentación Oficial en la sección *SQL Commands*

https://www.postgresql.org/docs/9.5/static/sql-createtrigger.html

Un ejemplo de disparador para la invocación de la función definida anteriormente puede ser el mostrado en el ejemplo 48, donde se especifica que la función disparadora previamente definida se ejecutará después de haberse insertado un registro en la tabla *categories*.

Ejemplo 48: Disparador que invoca la función disparadora del ejemplo 40

```
CREATE TRIGGER tgr_categories_ins
AFTER INSERT
ON categories
FOR EACH ROW
EXECUTE PROCEDURE funcion_disparadora();
```

En el ejemplo 49 se define un disparador que invocará la función disparadora cuando ocurra más de un evento, a diferencia del ejemplo anterior que solo la invocaba cuando se realizaba una inserción.

Ejemplo 49: Disparador que invoca la función disparadora cuando se realiza una inserción, actualización o eliminación sobre la tabla categories

```
CREATE TRIGGER tgr_categories_ins_up_del
AFTER INSERT OR UPDATE OR DELETE
ON categories
FOR EACH ROW
EXECUTE PROCEDURE funcion_disparadora();
```

Para definir cuál de los eventos invocó la función disparadora, PostgreSQL habilita una serie de variables especiales que contienen información sobre el disparador, como cuándo ocurrió, qué evento, etc.:

- TG_OP: variable de tipo cadena que indica qué tipo de evento está ocurriendo (INSERT, UPDATE, DELETE, TRUNCATE), siempre en mayúscula.
- TG_WHEN: variable de tipo cadena que indica el momento en que se invocará el disparador (BEFORE, AFTER, INSTEAD OF), siempre en mayúscula.

- TG_RELNAME o TG_TABLE_NAME: variable de tipo cadena que almacena el nombre de la tabla sobre la cual se estaba trabajando cuando se invocó el disparador.
- NEW: variable de tipo RECORD que almacena los nuevos valores de la fila que se están insertando o modificando.
- OLD: variable de tipo RECORD que almacena los valores antiguos de la fila que se están modificando o eliminando.

Nota

Existen otras variables especiales que se pueden analizar en detalle en la sección *Triggers on data changes* del epígrafe *Trigger Procedures*, en la Documentación Oficial

http://www.postgresql.org/docs/9.5/static/plpgsql-trigger.html

FOR EACH en la definición del disparador

Definida la función disparadora y el disparador, este estará observando la tabla para la que fue creado y, una vez que ocurra el evento especificado se invocará la función. El ejemplo 50 muestra una consulta que ejecuta el disparador definido sobre la tabla *categories* en el ejemplo anterior.

Ejemplo 50: Ejecución de una consulta que activa el disparador creado sobre la tabla categories

```
dell=# UPDATE categories SET categoryname = upper(categoryname) WHERE category
       >= 15;
NOTICE: Un disparador ha sido invocado
NOTICE: Un disparador ha sido invocado
UPDATE 2
```

En el ejemplo anterior puede verse cómo se ejecuta la función disparadora en dos ocasiones ya que fueron lanzadas dos notificaciones, esto indica que fueron afectadas con la consulta dos filas, y está dado porque en la definición del disparador se empleó el FOR EACH ROW, que significa que el disparador se ejecuta por cada fila afectada. Sin embargo, si se define en su lugar FOR EACH STATEMENT, la función disparadora se ejecutaría en una sola ocasión pues se le indica que se ejecutará por cada sentencia SQL ejecutada.

Si se define el disparador como se muestra en el ejemplo 51 y se ejecuta la misma sentencia de actualización, se mostraría otro mensaje indicando su ejecución una sola vez; note en este ejemplo que se ejecutan los disparadores definidos en los ejemplos 49 y 51.

Ejemplo 51: Definición de la función disparadora y el disparador sobre la tabla categories para chequear modificaciones

```
CREATE FUNCTION tgr_para_modificacion() RETURNS trigger AS
$$
BEGIN
   RAISE NOTICE 'Un disparador se ha invocado para modificación';
   RETURN null;
END;
$$ LANGUAGE plpgsql;

-- Disparador que invoca la función
CREATE TRIGGER tgr_modificacion
AFTER UPDATE OR DELETE
ON categories
FOR EACH STATEMENT
EXECUTE PROCEDURE tgr_para_modificacion();

-- Invocación del disparador al ejecutarse una actualización sobre categories
dell=# UPDATE categories SET categoryname = upper(categoryname) WHERE category
       >= 15;
NOTICE: Un disparador ha sido invocado
NOTICE: Un disparador ha sido invocado
NOTICE: Un disparador se ha invocado para modificación
UPDATE 2
```

Es válido destacar que la sentencia FOR EACH STATEMENT se ejecuta siempre que la consulta SQL cumpla con alguno de los eventos sobre la tabla para la cual fue definida aun cuando no afecte ninguna fila; a diferencia de FOR EACH ROW que solo se ejecuta cuando se afecta al menos una fila. El ejemplo 52 demuestra lo anterior al invocar el disparador *tgr_modificacion* la función una vez ejecutada una actualización sobre la tabla *categories*.

Ejemplo 52: Ejecución de tgr_modificacion en lugar de tgr_categories_ins debido a que no se actualiza ninguna tupla sobre la tabla categories

```
dell=# UPDATE categories SET categoryname = upper(categoryname) WHERE category
       >= 30;
NOTICE: Un disparador se ha invocado para modificación
UPDATE 0
```

Note que en este ejemplo el disparador que se ejecuta es *tgr_modificacion* en lugar de *tgr_categories_ins*, porque el primero utiliza la sentencia FOR EACH STATEMENT en lugar de FOR EACH ROW y la consulta no actualiza ninguna fila.

Nota El evento TRUNCATE solo puede ser utilizado con la sentencia FOR EACH STATEMENT, al afectar siempre a todas las tuplas de la tabla sobre la que se realiza

Retorno de la función disparadora

Otro aspecto a tener en cuenta para el trabajo con disparadores en PostgreSQL es el retorno de la función disparadora. Si el disparador se define con la opción:

- BEFORE: y la función retorna NULL, las tuplas indicadas en la sentencia SQL no se verán afectadas; esto posibilita que, incluso, se pueda modificar el valor del nuevo registro en determinada tabla antes de insertarlo o, simplemente, si no cumple con determinada condición la decisión sea no insertarlo.
- AFTER: la función puede retornar NULL, NEW o cualquier otro valor que cumpla con los requisitos de devolución de la función, debido a que ya el valor está insertado en la tabla.

El ejemplo 53 muestra cómo un disparador permite modificar el valor de un nuevo registro antes de insertarse. En este caso si se inserta un nuevo cliente y el nombre no comienza con mayúscula se modifica y se inserta el mismo con esta condición cumplida.

Ejemplo 53: Disparador que permite modificar un nuevo registro antes de insertarlo

```
CREATE FUNCTION mayuscula_nombre() RETURNS trigger AS
$$
DECLARE
   resultado text;
BEGIN
   IF ascii(substring(NEW.firstname FROM 1 FOR 1)) >= 65 AND ascii(substring
    (NEW.firstname FROM 1 FOR 1)) <= 90 THEN
      RAISE NOTICE 'Formato correcto';
      RETURN NEW;
   ELSE
      resultado := upper(substring(NEW.firstname FROM 1 FOR 1)) || substring
       (NEW.firstname FROM 2 FOR length(NEW.firstname));
      RAISE NOTICE 'Formato incorrecto, % es el correcto', resultado;
      NEW.firstname := resultado;
      RETURN NEW;
   END IF;
END;
$$ LANGUAGE plpgsql;

-- Definición del disparador
CREATE TRIGGER tgr_modificar_insercion
BEFORE INSERT
```

```
ON customers
FOR EACH ROW
EXECUTE PROCEDURE mayuscula_nombre();

-- Ejecución de una inserción sobre la tabla customers
dell=# INSERT INTO customers(firstname, lastname, address1, city, country,
       region, creditcardtype, creditcard, creditcardexpiration, username,
       password) VALUES ('Anthony', 'Sotolongo', 'dir1', 'Cienfuegos',
       'Cienfuegos', 1, 2, 1, 'una tarjeta de crédito', 'asotolongo',
       'mipasss');
NOTICE: Formato correcto
INSERT 0 1

-- Inserción en customers sin letra inicial del nombre en mayúscula
dell=# INSERT INTO customers(firstname, lastname, address1, city, country,
       region, creditcardtype, creditcard, creditcardexpiration, username,
       password) VALUES ('luis', 'Sotolongo', 'dir1', 'Cienfuegos',
       'Cienfuegos', 1, 2, 1, 'una tarjeta de crédito', 'lsotolongo',
       'mipasss');
NOTICE: Formato incorrecto, Luis es el correcto
INSERT 0 1
```

Este comportamiento puede ser utilizado para evitar eliminaciones, ver el ejemplo 54. Note, en este ejemplo, que se implementa un disparador para garantizar que no se eliminen datos de la tabla *orders*, el cual antes de intentar eliminarse un registro invoca una función disparadora que retorna NULL.

Ejemplo 54: Disparador que no permite eliminar tuplas de una tabla

```
CREATE FUNCTION asegurar_datos() RETURNS trigger AS
$$
BEGIN
   RAISE NOTICE 'No se pueden eliminar datos de esta tabla';
   RETURN NULL;
END;
$$ LANGUAGE plpgsql;

-- Definición del disparador
CREATE TRIGGER tgr_asegurar_datos
BEFORE DELETE
ON orders
FOR EACH ROW
EXECUTE PROCEDURE asegurar_datos();

--Ejecución de una consulta de eliminación sobre la tabla orders
dell=# DELETE FROM orders WHERE orderid=11000;
NOTICE: No se pueden eliminar datos de esta tabla
DELETE 0
```

Haciendo auditoría con disparadores

Uno de los empleos más significativos de los disparadores es para el control de cambios, una especie de registro de cambios de determinada tabla. Ello es posible mediante el uso de las variables especiales NEW y OLD, que permiten acceder a los nuevos y antiguos valores respectivamente.

El ejemplo 55 muestra cómo se almacenan en una tabla, llamada registro, los cambios de modificación o eliminación realizados sobre *customers*, donde, además, se almacenará la operación y la fecha en que fue realizada.

Ejemplo 55: Forma de realizar auditorías sobre la tabla customers

```
-- Tabla donde se almacenarán operaciones sobre customers
CREATE TABLE registro(accion text, fecha timestamp, antiguo text, nuevo text);

-- Función disparadora para registrar las operaciones
CREATE FUNCTION auditoria_clientes() RETURNS trigger AS
$$
BEGIN
   IF TG_OP = 'UPDATE' THEN
      RAISE NOTICE 'Se realizó una actualización sobre la tabla customers...';
      INSERT INTO registro VALUES (TG_OP, now(), OLD::text, NEW::text);
   END IF;
   IF TG_OP = 'DELETE' THEN
      RAISE NOTICE 'Se realizó una eliminación sobre la tabla customers...';
      INSERT INTO registro VALUES (TG_OP, now(),OLD::text,'');
   END IF;
   RETURN null;
END;
$$ LANGUAGE plpgsql;

--Disparador
CREATE TRIGGER tgr_aud_clientes
AFTER DELETE OR UPDATE
ON customers
FOR EACH ROW
EXECUTE PROCEDURE auditoria_clientes();

-- Ejemplo de operación DELETE
dell=# DELETE FROM customers WHERE customerid = 10000;
NOTICE: Se realizó una eliminación sobre la tabla customers...
DELETE 1
```

```
-- Datos en tabla registro
dell=# SELECT * FROM registro;
   accion   |          fecha          |       antiguo       | nuevo
------------+-------------------------+---------------------+-------
 DELETE     | 2016-09-02 13:35:20.25  | (10000, EUQOTG,...) |
(1 fila)
```

Nota

Pueden utilizarse tipos de datos con mejores descripciones para el antiguo y nuevo valor como JSON y HSTORE

Disparadores por columnas y condicionales

Desde la versión 9.0 de PostgreSQL se pueden definir disparadores por columnas sobre sentencias UPDATE y establecer condiciones para que se ejecute el disparador. Esto puede incidir en el rendimiento al no ser necesario programar en la función disparadora lógica de negocio para chequear determinada condición; significa, que si se establecen estas condiciones en la definición del disparador no es necesario hacer la llamada a la función de no cumplirse con dichas condiciones. Los disparadores por columnas se ejecutan cuando una o varias columnas determinadas por el usuario se actualizan. La sintaxis para definir uno de este tipo es la siguiente:

```
CREATE TRIGGER nombre
{ BEFORE | AFTER | INSTEAD OF } { UPDATE OF columna }
ON tabla
[ FOR EACH { ROW | STATEMENT } ]
EXECUTE PROCEDURE nombre_función();
```

Un ejemplo de cómo utilizarlo puede ser que cuando se modifique el valor del precio de algún producto se almacene en la tabla registro; el ejemplo 56 lo muestra.

Ejemplo 56: Función disparadora y disparador por columnas para controlar las actualizaciones sobre la columna price de la tabla products

```
CREATE FUNCTION registrar_update() RETURNS trigger AS
$$
BEGIN
   RAISE NOTICE 'Operación UPDATE sobre columna price';
   INSERT INTO registro VALUES (TG_OP, now(), OLD::text, NEW::text);
   RETURN null;
END;
$$ LANGUAGE plpgsql;
```

```
-- Definición del disparador por columna
CREATE TRIGGER tgr_registro_update1
AFTER UPDATE OF price
ON products
FOR EACH ROW
EXECUTE PROCEDURE registrar_update();
```

Note en este ejemplo que el disparador invocará a la función *registrar_update*, no cuando se realice una actualización sobre cualquier atributo de *products* sino, específicamente, sobre el atributo *price*. Sin hacer uso de esta funcionalidad debería chequearse esta condición en la función disparadora, consumiendo recursos al invocarla que pudieran evitarse si la actualización no fuera sobre *price*.

Los disparadores condicionales permiten especificar condiciones en su definición, pudiendo escribirse cualquier expresión con resultado lógico haciendo uso de los operadores lógicos (AND, OR, etc.), exceptuando subconsultas.

El ejemplo anterior se pudiera reimplementar haciendo uso de la cláusula WHEN como muestra el ejemplo 57.

Ejemplo 57: Definición de un disparador condicional para chequear que se actualice el precio

```
CREATE TRIGGER tgr_registro_update2
AFTER UPDATE
ON products
FOR EACH ROW
WHEN (OLD.price IS DISTINCT FROM NEW.price)
EXECUTE PROCEDURE registrar_update();
```

Note que en este caso, al igual que en el ejemplo anterior, el disparador se activará cuando se realice una actualización sobre la tabla *products* que, además, cumpla la condición de que se haya actualizado el atributo *price*.

Otro ejemplo de su uso pudiera ser que verificara que los nuevos valores del atributo actor comiencen con mayúscula; para que se ejecute la función disparadora se debe definir un disparador del siguiente modo. Note que se puede hacer eso en la definición del disparador de operadores lógicos.

Ejemplo 58: Disparador condicional para chequear que actor comienza con mayúsculas

```
CREATE TRIGGER tgr_registro_update3
AFTER UPDATE ON products
```

```
FOR EACH ROW
WHEN (ascii(substring(NEW.actor FROM 1 FOR 1) ) >= 65 AND ascii(substring(
 NEW.actor FROM 1 FOR 1)) <= 90)
EXECUTE PROCEDURE registrar_update();
```

Disparadores sobre vistas

Desde la versión 9.1 PostgreSQL soporta disparadores sobre vistas, siempre y cuando el disparador se defina haciendo uso de alguna de las opciones:

- INSTEAD OF y FOR EACH ROW
- BEFORE | AFTER y FOR EACH STATEMENT

La vista en sí no se puede manipular con DDL (INSERT, DELETE y UPDATE), pero esta versión de PostgreSQL ya lo posibilita siempre y cuando se realice la acción de forma manual en las respectivas tablas involucradas. Es decir, en la función disparadora se debe escribir el código que realice dicha actividad en las tablas relacionadas en la vista. El ejemplo 59 describe cómo hacer un disparador para que sea actualizada la vista *inventario_de_productos*.

Ejemplo 59: Función disparadora y disparador para actualizar una vista

```
-- Definición de la vista inventario_de_productos
CREATE VIEW inventario_de_productos AS (
   SELECT i.quan_in_stock, i.sales, p.title, i.prod_id
   FROM inventory i, products p
   WHERE p.prod_id = i.prod_id);

-- Definición de la función disparadora
CREATE FUNCTION actualizar_vista() RETURNS trigger AS
$$
BEGIN
   RAISE NOTICE 'Disparador sobre la vista inventario_de_productos...';
   IF (NEW.sales <> OLD.sales OR NEW.quan_in_stock <> OLD.quan_in_stock) THEN
      RAISE NOTICE 'Actualizando ventas y stock en inventory...';
      UPDATE inventory SET sales = NEW.sales, quan_in_stock =
       NEW.quan_in_stock WHERE prod_id = NEW.prod_id;
   END IF;
   IF NEW.title <> OLD.title THEN
      RAISE NOTICE 'Actualizando nombre del producto: % por %', OLD.title,
       NEW.title;
      UPDATE products SET title = NEW.title WHERE prod_id = NEW.prod_id;
   END IF;
```

```
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Definición del disparador
CREATE TRIGGER tgr_actualizar_vista
INSTEAD OF UPDATE
ON inventario_de_productos
FOR EACH ROW
EXECUTE PROCEDURE actualizar_vista();

-- Realizando UPDATE sobre la vista
dell=# UPDATE inventario_de_productos SET title = upper('Conducta') WHERE
       prod_id = 12;
NOTICE: Disparador sobre la vista inventario_de_productos...
NOTICE: Actualizando el nombre del producto: ACADEMY ALASKA por CONDUCTA
UPDATE 1

-- Consultando la tabla products para chequear actualización efectiva
dell=# SELECT title FROM products WHERE prod_id = 12;
   title
-----------
 CONDUCTA
(1 fila)
```

Disparadores sobre eventos

Los disparadores sobre los eventos DDL son soportados en PostgreSQL desde la versión 9.3, permitiendo realizar alguna rutina de función cuando se ejecuta un comando DDL. La sintaxis para ello es la siguiente:

```
CREATE EVENT TRIGGER nombre
ON evento
[ WHEN variable_filtro IN ( valor_filtro [ , ... ] ) [ AND ... ] ]
EXECUTE PROCEDURE nombre_función();
```

Donde:

- *evento*: puede tomar los valores *ddl_command_start*, *ddl_command_end* y *sql_drop*.
- WHEN: especifica el filtro para que se ejecute o no la función disparadora, donde *variable_filtro* debe tomar el valor TAG y *valor_filtro* debe ser cualquiera de los comandos DDL listados en la sección *Event Trigger Firing Matrix* de la

- Documentación Oficial; por ejemplo, CREATE TABLE, DROP TABLE, etc.

Nota Más detalles sobre las especificidades de los valores de *evento* y *valor_filtro* en las secciones *Overview of Event Trigger Behavior* y *Event Trigger Firing Matrix*, respectivamente, del capítulo *Event Triggers* en la Documentación Oficial

http://www.postgresql.org/docs/9.5/static/event-triggers.html

Existen para el trabajo con estos disparadores las variables especiales siguientes:

- TG_EVENT: variable de tipo texto que especifica el evento.
- TG_TAG: variable de tipo texto que especifica el comando que se ejecutó (CREATE TABLE, ALTER TABLE, etc.).

Un ejemplo de la utilización de disparadores sobre eventos puede ser registrar cuándo se realizaron en la base de datos algunas actividades de creación o eliminación de una tabla, como muestra el ejemplo 60.

Ejemplo 60: Uso de disparadores sobre eventos para registrar actividades de creación y eliminación sobre tablas de la base de datos

```
-- Definición de la tabla registro_evento
CREATE TABLE registro_eventos (evento text, fecha timestamp);

-- Definición de la función disparadora
CREATE FUNCTION registrar_evento() RETURNS event_trigger AS
$$
BEGIN
   RAISE NOTICE 'Evento: %, Horario: %', TG_TAG, now();
   INSERT INTO registro_eventos VALUES (TG_TAG, now());
END;
$$ LANGUAGE plpgsql;

-- Definición del disparador
CREATE EVENT TRIGGER tgr_evento
ON ddl_command_start
WHEN TAG IN ('CREATE TABLE', 'DROP TABLE')
EXECUTE PROCEDURE registrar_evento();

-- Creando una tabla
dell=# CREATE TABLE usuarios (nombre text, usuario text, pass text);
NOTICE: Evento: CREATE TABLE, Horario: 2015-06-24 06:33:32.773
```

```
-- Consultando la tabla registro_eventos para chequear registro efectivo
dell=# SELECT * FROM registro_eventos;
    evento    |          fecha
--------------+-------------------------
 CREATE TABLE | 2015-06-24 06:33:32.773
(1 fila)
```

3.10 Depurado

Las operaciones de depurado (*debug* en inglés) se utilizan, sobre todo, en tiempos de desarrollo y son útiles, por ejemplo, para conocer el camino que está tomando un bloque de código o el valor de una variable, entre otros escenarios.

Depurado con RAISE

Una forma sencilla de hacer depurado es utilizando el comando RAISE. El ejemplo 61 evidencia el uso de las opciones NOTICE y LOG para obtener el valor de un parámetro en varios momentos de la ejecución de la función. En el caso de LOG para guardar en los registros de PostgreSQL el valor de la variable, útil si se requiere dejar constancia de los valores para análisis futuros.

Ejemplo 61: Empleo de RAISE para realizar DEBUG

```
-- Utilizando la opción NOTICE
CREATE FUNCTION ejemplo_debug_notice(parametro integer) RETURNS void AS
$$
BEGIN
   RAISE NOTICE 'El valor del parámetro es %', parametro;
   IF parametro > 10 THEN
      RAISE NOTICE 'El valor del parámetro es mayor que 10 y es %', parametro;
   END IF;
   parametro := parametro + 20;
   RAISE NOTICE 'El valor del parámetro una vez sumado con 20 es %',
    parametro;
END;
$$ LANGUAGE plpgsql;

-- Utilizando la opción LOG
CREATE FUNCTION ejemplo_debug_log(parametro integer) RETURNS void AS
$$
BEGIN
   RAISE LOG 'El valor del parámetro es %', parametro;
   IF parametro > 10 THEN
```

```
    RAISE LOG 'El valor del parámetro es mayor que 10 y es %', parametro;
  END IF;
  parametro := parametro + 20;
  RAISE LOG 'El valor del parámetro una vez sumado con 20 es %', parametro;
END;
$$ LANGUAGE plpgsql;
```

Depurado con PLDEBUGGER

PostgreSQL permite realizar el depurado utilizando una extensión llamada pldebugger que se integra con el pgAdmin; para su uso se debe:

- Instalar la extensión en el servidor PostgreSQL.
- Modificar el valor de la variable *shared_preload_libraries* a *'plugin_debugger'* en el postgresql.conf y reiniciar el servidor de bases de datos para que asuma la nueva configuración.
- Crear en el pgAdmin la extensión con el comando *create extension pldbgapi*.

Una vez creada la extensión se puede emplear realizando las siguientes acciones:

- Dar clic derecho sobre la función que se desea depurar, clic en la opción *Debugging* y seleccionar *Debug*, con lo que aparecerá una ventana como la mostrada en la figura siguiente para colocar los valores de los parámetros de entrada de la función.

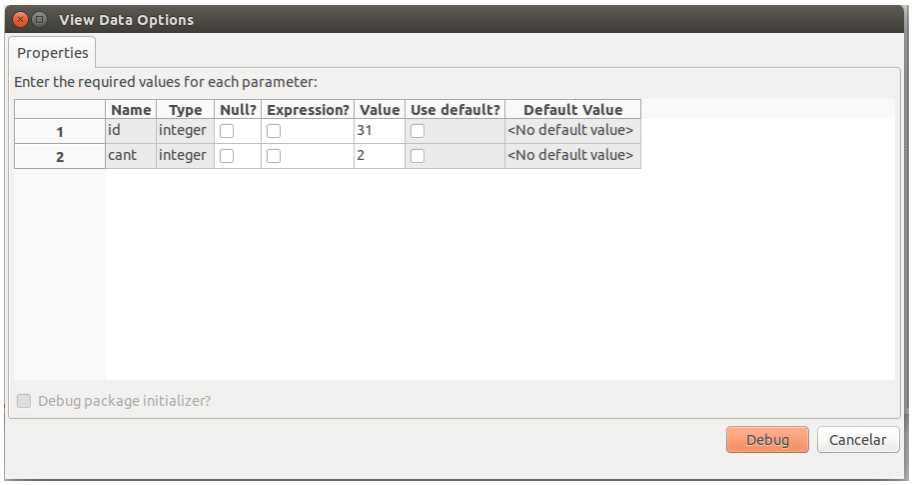


Figura 3: Opciones de vista de datos de la extensión pldebugger

- Realizar el recorrido por el código en la ventana que aparece al seleccionar la opción *Debug* mediante el empleo de los íconos de la barra superior o con los atajos Ctrl+F11 o Ctrl+F10, Ctrl+F5, como muestra la figura siguiente.

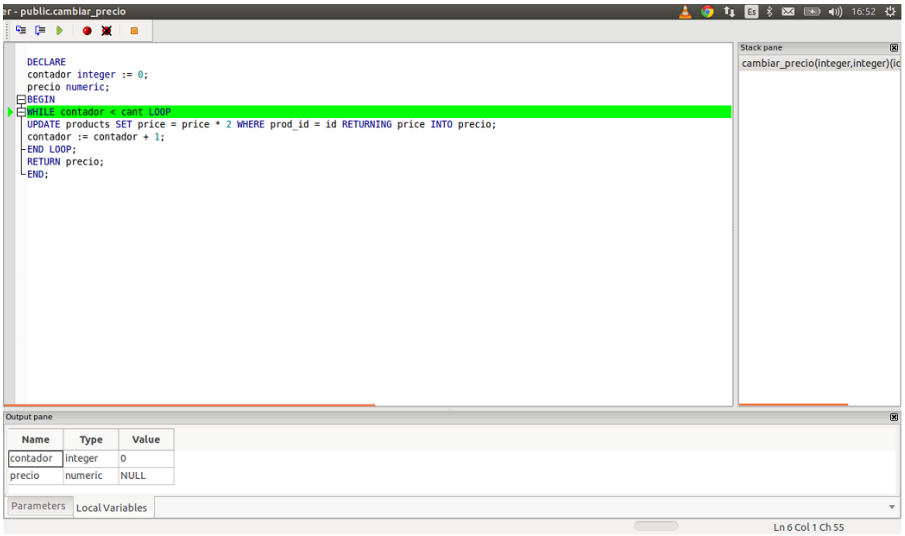


Figura 4: Depurado de una función con la extensión pldebugger

De esta forma, como muestra la figura 4, se puede ir observando el camino que toma el código, así como el valor de las variables locales de la función en la parte inferior de la ventana.

3.11 Resumen

Las funciones en PL/pgSQL son la forma más utilizada de escribir lógica de negocio del lado del servidor, pues este lenguaje da la posibilidad de emplear varios componentes de la programación en ellas, como variables, estructuras iterativas y condicionales, paso de mensajes, disparadores, cursores, etc. Además, permiten la ejecución de sentencias SQL dentro de ellas, pudiéndose manipular los datos de la base de datos.

Para crear una función PL/pgSQL 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; su cuerpo está estructurado por los bloques de declaración (DECLARE, opcional) y de sentencias (acotado por las palabras reservadas BEGIN y END).

Las funciones en PL/pgSQL pueden retornar uno de los tipos básicos definidos en el estándar SQL o por el usuario, el valor nulo o un conjunto de valores (utilizando SETOF o RETURNS TABLE).

Los disparadores son un tipo especial de función que permite ejecutar de manera automática operaciones cuando ocurra un determinado evento sobre alguna tabla de la base de datos; posibilidad ampliamente empleada para la replicación de los datos, la realización de auditorías, entre otras.

Los cursores son un mecanismo que permite iterar por el resultado de una consulta para realizar operaciones sobre él; útiles, fundamentalmente, para el trabajo con grandes volúmenes de datos.

3.12 Para hacer con PL/pgSQL

1. Implementar funciones que permitan la inserción de datos en cada una de las tablas siguientes, teniendo en cuenta que se pasarán por parámetro todos sus datos, que se debe chequear que no exista el identificador de la tabla (en cuyo caso se mostrará una notificación y retornará falso) y que retornará verdadero en caso de que la inserción sea efectiva:
   a. *categories*
   b. *products*
   c. *customers*
   d. *orderlines*
   e. *order*

2. Desarrollar una función que dado el identificador de un cliente devuelva nombre, apellidos, dirección, ciudad y país; de no existir el cliente debe mostrar una notificación especificándolo.

3. Elaborar una función que dada una fecha devuelva todos los datos de órdenes que se realizaron antes de ella, de no haber, mostrar un mensaje especificándolo.

4. Obtener una función que devuelva el nombre de los productos que en el inventario ya no queden en almacén (*inventory.quan_in_stock* - *inventory.sales* = 0), si no hay productos en esta situación emitir un mensaje indicándolo.

5. Crear una función que dada una categoría de un producto calcule el promedio del precio si la categoría es "Games" o "Music", de lo contrario calcular el valor mínimo del precio.

6. Implementar una función que permita actualizar dinámicamente una tabla pasada por parámetro, el atributo que se desea modificar, su nuevo valor y la condición que deben cumplir las filas a actualizar.

7. Elaborar una función que permita mostrar todos los datos de una tabla pasada por parámetro dinámicamente.

8. Obtener una función que, dado el identificador de una categoría, sume un monto pasado por parámetro a los precios existentes de los productos pertenecientes a dicha categoría. Debe, además, retornar todos los datos actualizados de los productos de dicha categoría.

9. Haciendo uso de disparadores, implementar un mecanismo para:
    a. Impedir la eliminación de categorías, cuando se intente mostrar un mensaje de notificación especificando que no pueden ser eliminadas.
    b. Guardar en los registros de PostgreSQL los nuevos productos que se inserten con precio superior a las 200 unidades.
        i. Registrar no solamente los nuevos, sino todos aquellos que al actualizarse tengan un precio superior a las 200 unidades.
    c. Registrar en una tabla llamada eliminados, que guarda el nombre de los productos, aquellos eliminados.
    d. Registrar los cambios sobre los precios de los productos; para ello en caso de alguna modificación sobre los precios, se debe registrar en una tabla *act_precios* el identificador del producto, el precio anterior, el precio actual y la fecha en que fue realizada la modificación.
        i. Implementar dicho disparador con uno de tipo condicional.
        ii. ¿Con qué otro tipo de disparador se garantiza que sólo se invoque la función cuando se actualice el precio? Implementarlo.
    e. Chequear cuando se inserte una nueva orden que, para aquellos clientes con más de 3 órdenes se les rebaje el impuesto al 5% del monto neto, y de tener más de 5 órdenes se les rebaje el impuesto al 6% del monto neto.
    f. Implementar un solo disparador que englobe todas las condiciones de los incisos anteriores relacionados con la tabla *products*:
        - Registrar en los logs de PostgreSQL los productos con precio superior a las 200 unidades.
        - Registrar en la tabla eliminados aquellos productos que se borren.
        - Registrar en la tabla *act_precios* el identificador del producto, el precio anterior, el precio actual y la fecha en que fue realizada la modificación.

10. Dada una vista que muestre la información de la consulta siguiente:

```
SELECT p.title, c.categoryname
FROM categories c JOIN products p USING (category);
```

    a. Realizar un mecanismo en que, si se modifica el título de un producto o el nombre de la categoría de dicha vista, el mismo se propague a la tabla correspondiente.

11. Haciendo uso de cursores, implementar una función para:
    a. Sumar un monto pasado por parámetro a los precios existentes de los productos pertenecientes a una categoría, dado el identificador de dicha categoría. Debe, además, retornar todos los datos actualizados de los productos de dicha categoría.
    b. Eliminar de las 36 órdenes realizadas el 24-06-2004 de la 12 a la 25, que fueron generadas por error. Mostrar, además, los datos de las órdenes resultantes ese día.
        i. Recibir por parámetro, teniendo en cuenta que el problema de las órdenes generadas por error es bastante común, el día y el rango de órdenes a eliminar para que las elimine y muestre los datos de las resultantes.
        ii. Hacer uso de la cláusula CURRENT OF para la implementación de la función.
    c. Rebajar el impuesto en un 1% (*totalamount**0.01) a aquellos clientes con más de 10 órdenes. Mostrar, además, todos los datos de dichos clientes.
    d. Rebajar, a aquellos clientes con edad superior a una pasada por parámetro, el impuesto (*tax*) en una cantidad también pasada por parámetro. Mostrar, además, identificador, nombre, apellidos, edad y país de dichos clientes, así como el promedio de sus impuestos.
12. Haciendo uso de cursores, implementar una función y usarla en una transacción para:
    a. Retornar los productos ordenados por su identificador de menor a mayor, de cinco mil en cinco mil.
    b. Retornar nombre y apellido de todos los clientes, ordenados alfabéticamente por el nombre.
    c. Rebajar, a aquellos productos (products) que tengan más de un número de pedidos (orderlines) asociados pasado por parámetro, el precio (price) en un monto también pasado por parámetro. Mostrar, además, los datos actualizados de dichos productos.

Capítulo 4
Programación de funciones en lenguajes procedurales de desconfianza de PostgreSQL

Metas:

- Conocer las ventajas y desventajas del uso de lenguajes procedurales de desconfianza
- Dominar la sintaxis, el empleo de parámetros, la homologación de tipos a PostgreSQL, el retorno de valores y la implementación de disparadores en PL/Python
- Dominar la sintaxis, el empleo de parámetros, la homologación de tipos a PostgreSQL, el retorno de valores y la implementación de disparadores en PL/R

PROGRAMACIÓN DE FUNCIONES EN LENGUAJES PROCEDURALES DE DESCONFIANZA DE POSTGRESQL

4.1 Introducción a los lenguajes de desconfianza

En capítulos previos se ha mostrado cómo programar funciones en SQL y PL/pgSQL, potentes lenguajes que permiten realizar todas las operaciones con los datos almacenados en las bases de datos, y que se recomienda utilizarlos siempre que lo que se necesite hacer pueda solventarse con ellos, sobre todo por ser lenguajes de confianza. En ocasiones estos lenguajes no son suficientes para ejecutar alguna actividad, como lo puede ser consumir algún recurso del sistema operativo o *hardware*, escribir directamente en los discos del servidor, entre otros. PostgreSQL para estas actividades permite que se puedan desarrollar rutinas de código en otros lenguajes, como por ejemplo en C.

Nota

Para consultar sobre las rutinas en C remitirse a la sección *C-Language Function* en la Documentación Oficial

https://www.postgresql.org/docs/9.5/static/xfunc-c.html

En este capítulo se analizarán los lenguajes procedurales de desconfianza, útiles para realizar dichas actividades. El que su apellido sea "desconfianza", no quiere decir que no se deban utilizar, sino que se debe tener cuidado y tomar medidas para su uso. La primera medida la toma el gestor de bases de datos y es que solo pueden ser creados por administradores, al igual que las funciones en dichos lenguajes.

Existen varios lenguajes procedurales como son PL/Perl, PL/TCL, PL/PHP, PL/Java, PL/Python, PL/R, entre otros; los cuales permiten escribir código de funciones para PostgreSQL en sus respectivos lenguajes. Comúnmente se escribe una "u" como sufijo del lenguaje para su identificación del inglés *untrusted* (desconfianza), es el caso de PL/Perlu o PL/Pythonu. En este libro se particularizará en los lenguajes procedurales de desconfianza PL/Python y PL/R.

4.2 Lenguaje procedural PL/Python

PL/Python fue introducido en la versión 7.2 de PostgreSQL por Andrew Bosma en el año 2002 y ha sido mejorado paulatinamente en cada versión del gestor. El mismo posibilita escribir funciones en lenguaje Python desde PostgreSQL.

Python es un lenguaje simple y fácil de aprender, posee múltiples bibliotecas para hacer variadas actividades, ya sea en sistemas de gestión comercial, interacciones con el sistema operativo, etc.

A partir de la versión 9.1 de PostgreSQL, donde fue creado el mecanismo de extensiones, PL/Python se puede instalar como una extensión con la sentencia *create extension plpythonu*; en versiones previas puede instalarse desde la línea de comandos con la sentencia *createlang plpythonu nombre_basededatos*.

Nota Para el empleo correcto de PL/Python en Debian/Ubuntu, el sistema debe tener instalado el paquete *postgresql-plpython-9.5* o superior

Escritura de 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, con la particularidad de que el cuerpo de la función es código Python. El retorno de la función se realiza con la cláusula RETURN, clásica en Python para retornar valores; también puede ser empleado YIELD y, de no proveerse una salida se devuelve NONE, que se traduce a PostgreSQL como NULL. El siguiente es un ejemplo de una función implementada haciendo uso de este lenguaje procedural.

Ejemplo 62: Hola Mundo con PL/Python

```
CREATE FUNCTION holamundo() RETURNS text AS
$$
   return "Hola Mundo"
$$ LANGUAGE plpythonu;
```

Note, en el ejemplo anterior, que al igual que las funciones en SQL y PL/pgSQL, a esta se le debe especificar el lenguaje en que está siendo creada, en el caso de Python, *plpythonu*.

Parámetros de funciones PL/Python

Los parámetros de una función PL/Python se pasan como en cualquier otro lenguaje procedural y, una vez en la función, deben llamarse por su nombre, ver el ejemplo 63.

Ejemplo 63: Paso de parámetros en PL/Python

```
CREATE FUNCTION hola(nombre text) RETURNS text AS
$$
   return 'Hola %s' % nombre
$$ LANGUAGE plpythonu;
```

Al igual que en PL/pgSQL pueden definirse parámetros de salida, como se muestra en el ejemplo 64.

Ejemplo 64: Empleo de parámetros de salida en PL/Python

```
CREATE FUNCTION saludo(INOUT nombre text, OUT mayuscula text) AS
$$
   mayuscula = 'HOLA %s' % nombre.upper()
   return ('Hola ' + nombre, mayuscula)
$$ LANGUAGE plpythonu;

-- Invocación de la función
dell=# SELECT saludo('Carlos Lázaro');
                saludo
--------------------------------------
Hola Carlos Lázaro, HOLA CARLOS LÁZARO
(1 fila)
```

Utilizando tipos compuestos como parámetros

Los tipos de datos compuestos de PostgreSQL también pueden pasarse como parámetro en PL/Python y son, automáticamente, convertidos en diccionarios dentro de Python.

El ejemplo 65 muestra cómo se utilizan.

Ejemplo 65: Paso de parámetros como tipo de dato compuesto en PL/Python

```
CREATE FUNCTION parametros_mi_tipo(tabla mini_prod) RETURNS character varying
 AS
$$
   if 'A' in tabla['nombre'][0]:
      return 'Comienza con A'
   else:
      return 'No comienza con A'
$$ LANGUAGE plpythonu;

-- Invocación de la función
dell=# SELECT parametros_mi_tipo(ROW('Clandestinos', 1.10));
 parametros_mi_tipo
--------------------
 No comienza con A
(1 fila)
```

Note que la función del ejemplo 65 chequea que un producto comience o no con letra A, recibiendo un parámetro de tipo *mini_prod*, que fue el tipo de dato creado en el "Ejemplo 36" en la página 46, compuesto por el nombre y precio del producto.

Utilizando arreglos como parámetros

También se pueden pasar como parámetros arreglos de PostgreSQL, los que son convertidos a listas o tuplas. El ejemplo 66 muestra cómo emplearlos, implementando una función que devuelve el primer elemento de un arreglo pasado por parámetro.

Ejemplo 66: Arreglos pasados por parámetros en PL/Python

```
CREATE FUNCTION arreglos(a character varying[]) RETURNS character varying AS
$$
   return a[0]
$$ LANGUAGE plpythonu;

-- Invocación de la función
dell=# SELECT arreglos(array['Álex','Anthony']);
 arreglos
----------
 Álex
(1 fila)
```

Homologación de tipos de datos PL/Python

Según la Documentación Oficial de PostgreSQL, la homologación de tipos de datos de PostgreSQL a PL/Python se realiza como muestra la tabla siguiente.

Tabla 3: Homologación de tipos de datos PL/Python

PostgreSQL	Python 2	Python 3
text, char, varchar	str	str
boolean	bool	bool
real, numeric, double	float	float
smallint, integer	int	int
bigint, oid	long	int
null	none	none

Retorno de valores en funciones PL/Python

En epígrafes anteriores se ha mostrado cómo pasar parámetros y cómo se utiliza la cláusula RETURN en Python para retornar valores; en esta sección se analizarán algunas particularidades del retorno de valores en PL/Python.

Retornando arreglos

El retorno de una lista o tupla en Python es similar a un arreglo en PostgreSQL;

el ejemplo 67 lo muestra.

Ejemplo 67: Retornando una tupla en PL/Python

```
CREATE FUNCTION retorna_arreglo_texto() RETURNS text[] AS
$$
   return ('Hola','Mundo')
$$ LANGUAGE plpythonu;

-- Invocación de la función
dell=# SELECT retorna_arreglo_texto();
 retorna_arreglo_texto
-----------------------
 {Hola,Mundo}
(1 fila)
```

Retornando tipos compuestos

El retorno de tipos compuestos en Python puede realizarse de tres modos; como:

- Tupla o lista: que debe tener la misma cantidad y tipos de datos que el compuesto definido por el usuario; el ejemplo 68 muestra el caso.
- Diccionario: que debe tener la misma cantidad y tipos de datos que el tipo compuesto definido, además, los nombres de las columnas deben coincidir; el ejemplo 69 muestra el caso.
- Objeto: donde la definición de la clase debe tener los atributos similares al del tipo de dato compuesto por el usuario.

Ejemplo 68: Devolviendo valores de tipo compuesto como una tupla en PL/Python

```
CREATE FUNCTION retorna_tipo_lista() RETURNS mini_prod AS
$$
   nombre = 'Una novia para David'
   precio = 2.80
   return (nombre, precio)
$$ LANGUAGE plpythonu;

-- Invocación de la función
dell=# SELECT retorna_tipo_lista();
      retorna_tipo_lista
------------------------------
 ("Una novia para David", 2.8)
(1 fila)
```

En este y los siguientes ejemplos se emplea *mini_prod*, creado en el "Ejemplo 36" en la página 46.

Ejemplo 69: Devolviendo valores de tipo compuesto como un diccionario en PL/Python

```
CREATE FUNCTION retorna_tipo_dic() RETURNS mini_prod AS
$$
   nombre = 'Una novia para David'
   precio = 2.80
   return { "nombre": nombre, "precio": precio }
$$ LANGUAGE plpythonu;

-- Invocación de la función
dell=# SELECT retorna_tipo_dic();
       retorna_tipo_dic
------------------------------
 ("Una novia para David", 2.8)
(1 fila)
```

Retornando conjuntos de resultados

Para devolver conjuntos de datos puede utilizarse una lista o tupla (ver el ejemplo 70), un generador (YIELD, ver el ejemplo 71) o un iterador.

Ejemplo 70: Devolviendo conjunto de valores como una lista o tupla en PL/Python

```
CREATE FUNCTION mini_prod_conjunto() RETURNS SETOF mini_prod AS
$$
   nombre = 'Hello Hemingway'
   precio = 4.31
   return ([nombre, precio], [nombre + nombre, precio + precio])
$$ LANGUAGE plpythonu;

-- Invocación de la función
dell=# SELECT * FROM mini_prod_conjunto();
nombre | precio
-------------------------------------
Hello Hemingway | 4.31
Hello HemingwayHello Hemingway | 8.62
(2 filas)
```

Ejemplo 71: Devolviendo conjunto de valores como un generador en PL/Python

```
CREATE FUNCTION mini_prod_conj_generador() RETURNS SETOF record AS
$$
   lista = [('La pared de las palabras', 5.00), ('Memorias del subdesarrollo',
    5.10), ('Suite Habana', 3.90)]
   for producto in lista:
      yield (producto[0], producto[1])
$$ LANGUAGE plpythonu;
```

```
-- Invocación de la función
dell=# SELECT * FROM mini_prod_conj_generador() AS (titulo text, precio
       numeric);
titulo | precio
--------------------------------
La pared de las palabras | 5.00
Memorias del subdesarrollo | 5.10
Suite Habana | 3.90
(3 filas)
```

Ejecución de consultas SQL en funciones PL/Python

PL/Python permite realizar consultas SQL sobre la base de datos a través de un módulo llamado *plpy*, importado por defecto; que tiene varias funciones como:

- *plpy.execute*(*consulta* [, *max-rows*]): permite ejecutar una consulta con una cantidad finita de tuplas en el resultado, especificada en el segundo parámetro (opcional). Devuelve un objeto similar a una lista de diccionarios, al que se puede acceder de la forma *resultado*[0]['*micolumna*'], para acceder al primer registro y a la columna '*micolumna*'. Ver el ejemplo 72 para su uso.
- *plpy.prepare*(*consulta* [, *argtypes*]): permite preparar planes de ejecución para determinadas consultas que luego serán ejecutadas por la función *execute*(*plan* [, *arguments* [, *max-rows*]]). Ver el ejemplo 73 para su uso.
- *plpy.cursor*(*consulta*): añadido a partir de la versión 9.2 del gestor.

Note que la consulta ejecutada con *plpy.execute* pudiera prescindir del segundo parámetro de haberse escrito como SELECT * FROM products LIMIT 10.

Ejemplo 72: Devolviendo valores de la ejecución de una consulta desde PL/Python con plpy.execute

```
CREATE FUNCTION obtener_productos() RETURNS TABLE (id int, titulo text, precio
 numeric) AS
$$
   resultado = plpy.execute("SELECT * FROM products", 10)
   for tupla in resultado:
      yield (tupla['prod_id'], tupla['title'], tupla['price'])
$$ LANGUAGE plpythonu;

-- Invocación de la función
dell=# 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)
```

En el ejemplo siguiente constate que el parámetro con que es comparado, *prod_id* en la consulta preparada, es suministrado cuando se ejecuta con *plpy.execute,* en este caso, se preparó una consulta que retornara todos los datos del producto con identificador 4.

Ejemplo 73: Ejecución con plpy.execute de una consulta preparada con plpy.prepare

```
CREATE FUNCTION preparada() RETURNS text AS
$$
   miplan = plpy.prepare("SELECT * FROM products WHERE prod_id = $1", ["int"])
   resultado = plpy.execute(miplan, [4])
   return resultado[0]['title']
$$ LANGUAGE plpythonu;

-- Invocación de la función
dell=# SELECT * FROM preparada();
   preparada
---------------
 ACADEMY AFFAIR
(1 fila)
```

Nota

PL/Python cuenta con otras funciones que pueden ser útiles, como *plpy.debug(msg)*, *plpy.log(msg)*, *plpy.info(msg)*, *plpy.notice(msg)*, *plpy.warning(msg)* y *plpy.error(msg)*; para más información sobre estas funciones ver la sección *Utility Functions* del capítulo *PL/Python - Python Procedural Language*, en la Documentación Oficial

http://www.postgresql.org/docs/9.5/static/plpython-util.html

Uso de PL/Python para generar un XML

Como se ha mostrado anteriormente, los lenguajes de desconfianza son utilizados para hacer rutinas externas al servidor de bases de datos, a continuación se muestra un ejemplo de cómo guardar en un archivo XML el resultado de una consulta.

Note que para que esta función se ejecute satisfactoriamente la ruta especificada en fichero debe existir.

Ejemplo 74: Guardar en un XML el resultado de una consulta con PL/Python

```
CREATE FUNCTION salva_tabla_xml() RETURNS character varying AS $$
   from xml.etree import ElementTree
   rv = plpy.execute("SELECT * FROM categories")
   raiz = ElementTree.Element('consulta')
   for valor in rv:
      datos = ElementTree.SubElement(raiz,'datos')
      for key, value in valor.items():
         element = ElementTree.SubElement(datos, key)
         element.text = str(value)
         contenido = ElementTree.tostring(raiz)
         fichero = open('/tmp/archivo.xml','w')
         fichero.write(contenido)
         fichero.close()
   return contenido
$$ LANGUAGE plpythonu;
```

Realización de disparadores con PL/Python

PL/Python permite escribir funciones disparadoras, para lo que cuenta con un variable diccionario TD que posee varios pares llave/valor útiles para su trabajo; algunos de los que 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 disparador (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 observa el disparador.

Se puede retornar NONE u OK para dar a conocer que la operación con la fila se ejecutó correctamente o SKIP para abortar el evento.

El ejemplo 75 muestra un disparador que verifica, al insertar un nuevo producto, si el título es "HOLA MUNDO", de ser así no permite su registro. Note el empleo de *plpy.notice* para realizar notificaciones.

Ejemplo 75: Disparador en PL/Python

```
-- Función disparadora
CREATE FUNCTION tgr_plpython() RETURNS trigger AS
$$
   plpy.notice('Comenzando el disparador')
   if TD["event"] == "INSERT" and TD["new"]['title'] == "HOLA MUNDO":
      plpy.notice('Se abortó la inserción por tener el título: ' + TD["new"]
       ['title'])
      return "SKIP"
   else:
      plpy.notice('Se registró correctamente')
      return "OK"
$$ LANGUAGE plpythonu;

-- Disparador
CREATE TRIGGER tgr_plpython
BEFORE INSERT or UPDATE
ON products
FOR EACH ROW
EXECUTE PROCEDURE tgr_plpython();

-- Invocación de la función
dell=# INSERT INTO products(prod_id, category, title, actor, price, special,
       common_prod_id) VALUES (20001, 13, 'HOLA MUNDO', 'Laura de la Uz',
       12.0, 0, 1000);
NOTICE: Comenzando el disparador
CONTEXT: PL/Python function "tgr_plpython"
NOTICE: Se abortó la inserción por tener el título: HOLA MUNDO
CONTEXT: PL/Python function "tgr_plpython"
```

4.3 Lenguaje procedural PL/R

PL/R es un lenguaje procedural para PostgreSQL que permite escribir funciones en el lenguaje R para su empleo dentro del gestor; desarrollado por Joseph E. Conway desde el 2003 y compatible con PostgreSQL a partir de su versión 7.4, soporta casi todas las funcionalidades de R desde el gestor. Este lenguaje es orientado específicamente a realizar operaciones estadísticas, siendo muy potente en esta rama, cuenta con disímiles paquetes (conjunto de funcionalidades) y se utiliza en varias áreas de la informática, la medicina, la bioinformática, entre otras.

A partir de la versión 9.1 de PostgreSQL, donde fue creado el mecanismo de extensiones, PL/R se puede instalar como una extensión con la sentencia *create extension plr;* en versiones previas del gestor puede instalarse desde la línea de comandos con la sentencia *createlang plr nombre_basededatos*.

Nota Se debe tener instalado en sistemas Debian/Ubuntu, el paquete *postgresql-9.5-plr* o superior para el trabajo con PL/R

Escritura de 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 en este lenguaje las funciones deben nombrarse de forma diferente, aunque sus parámetros no sean los mismos.

El retorno de la función se realiza con la cláusula clásica de R para retornar valores de una función, RETURN, pero en ocasiones no es necesario explicitarla, ver el ejemplo 76.

Ejemplo 76: Función en PL/R que suma 2 números pasados por parámetros

```
CREATE FUNCTION suma(a integer, b integer) RETURNS integer AS
$$
   return (a + b)
$$ LANGUAGE plr;
```

Parámetros de funciones PL/R

Los parámetros de una función PL/R 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 arg*N,* siendo *N* el orden que ocupan en la lista de parámetros, ver el ejemplo 77.

Ejemplo 77: Función en PL/R que suma 2 números pasados por parámetros sin nombrarlos

```
CREATE OR REPLACE FUNCTION suma(integer, integer) RETURNS integer AS
$$
   return (arg1 + arg2)
$$ LANGUAGE plr;
```

Utilizando arreglos como parámetros

Cuando una función PL/R recibe un arreglo como parámetro automáticamente lo convierte a un vector c(...).

La función implementada en el ejemplo 78 calcula la desviación estándar de un arreglo pasado por parámetro.

Ejemplo 78: Cálculo de la desviación estándar desde PL/R

```
CREATE FUNCTION desv_estandar(arreglo int[]) RETURNS real AS
$$
   desv <- sd(arreglo)
   return (desv)
$$ LANGUAGE plr;

-- Invocación de la función
dell=# SELECT desv_estandar(array[4,2,3,4,5,6,3]);
desv_estandar
-------------
     1.34519
(1 fila)
```

Utilizando tipos de datos compuestos como parámetros

PL/R permite que se puedan pasar tipos de datos compuestos definidos por el usuario, los cuales son convertidos en R a *data.frames* de una fila. El ejemplo 79 muestra cómo hacerlo; note que la forma de acceder a un elemento del tipo de dato compuesto es mediante *variable*$*elemento*.

Ejemplo 79: Pasando un tipo de dato compuesto como parámetro en PL/R

```
-- Función que recibe un tipo de dato compuesto por parámetro
CREATE FUNCTION compuesto(a mini_prod) RETURNS text AS
$$
   if (a$precio == 0)
      {return (print("Precio incorrecto"))}
   return (print("Precio correcto"))}
$$ LANGUAGE plr;

-- Invocación de la función
dell=# SELECT compuesto((ROW('Vestido de Novia', 0)));
     compuesto
------------------
 Precio incorrecto
(1 fila)
```

Homologación de tipos de datos PL/R

Según la Documentación Oficial de PL/R, la homologación de tipos de datos de PostgreSQL a PL/R se realiza como muestra la tabla 4.

Tabla 4: Homologación de tipos de datos PL/R

PostgreSQL	R
boolean	logical
int8, flat4, flat8, cash, numeric	numeric
int, int4	integer
bytea	object
cualquier otro	character

Retorno de valores de funciones PL/R

En PL/R los resultados de una función se retornan haciendo uso de la palabra reservada RETURN y entre paréntesis se especifica el valor a retornar, del que su tipo de dato debe coincidir con el definido en la declaración de la función.

Retornando arreglos

Para retornar arreglos se deben devolver, desde PL/R, vectores o arreglos de una dimensión, por ejemplo, la función del ejemplo 80 retorna el resumen estadístico de un vector como un arreglo de PostgreSQL.

Ejemplo 80: Retorno de valores con arreglos desde PL/R

```
-- Función que realiza un resumen estadístico de un arreglo
CREATE FUNCTION resumen_estadistico(a integer[]) RETURNS real[] AS
$$
   resumen <- summary(a)
   return (resumen)
$$ LANGUAGE plr;

-- Invocación de la función
dell=# SELECT resumen_estadistico(array[2,5,3,2,2,7,8,0]);
 resumen_estadistico
----------------------
{0,2,2.5,3.625,5.5,8}
(1 fila)
```

Retornando tipos compuestos

Para devolver tipos compuestos se debe utilizar, desde PL/R, un *data.frame* con los respectivos atributos del tipo de dato compuesto definido por el usuario, ordenados

respecto a los tipos de datos. El ejemplo 81 muestra su uso.

Ejemplo 81: Retorno de valores con tipos de datos compuestos desde PL/R

```
CREATE FUNCTION devolvercompuesto() RETURNS SETOF mini_prod AS
$$
   return (data.frame(nombre = "Los pájaros tirándole a la escopeta", precio =
    2.50))
$$ LANGUAGE plr;

-- Invocación de la función
dell=# SELECT devolvercompuesto();
nombre | precio
----------------------------------------
Los pájaros tirándole a la escopeta | 2.5
(1 fila)
```

Note que en el ejemplo anterior se está utilizando el tipo de dato *mini_prod*, creado por el usuario, que tiene la estructura (nombre, precio) y que, por tanto, el *data.frame* debe utilizar los atributos en ese mismo orden.

Retornando conjuntos

En PL/R se pueden retornar conjuntos de datos de algún tipo compuesto, TABLE o RECORD. En el ejemplo siguiente se muestran dos funciones que hacen uso de ambas opciones.

Ejemplo 82: Retorno de conjuntos desde PL/R

```
-- Devolviendo conjuntos haciendo uso de TABLE
CREATE FUNCTION devolver_varios_table() RETURNS TABLE(titulo text, precio
 numeric) AS
$$
   nombres <- c("Fresa y Chocolate", "La película de Ana", "Conducta")
   precios <- c(25.01, 10.65, 60)
   dataframe <- data.frame(nombre = nombres, precio = precios)
   return (as.matrix(dataframe))
$$ LANGUAGE plr;

-- Invocación de la función
dell=# SELECT * FROM devolver_varios_table();
titulo | precio
---------------------------
Fresa y Chocolate | 25.01
La película de Ana | 10.65
Conducta | 60.00
(3 filas)
```

```
-- Devolviendo conjuntos haciendo uso de RECORD
CREATE FUNCTION devolver_varios_record() RETURNS SETOF record AS
$$
   nombres <- c("Fresa y Chocolate", "La película de Ana", "Conducta")
   precios <- c(25.01, 10.65, 60)
   dataframe <- data.frame(nombre = nombres, precio = precios)
   return (dataframe)
$$ LANGUAGE plr;

-- Invocación de la función
dell=# SELECT * FROM devolver_varios_record() AS (titulo text, precio
       numeric);
titulo | precio
---------------------------
Fresa y Chocolate | 25.01
La película de Ana | 10.65
Conducta | 60.00
(3 filas)
```

Ejecución de consultas en funciones PL/R

Para ejecutar consultas desde PL/R 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. Ver el ejemplo 83.
- *pg.spi.prepare*: permite preparar consultas y salvar el plan generado para una ejecución posterior de las mismas; el plan solo se salvará durante la conexión o transacción en curso.
- *pg.spi.execp*: ejecuta una consulta previamente preparada con *pg.spi.prepare* y permite el paso de los parámetros para la consulta preparada.
- *pg.spi.cursor_open* y *pg.spi.cursor_fetch*: empleados para el trabajo con cursores.

Ejemplo 83: Retorno del resultado de una consulta desde PL/R usando pg.spi.exec

```
CREATE FUNCTION devolver_consulta() RETURNS SETOF mini_prod AS
$$
   resultado <- pg.spi.exec("SELECT title, price FROM products WHERE prod_id <
    100")
   return (resultado)
$$ LANGUAGE plr;

-- Invocación de la función
dell=# SELECT * FROM devolver_consulta();
```

```
nombre | precio
--------------------------
ACADEMY ACADEMY | 25.99
ACADEMY ACE | 20.99
ACADEMY ADAPTATION | 28.99
-- More --
```

Uso de PL/R para generar una gráfica de barras

Como se ha mostrado, estos lenguajes de desconfianza son utilizados para hacer rutinas externas al servidor de bases de datos; el ejemplo 84 muestra cómo generar el gráfico de barras, de la figura 3, en un archivo *.png* con el resultado de una consulta.

Ejemplo 84: Función que genera una gráfica de barras con el resultado de una consulta en PL/R

```
CREATE FUNCTION barras_simples(nombre text, consulta text, texto text, ejex
 text[]) RETURNS integer AS
$$
   png(paste(nombre,"png", sep = "."))
   resultado <- pg.spi.exec(consulta)
   barplot(as.matrix(resultado), beside=TRUE, main=texto, col=rainbow(length(
    as.matrix(resultado))), names.arg=c(ejex))
   dev.off()
$$ LANGUAGE plr;

-- Invocación de la función
dell=# SELECT * FROM barras_simples('grafica_barras', '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', 'Cantidad
       producciones x Categoría', array(SELECT categoryname FROM categories
       ORDER BY categoryname LIMIT 4)::text[]);
```

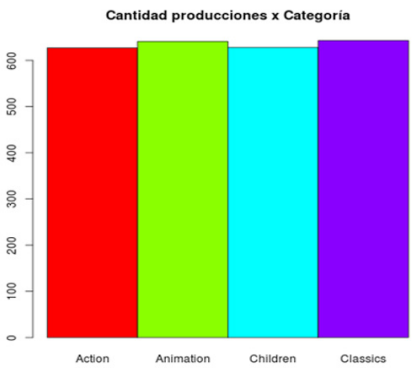


Figura 5: Gráfica de barras generada del resultado de una consulta en PL/R

Realización de disparadores con PL/R

PL/R permite escribir funciones disparadoras, para lo que cuenta con un variable diccionario TD que posee varios pares llave/valor útiles; algunos de los que se describen a continuación:

- *pg.tg.relname*: devuelve el nombre de la tabla que invocó al disparador.
- *pg.tg.when*: devuelve una cadena, en mayúsculas, especificando cuándo se ejecutó el disparador (BEFORE o AFTER).
- *pg.tg.op*: devuelve una cadena, en mayúsculas, del evento que ejecutó el disparador (INSERT, UPDATE o DELETE).
- *pg.tg.new*: *data.frame* que contiene los valores nuevos que tiene la fila nueva insertada o modificada, puede llamarse utilizando el nombre de la columna de la tabla, por ejemplo *pg.tg.new$columna*.
- *pg.tg.old*: *data.frame* que contiene los valores viejos que tiene la fila actualizada o eliminada, puede llamarse utilizando el nombre de la columna de la tabla, por ejemplo *pg.tg.old$columna*.

Nota Existen otras variables como *pg.tg.level*, *pg.tg.relid*, *pg.tg.args* y *pg.tg.name* que pueden analizarse en *PL/R Users Guide - R Procedural Language*

El retorno del disparador puede ser NULL (el resultado de la operación realizada será ignorado), una fila en forma de *data.frame*(*pg.tg.new*, *pg.tg.old*) o algún valor que contenga las mismas columnas de la tabla que lo invocó. El ejemplo 85 muestra el empleo de disparadores desde PL/R.

Ejemplo 85: Utilizando disparadores en PL/R

```
-- Función disparadora
CREATE FUNCTION tgrplrfunc() RETURNS trigger AS
$$
   pg.thrownotice("Comenzado el disparador")
   if (pg.tg.op == "INSERT" & pg.tg.new$price == 0)
   {
      pg.thrownotice("Registro no insertado")
      return (NULL)
   }
   return (pg.tg.new)
$$ LANGUAGE plr;

-- Disparador
CREATE TRIGGER testplr_trigger
BEFORE INSERT OR UPDATE
```

```
ON products
FOR EACH ROW
EXECUTE PROCEDURE tgrplrfunc();

-- Invocación de la función
dell=# INSERT INTO products (prod_id, category, title, actor, price, special,
       common_prod_id) VALUES (20003, 13, 'Un Rey en La Habana',
       'Alexis Valdés', 30, 0, 1000);
NOTICE: Comenzado el disparador
NOTICE: Registro no insertado
Consulta retornada exitosamente: 0 filas afectadas, tiempo de ejecución 33 ms
```

4.4 Resumen

Los llamados lenguajes de desconfianza permiten escribir lógica de negocio, en el servidor de bases de datos PostgreSQL, en sus propios lenguajes. En el 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 y; su cuerpo estará compuesto por las características del lenguaje utilizado.

Además, se analizó la homologación de los tipos de datos de ambos lenguajes con los de PostgreSQL, así como la implementación de disparadores en ellos.

Se debe tener cuidado en el uso de estos lenguajes pues desde ellos se puede acceder a recursos del servidor, más allá de los datos almacenados. Se recomienda utilizarlos en entornos controlados y para actividades que no se puedan realizar desde los lenguajes nativos SQL y PL/pgSQL, por la seguridad y porque el rendimiento de los mismos no suele ser el óptimo.

4.5 Para hacer con PL/Python y PL/R

1. Desarrollar una función en PL/Python que devuelva el monto total (*price*quantity*) de las órdenes donde el actor es "VIVIEN COOPER".

2. Construir una función en PL/Python que permita devolver los 100 productos más caros.

3. Elaborar una función en PL/Python que permita exportar los datos del ejercicio 2 a un archivo CSV.

4. Realizar una función en PL/Python para devolver los productos en que su título termine con un caracter pasado por parámetro, prepare dicha consulta antes de ejecutarla.

5. Concebir una función en PL/Python que permita exportar los datos del ejercicio 4 a un archivo Excel.

6. Implementar un mecanismo basado en disparadores en PL/Python que permita 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 notificando la situación (utilice el correo admin@dellstore.com).

7. Desarrollar una función en PL/R que devuelva los productos en que su precio sea mayor a uno pasado por parámetro.

8. Confeccionar una función en PL/R que permita obtener un gráfico de pastel con el resultado de la consulta que muestre la cantidad de clientes por países.

9. Realizar una función en PL/R que permita calcular la mediana de todos los productos especiales.

Resumen

En PL/pgSQL y otros lenguajes procedurales en PostgreSQL se destacan las ventajas de la programación lado del servidor de bases de datos, enfatizando en cómo brinda esta característica el sistema de gestión de bases de datos PostgreSQL, que la implementa a través del desarrollo de funciones.

El libro se enfoca en dos de las formas principales de realizarla, mediante el desarrollo de funciones en SQL y en lenguajes procedurales. En los capítulos propuestos se analizó la forma de implementar este tipo de funciones y su uso en algunos escenarios, abarcando con variados ejemplos las sintaxis, estructuras y trabajo con ellas en los lenguajes SQL, PL/pgSQL, PL/Python y PL/R.

De desarrollarse los ejercicios propuestos en los capítulos se debe haber alcanzado una habilidad básica para la programación del lado del servidor PostgreSQL con características disponibles hasta su versión 9.5.

Textos consultados

Conway, Joseph E. 2009. PL/R User's Guide - R Procedural Language. Boston : s.n., 2009.

Date, C. J. y Darwen, Hugh. 1996. A Guide to the SQL Standard . California : Addison-Wesley Professional, 1996. ISBN: 978-0201964264.

Krosing, Hannu, Mlodgenski, Jim y Roybal, Kirk. 2013. PostgreSQL Server Programming. Birmingham : Packt Publishing, 2013. ISBN 978-1-84951-698-3.

Matthew, Neil y Stones, Richard. 2005. Beginning databases with PostgreSQL, from novice to professional. 2nd edition. 2005.

Melton, Jim y Simon, Alan R. 1993. Understanding the New SQL. s.l. : Morgan Kaufmann Publishers, 1993. ISBN: 9781558602458.

Momjian, Bruce. 2001. PostgreSQL Introduction and Concepts. 2001.

Smith, Gregory. 2010. PostgreSQL 9.0 High Performance. Birmingham-Mumbai : Packt Publishing, 2010. ISBN 978-1-849510-30-1.

The PostgreSQL Global Development Group. 2016. PostgreSQL 9.5.0 Documentation. California : s.n., 2016.

Posibles soluciones a ejercicios propuestos

Ejercicios del Capítulo 1

1_1.

- Función en SQL: ejecuta puramente operaciones SQL.
- Función en lenguaje procedural: ejecuta operaciones empleando lenguajes de tipo procedural como PL/pgSQL, PL/Python, etc.
- Función interna: ejecuta operaciones en lenguaje C que están enlazadas directamente al servidor PostgreSQL lo que implica que están predefinidas dentro del gestor.
- Función en lenguaje C: ejecuta operaciones, escritas en el lenguaje C o compatible, y que pueden ser cargadas a PostgreSQL dinámicamente bajo demanda.

1_2.

- Aumento del rendimiento: se impide que los datos viajen de la base de datos a la aplicación, evitando la latencia por esta operación.
- Fácil mantenimiento: si la lógica de negocio se modifica los cambios se realizan en la base de datos central y pueden ser hechos fácilmente mediante el Lenguaje de Definición de Datos para actualizar las funciones implicadas en ellos.
- Simplicidad para garantizar la seguridad en la lógica de negocio: a las funciones

se les pueden definir privilegios de acceso a través del Lenguaje de Control de Datos (DCL) y se evita que los datos viajen por la red.

- Evita reescritura de código de negocio: en varias aplicaciones que consumen de una única base de datos.

1_3.

- Incluir varias operaciones en una única transacción, evitando el tráfico de la red.
- Realizar auditorías de datos, para controlar operaciones realizadas sobre los datos.
- Consumir características de otros lenguajes, como Python, R, etc. desde la base de datos.
- Exportar datos a formatos CSV, Excel, etc.

Ejercicios del Capítulo 2

2_1.a.

```
CREATE FUNCTION f2_1a(p_categoryname text) RETURNS categories AS
$$
   INSERT INTO categories(categoryname) VALUES ($1) RETURNING *;
$$ LANGUAGE sql;

-- Insertar nueva categoría
SELECT * FROM f2_1a('3D');
```

2_1.b.

```
CREATE FUNCTION f2_1b(p_category integer, p_title text, p_actor text, p_price
 numeric, p_special integer, p_common_prod_id integer) RETURNS products AS
$$
   INSERT INTO products(category, title, actor, price, special,
    common_prod_id) VALUES ($1, $2, $3, $4, $5, $6) RETURNING *;
$$ LANGUAGE sql;

-- Insertar nuevo producto
SELECT * FROM f2_1b(7,'Plaf'::text,'Daysi Granados'::text,25.2,1,1);
```

2_1.c.

```
CREATE FUNCTION f2_1c(p_firstname text, p_lastname text, p_address1 text,
 p_address2 text, p_city text, p_state text, p_zip integer, p_country text,
 p_region smallint, p_email text, p_phone text, p_creditcardtype integer,
 p_creditcard text, p_creditcardexpiration text, p_username text, p_password
```

```
 text, p_age smallint, p_income integer, p_gender char) RETURNS customers AS
$$
   INSERT INTO customers(firstname, lastname, address1, address2, city, state,
    zip, country, region, email, phone, creditcardtype, creditcard,
    creditcardexpiration, username, password, age, income, gender) VALUES
    ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13, $14, $15, $16,
    $17, $18, $19) RETURNING *;
$$ LANGUAGE sql;

-- Insertar nuevo cliente
SELECT * FROM f2_1c('Álex'::text,'Vazquez'::text,'H no. 18'::text,
 'I no. 31'::text,'La Habana'::text,'-'::text, 1,'CU'::text,
 1::smallint,'avazquez@gmail.com'::text,'-'::text,1,'0535'::text,
 '-'::text,'avazquez'::text,'4lex'::text,24::smallint,1,'m'::text);
```

2_1.d.

```
CREATE FUNCTION f2_1d(p_orderlineid integer, p_orderid integer, p_prod_id
 integer, p_quantity smallint, p_orderdate date) RETURNS orderlines AS
$$
   INSERT INTO orderlines(orderlineid, orderid, prod_id, quantity, orderdate)
    VALUES ($1, $2, $3, $4, $5) RETURNING *;
$$ LANGUAGE sql;

-- Insertar nuevo pedido
SELECT * FROM f2_1d(13131,1,1,1::smallint,'2016-07-18'::date);
```

2_1.e.

```
CREATE FUNCTION f2_1e(p_orderdate date, p_customerid integer, p_netamount
 numeric, p_tax numeric, p_totalamount numeric) RETURNS orders AS
$$
   INSERT INTO orders(orderdate, customerid, netamount, tax, totalamount)
    VALUES ($1, $2, $3, $4, $5) RETURNING *;
$$ LANGUAGE sql;

-- Insertar nueva orden
SELECT * FROM f2_1e('2016-02-04',1,1.50,1.0,0.50);
```

2_2.

```
CREATE FUNCTION f2_2(p_prod_id integer) RETURNS products AS
$$
   SELECT * FROM products WHERE prod_id = $1;
$$ LANGUAGE sql;

-- Mostrar producto con identificador 1
SELECT * FROM f2_2(1);
```

2_3.a.

```
CREATE FUNCTION f2_3a(p_customerid integer, OUT firstname text, OUT lastname
 text, OUT age smallint, OUT gender text, OUT email text, OUT phone text,
 OUT city text, OUT country text) AS
$$
   SELECT firstname, lastname, age, gender, email, phone, city, country FROM
    customers WHERE customerid = $1;
$$ LANGUAGE sql;

-- Mostrar cliente con identificador 1 usando parámetros de salida
SELECT * FROM f2_3a(1);
```

2_3.b.

RECORD, de la forma:

```
CREATE FUNCTION f2_3b(p_customerid integer) RETURNS record AS
$$
   SELECT firstname, lastname, age, gender, email, phone, city, country FROM
    customers WHERE customerid = $1;
$$ LANGUAGE sql;

-- Mostrar cliente con identificador 1 usando RECORD
SELECT * FROM f2_3b(1) AS (nombre text, apellido text, edad smallint,
 genero text, email text, tlf text, ciudad text, pais text);
```

2_4.

```
CREATE FUNCTION f2_4(p_prod_id integer, p_price numeric) RETURNS products AS
$$
   UPDATE products SET price = $2 WHERE prod_id = $1 RETURNING *;
   /*Sin usar el RETURNING en el UPDATE hay que utilizar las sentencias
   UPDATE y SELECT*/
$$ LANGUAGE sql;

-- Actualizar precio de producto con identificador 1
SELECT * FROM f2_4(1,20.50);
```

2_5.

```
CREATE FUNCTION f2_5(p_prod_id integer) RETURNS record AS
$$
SELECT p.*, count(*) FROM products p JOIN orderlines o ON
 (p.prod_id = o.prod_id) WHERE p.prod_id = $1 GROUP BY p.prod_id;
$$ LANGUAGE sql;

-- Mostrar información del producto con identificador 1 y el total de pedidos
```

```
SELECT f2_5(1) AS (prod_id integer,category integer,title text,actor text,
 price numeric,special smallint,common_prod_id integer,total bigint);
```

2_6.

```
CREATE FUNCTION f2_6() RETURNS SETOF customers AS
$$
   SELECT * FROM customers WHERE age < 30 AND gender = 'F';
$$ LANGUAGE sql;

-- Mostar mujeres menores de 30 años
SELECT * FROM f2_6();
```

2_6.a

Pudiera definirse el tipo de retorno:

- De tipo RECORD, implica definir al invocarse la función los tipos de datos de cada atributo de la salida.
- Empleando las cláusulas RETURNS TABLE, implica definir luego de la cláusula y entre paréntesis nombre y tipo de dato de cada atributo de la salida.

2_7.

```
CREATE FUNCTION f2_7(p_customerid integer) RETURNS SETOF record AS
$$
   DELETE FROM orders WHERE customerid = $1 AND orderdate < '2004-04-24';
   SELECT orderid, orderdate, netamount, totalamount FROM orders WHERE
    customerid = $1;
$$ LANGUAGE sql;

-- Eliminar órdenes del cliente con identificador 41
SELECT f2_7(41) AS (orderid integer,orderdate date,netamount numeric,
 totalamount numeric);
```

2_8.

```
CREATE FUNCTION f2_8(p_category integer, p_title text, p_actor text, p_price
 numeric, p_special smallint, p_common_pid integer) RETURNS SETOF record AS
$$
   INSERT INTO products(category, title, actor, price, special,
    common_prod_id) VALUES ($1, $2, $3, $4, $5, $6);
   SELECT title, price, category FROM products;
$$ LANGUAGE sql;

-- Actualizar precio de producto con identificador 1
SELECT * FROM f2_8(7,'Esteban'::text,'Yuliet Cruz'::text,30.0,1,1) AS (title
```

```
 text,price numeric,category integer);
```

2_9.

```
CREATE FUNCTION f2_9() RETURNS SETOF bigint AS
$$
   SELECT count(*) FROM products GROUP BY category
$$ LANGUAGE sql;

-- Mostar total de productos por categoría
SELECT * FROM f2_9();
```

Ejercicios del Capítulo 3

3_1.a.

```
CREATE FUNCTION f3_1a(p_category integer, p_categoryname text) RETURNS
 boolean AS
$$
DECLARE
   id integer;
BEGIN
   SELECT category INTO id FROM categories WHERE category = $1;
   IF found THEN
      RAISE NOTICE 'Ya existe una categoría con el identificador %', $1;
      RETURN false;
   ELSE
      INSERT INTO categories(category, categoryname) VALUES ($1, $2);
      RETURN true;
   END IF;
END;
$$ LANGUAGE plpgsql;

-- Insertar nueva categoría
SELECT * FROM f3_1a(32,'3D');
```

3_1.b.

```
CREATE FUNCTION f3_1b(p_prod_id integer, p_category integer, p_title text,
 p_actor text, p_price numeric, p_special integer, p_common_prod_id integer)
 RETURNS boolean AS
$$
DECLARE
   id integer;
BEGIN
```

```
   SELECT prod_id INTO id FROM products WHERE prod_id = $1;
   IF found THEN
      RAISE NOTICE 'Identificador del producto existente';
      RETURN false;
   ELSE
      INSERT INTO products(prod_id, category, title, actor, price, special,
       common_prod_id) VALUES ($1, $2, $3, $4, $5, $6, $7);
      RETURN true;
   END IF;
END;
$$ LANGUAGE plpgsql;

-- Insertar nuevo producto
SELECT * FROM f3_1b(10500,7,'Amor vertical'::text,
 'Silvia Águila'::text,25.2,1,1);
```

3_1.c.

```
CREATE FUNCTION f3_1c(p_customerid integer, p_firstname text, p_lastname text,
 p_address1 text, p_address2 text, p_city text, p_state text, p_zip integer,
 p_country text, p_region smallint, p_email text, p_phone text,
 p_creditcardtype integer, p_creditcard text, p_creditcardexpiration text,
 p_username text, p_password text, p_age smallint, p_income integer, p_gender
 char) RETURNS boolean AS
$$
DECLARE
   id integer;
BEGIN
   SELECT customerid INTO id FROM customers WHERE customerid=$1;
   IF found THEN
      RAISE NOTICE 'Identificador del cliente existente';
      RETURN false;
   ELSE
      INSERT INTO customers(customerid, firstname, lastname, address1,
       address2, city, state, zip, country, region, email, phone,
       creditcardtype, creditcard, creditcardexpiration, username, password,
       age, income, gender) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10,
       $11, $12, $13, $14, $15, $16, $17, $18, $19, $20);
      RETURN true;
   END IF;
END;
$$ LANGUAGE plpgsql;

-- Insertar nuevo cliente
SELECT * FROM f3_1c(20500,'Carlos Lázaro'::text,'Vazquez'::text,
 'H no. 18'::text,'I no. 31'::text,'La Habana'::text,'-'::text,1,
 'CU'::text,1::smallint,'clvazquez@gmail.com'::text,'-'::text,1,
 '0535'::text,'-','clvazquez'::text,'c4rlos'::text,24::smallint,1,'m'::text);
```

3_1.d.

```
CREATE FUNCTION f3_1d(p_orderlineid integer, p_orderid integer,
 p_prod_id integer, p_quantity smallint, p_orderdate date) RETURNS boolean AS
$$
DECLARE
   id integer;
BEGIN
   SELECT orderlineid INTO id FROM orderlines WHERE orderlineid=$1;
IF found THEN
   RAISE NOTICE 'Identificador del pedido existente';
   RETURN false;
ELSE
   INSERT INTO orderlines(orderlineid, orderid, prod_id, quantity, orderdate)
    VALUES ($1, $2, $3, $4, $5);
   RETURN true;
END IF;
END;
$$ LANGUAGE plpgsql;

-- Insertar nuevo pedido
SELECT * FROM f3_1d(13132,1,1,1::smallint,'2016-02-04'::date);
```

3_1.e.

```
CREATE FUNCTION f3_1e(p_orderid integer, p_orderdate date, p_customerid
 integer, p_netamount numeric, p_tax numeric, p_totalamount numeric) RETURNS
 boolean AS
$$
DECLARE
   id integer;
BEGIN
   SELECT orderid INTO id FROM orders WHERE orderid = $1;
   IF found THEN
      RAISE NOTICE 'Identificador de la orden existente';
      RETURN false;
   ELSE
      INSERT INTO orders(orderid, orderdate, customerid, netamount, tax,
       totalamount) VALUES ($1, $2, $3, $4, $5, $6);
      RETURN true;
   END IF;
END;
$$ LANGUAGE plpgsql;

-- Insertar nueva orden
SELECT * FROM f3_1e(31312,'2016-02-04',1,1.50,1.0,0.50);
```

3_2.

```
CREATE FUNCTION f3_2(p_customerid integer) RETURNS record AS
$$
DECLARE
   id integer;
   res record;
BEGIN
   SELECT customerid INTO id FROM customers WHERE customerid = $1;
   IF not found THEN
      RAISE NOTICE 'Cliente inexistente en la base de datos';
   ELSE
      SELECT firstname, lastname, address1, address2, city, country INTO res
       FROM customers WHERE customerid=$1;
   END IF;
   RETURN res;
END;
$$ LANGUAGE plpgsql;

-- Mostrar cliente con identificador 1
SELECT * FROM f3_2(1) AS (nombre varchar(50), apellidos varchar(50), dir1
 varchar(50), dir2 varchar(50), ciudad varchar(50), pais varchar(50));
```

3_3.

```
CREATE FUNCTION f3_3(p_orderdate date) RETURNS SETOF orders AS
$$
DECLARE
   res orders;
BEGIN
   FOR res IN SELECT * FROM orders WHERE orderdate<$1 ORDER BY orderdate LOOP
      RETURN NEXT res;
   END LOOP;
   IF not found THEN
      RAISE NOTICE 'No existen órdenes previas a la fecha especificada';
   END IF;
END;
$$ LANGUAGE plpgsql;

-- Mostrar órdenes antes del 24 de abril de 1800
SELECT * FROM f3_3('1800-04-24');
```

3_4.

```
CREATE FUNCTION f3_4() RETURNS SETOF text AS
```

```
$$
DECLARE
   id integer;
   res text;
BEGIN
   SELECT prod_id INTO id FROM inventory WHERE quan_in_stock - sales = 0;
   IF not found THEN
      RAISE NOTICE 'No hay productos en esta situación';
   ELSE
      FOR res IN SELECT p.title FROM products p JOIN inventory i
       ON(p.prod_id = i.prod_id) WHERE quan_in_stock - sales = 0 LOOP
         RETURN NEXT res;
      END LOOP;
   END IF;
   RETURN;
END;
$$ LANGUAGE plpgsql;

-- Mostrar productos que no queden en existencia
SELECT * FROM f3_4();
```

3_5.

```
CREATE FUNCTION f3_5(p_category integer) RETURNS numeric AS
$$
DECLARE
   cat text;
BEGIN
   SELECT categoryname INTO cat FROM categories WHERE category = $1;
   IF cat = 'Games' OR cat= 'Music' THEN
      RETURN (SELECT avg(p.price) FROM products p JOIN categories c
       ON(p.category = c.category) WHERE p.category = $1);
   ELSE
      RETURN (SELECT min(p.price) FROM products p JOIN categories c
       ON(p.category = c.category) WHERE p.category = $1);
   END IF;
END;
$$ LANGUAGE plpgsql;

-- Trabajar con el precio del producto con identificador 10 según categoría
SELECT * FROM f3_5(10);
```

3_6.

```
CREATE FUNCTION f3_6(tab text, atr text, val text, cond text) RETURNS
 boolean AS
```

```
$$
BEGIN
   IF quote_ident($1) IN (SELECT tablename FROM pg_tables) THEN
      EXECUTE 'UPDATE ' || quote_ident($1) || ' SET ' || quote_ident($2) ||
       ' = ' || quote_nullable($3) || ' WHERE ' || $4;
      RETURN true;
   ELSE
      RAISE EXCEPTION 'Tabla no existente';
   END IF;
   RETURN false;
END;
$$ LANGUAGE plpgsql;

-- Actualizar dinámicamente la tabla categories
SELECT * FROM f3_6('categories','categoryname','Comedia silente',
 'category=1');
```

3_7.

```
CREATE FUNCTION f3_7(tab text) RETURNS SETOF record AS
$$
DECLARE
   res RECORD;
BEGIN
   IF (quote_ident($1) IN (SELECT tablename FROM pg_tables)) THEN
      FOR res IN EXECUTE 'SELECT * FROM ' || quote_ident($1) LOOP
         RETURN NEXT res;
      END LOOP;
   ELSE
      RAISE EXCEPTION 'Tabla no existente';
   END IF;
   RETURN;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM f3_7('categories') AS (id integer, nombre varchar(50));
```

3_8.

```
CREATE FUNCTION f3_8(p_category integer, monto numeric) RETURNS  SETOF
 products AS
$$
DECLARE
   res products;
BEGIN
   IF $1 IN (SELECT category FROM categories) THEN
```

```
      FOR res IN SELECT * FROM products WHERE category=$1 ORDER BY prod_id
       LOOP
         UPDATE products SET price = price + $2 WHERE prod_id = res.prod_id;
         RETURN NEXT res;
      END LOOP;
   ELSE
      RAISE EXCEPTION 'Categoría no existente';
   END IF;
   RETURN;
END;
$$ LANGUAGE plpgsql;

-- Actualizar precio de los productos de la categoría con identificador 1
SELECT * FROM f3_8(1, 5.00);
```

3_9.a.

```
-- Función disparadora
CREATE FUNCTION f3_9a() RETURNS trigger AS
$$
BEGIN
   RAISE NOTICE 'La categoría % no puede ser eliminada', OLD.categoryname;
   RETURN NULL;
END;
$$ LANGUAGE plpgsql;

-- Disparador
CREATE TRIGGER t1
BEFORE DELETE
ON categories
FOR EACH ROW
EXECUTE PROCEDURE f3_9a();

-- Eliminar una categoría para chequear funcionamiento del disparador
DELETE FROM categories WHERE category = 1;
```

3_9.b.

```
-- Función disparadora
CREATE FUNCTION f3_9b() RETURNS trigger AS
$$
BEGIN
   IF NEW.price > 200 THEN
      RAISE LOG 'El producto % tiene un precio superior a las 200 unidades',
       NEW.title;
   END IF;
```

```
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Disparador
CREATE TRIGGER t2
AFTER INSERT
ON products
FOR EACH ROW
EXECUTE PROCEDURE f3_9b();

-- Insertar un producto para chequear funcionamiento del disparador
INSERT INTO products VALUES (313231,1,'Vestido de novia','Laura de la Uz',
 201.00,1,1976);
```

3_9.b.i.

```
-- Disparador
CREATE TRIGGER t2i
AFTER INSERT or UPDATE
ON products
FOR EACH ROW
EXECUTE PROCEDURE f3_9b();

-- Actualizar un producto para chequear funcionamiento del disparador
UPDATE products SET price=210 WHERE prod_id = 1;
```

3_9.c.

```
-- Tabla eliminados
CREATE TABLE eliminados(
   id integer PRIMARY KEY,
   cat integer NOT NULL,
   title text NOT NULL,
   actor text NOT NULL,
   price numeric NOT NULL,
   special smallint NOT NULL,
   common integer NOT NULL);

-- Función disparadora
CREATE FUNCTION f3_9c() RETURNS trigger AS
$$
BEGIN
   INSERT INTO eliminados VALUES (OLD.prod_id, OLD.category, OLD.title,
    OLD.actor, OLD.price, OLD.special, OLD.common_prod_id);
   RETURN NULL;
END;
```

```
$$ LANGUAGE plpgsql;

-- Disparador
CREATE TRIGGER t3
AFTER DELETE
ON products
FOR EACH ROW
EXECUTE PROCEDURE f3_9c();

-- Eliminar un producto para chequear funcionamiento del disparador
DELETE FROM products WHERE prod_id = 313231;

-- Consultar tabla eliminados
SELECT * FROM eliminados;
```

3_9.d.

```
-- Tabla para guardar productos con precios actualizados
CREATE TABLE act_precios(
   id_prod integer NOT NULL,
   precio_viejo numeric NOT NULL,
   precio_nuevo numeric NOT NULL,
   fecha timestamp NOT NULL,
   PRIMARY KEY (id_prod, fecha));

-- Función disparadora
CREATE FUNCTION f3_9d() RETURNS trigger AS
$$
BEGIN
   IF NEW.price <> OLD.price THEN
      INSERT INTO act_precios VALUES (NEW.prod_id, OLD.price, NEW.price,
       now());
   END IF;
   RETURN NULL;
END;
$$ LANGUAGE plpgsql;

-- Disparador
CREATE TRIGGER t4
AFTER UPDATE
ON products
FOR EACH ROW
EXECUTE PROCEDURE f3_9d();

-- Actualizar un precio para chequear funcionamiento del disparador
UPDATE products SET price = 20.10 WHERE prod_id = 31;
```

```
-- Consultar tabla act_precios
SELECT * FROM act_precios;
```

3_9.d.i

```
-- Eliminar disparador y función disparadora
DROP TRIGGER t4 ON products;
DROP FUNCTION f3_9d();

-- Función disparadora
CREATE FUNCTION f3_9d() RETURNS trigger AS
$$
BEGIN
   INSERT INTO act_precios VALUES (NEW.prod_id, OLD.price, NEW.price, NOW());
   RETURN null;
END;
$$ LANGUAGE plpgsql;

-- Disparador
CREATE TRIGGER t4
AFTER UPDATE
ON products
FOR EACH ROW
WHEN (OLD.price <> NEW.price)
EXECUTE PROCEDURE f3_9d();

-- Actualizar un precio para chequear funcionamiento del disparador
UPDATE products SET price = 231.10 WHERE prod_id = 1;

-- Consultar tabla act_precios
SELECT * FROM act_precios;
```

3_9.d.ii

```
-- Eliminar disparador
DROP TRIGGER t4 ON products;

-- Crear disparador condicional
CREATE TRIGGER t4
AFTER UPDATE OF price
ON products
FOR EACH ROW
EXECUTE PROCEDURE f3_9d();

-- Actualizar un atributo que no sea el precio
UPDATE products SET title = 'Lucía' WHERE prod_id = 1;
```

```
-- Consultar tabla act_precios
SELECT * FROM act_precios;
```

3_9.e.

```
-- Función disparadora
CREATE FUNCTION f3_9e() RETURNS trigger AS
$$
DECLARE
   cant integer;
BEGIN
   SELECT count(*) INTO cant FROM orders WHERE customerid=NEW.customerid;
   IF cant BETWEEN 3 AND 5 THEN
      RAISE NOTICE 'Actualizando el impuesto rebajándolo un 5 porciento...';
      UPDATE orders SET tax = NEW.netamount * 0.05 WHERE orderid =
       NEW.orderid;
   ELSIF cant > 5 THEN
      RAISE NOTICE 'Actualizando el impuesto rebajándolo un 6 porciento...';
      UPDATE orders SET tax = NEW.netamount * 0.06 WHERE orderid =
       NEW.orderid;
   END IF;
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Disparador
CREATE TRIGGER t5
AFTER INSERT
ON orders
FOR EACH ROW
EXECUTE PROCEDURE f3_9e();

-- Insertar orden para chequear funcionamiento del disparador
INSERT INTO orders VALUES (12530,'2016-02-29',14809,80.00,5.6,87);
INSERT INTO orders VALUES (12531,'2016-02-29',17928,80.00,5.6,87);
```

3_9.f.

```
-- Eliminar todos los disparadores anteriores para que no interfieran
-- Función disparadora
CREATE FUNCTION f3_9f() RETURNS TRIGGER AS
$$
BEGIN
   IF ((TG_OP = 'INSERT' OR TG_OP = 'UPDATE') AND (NEW.price > 200)) THEN
      RAISE NOTICE 'Registrando en los logs producto con precio superior a
```

```
        las 200 unidades...';
        RAISE LOG 'El producto % tiene un precio superior a las 200 unidades',
         NEW.title;
    ELSIF (TG_OP = 'UPDATE' AND NEW.price <> OLD.price) THEN
        RAISE NOTICE 'Insertando en tabla act_precios el producto con precio
         modificado...';
        INSERT INTO act_precios VALUES (NEW.prod_id, OLD.price, NEW.price,
         now());
    ELSIF (TG_OP = 'DELETE') THEN
        RAISE NOTICE 'Insertando en tabla eliminados el producto borrado...';
        INSERT INTO eliminados VALUES (OLD.prod_id, OLD.category, OLD.title,
         OLD.actor, OLD.price, OLD.special, OLD.common_prod_id);
    END IF;
    RETURN NULL;
END;
$$LANGUAGE plpgsql;

-- Disparador
CREATE TRIGGER t7
AFTER INSERT or UPDATE or DELETE
ON products
FOR EACH ROW
EXECUTE PROCEDURE f3_9f();

-- Insertar, actualizar y eliminar productos para chequear disparador
INSERT INTO products VALUES (242424,1,'La pared de las palabras',
 'Isabel Santos',230.00,1,1976);
UPDATE products SET price = 25.00 WHERE prod_id = 242424;
SELECT * FROM act_precios;
DELETE FROM products WHERE prod_id = 18;
SELECT * FROM eliminados;
```

3_10.

```
-- Crear vista
CREATE VIEW producto_categoria AS
    SELECT p.title, c.categoryname
    FROM categories c JOIN products p USING (category);

-- Crear función disparadora para actualizar las tablas asociadas
CREATE FUNCTION f3_10() RETURNS trigger AS
$$
DECLARE
    id integer;
BEGIN
    SELECT prod_id INTO id FROM products WHERE title = OLD.title;
    IF NEW.title <> OLD.title THEN
```

```
      RAISE NOTICE 'Actualizando el título en la tabla products...';
      UPDATE products SET title = NEW.title WHERE prod_id = id;
   END IF;
   SELECT category INTO id FROM categories WHERE categoryname =
    OLD.categoryname;
   IF NEW.categoryname <> OLD.categoryname THEN
      RAISE NOTICE 'Actualizando el nombre de la categoría en la tabla
       categories...';
      UPDATE categories SET categoryname = NEW.categoryname WHERE category
       = id;
   END IF;
   RETURN NULL;
END;
$$ LANGUAGE plpgsql;

-- Crear disparador
CREATE TRIGGER tgr_act_producto_categoria
INSTEAD OF UPDATE
ON producto_categoria
FOR EACH ROW
EXECUTE PROCEDURE f3_10();

-- Actualizar vista
UPDATE producto_categoria SET title = 'Amor vertical', categoryname =
'Comedy' WHERE title='ACADEMY AGENT';

-- Consultar vista para chequear actualización
SELECT * FROM producto_categoria WHERE title = 'Amor vertical';
```

3_11.a.

```
CREATE FUNCTION f3_11a(p_category integer, p_monto numeric) RETURNS SETOF
 products AS
$$
DECLARE
   c1 refcursor;
   id integer;
   res products;
BEGIN
   IF $1 IN (SELECT category FROM categories) THEN
      OPEN c1 FOR SELECT prod_id FROM products WHERE category = $1;
      LOOP
         FETCH c1 INTO id;
         EXIT WHEN not found;
         UPDATE products SET price=price + $2 WHERE prod_id = id;
      END LOOP;
      CLOSE c1;
```

```
      OPEN c1 FOR SELECT * FROM products WHERE category = $1 ORDER BY
       prod_id;
      LOOP
         FETCH c1 INTO res;
         EXIT WHEN not found;
         RETURN NEXT res;
      END LOOP;
   ELSE
      RAISE EXCEPTION 'Categoría no existente';
   END IF;
   RETURN;
END;
$$ LANGUAGE plpgsql;

-- Quitar 1 unidad a los precios de los productos de la categoría 3
SELECT f3_11a(3,-1.00);
```

3_11.b.

```
CREATE FUNCTION f3_11b() RETURNS SETOF orders AS
$$
DECLARE
   c1 refcursor;
   id integer;
   cont integer := 0;
   res orders;
BEGIN
   OPEN c1 FOR SELECT orderid FROM orders WHERE orderdate = '2004-06-24'
    ORDER BY orderid; -- OFFSET 12 LIMIT 13: evita mover el cursor a la
    -- posición 11 y el contador, iterándose con el LOOP solamente
   MOVE ABSOLUTE 11 FROM c1;
   WHILE cont <= 13 LOOP
      FETCH c1 INTO id;
      DELETE FROM orders WHERE orderid = id;
      cont := cont + 1;
   END LOOP;
   CLOSE c1;
   OPEN c1 FOR SELECT * FROM orders WHERE orderdate = '2004-06-24'
    ORDER BY orderid;
   LOOP
      FETCH c1 INTO res;
      EXIT WHEN not found;
      RETURN NEXT res;
   END LOOP;
   CLOSE c1;
```

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

-- Eliminar órdenes del 24-06-2004 de la 12 a la 25
SELECT * FROM f3_11b()
```

3_11.b.i.

```
CREATE FUNCTION f3_11bi(fecha date, inicio integer, fin integer) RETURNS SETOF
 orders AS
$$
DECLARE
   c1 refcursor;
   id integer;
   cont integer := 0;
   res orders;
BEGIN
   OPEN c1 FOR SELECT orderid FROM orders WHERE orderdate = $1 ORDER BY
    orderid;
   MOVE ABSOLUTE ($2 - 1) FROM c1;
   WHILE cont <= ($3 - $2) LOOP
      FETCH c1 INTO id;
      DELETE FROM orders WHERE orderid = id;
      cont := cont + 1;
   END LOOP;
   CLOSE c1;
   OPEN c1 FOR SELECT * FROM orders WHERE orderdate = $1 ORDER BY orderid;
   LOOP
      FETCH c1 INTO res;
      EXIT WHEN not found;
      RETURN NEXT res;
   END LOOP;
   CLOSE c1;
   RETURN;
END;
$$ LANGUAGE plpgsql;

-- Eliminar las órdenes 2 y 3 del 25-06-2004
SELECT * FROM f3_11bi('2004-06-25',2,3)
```

3_11.b.ii.

```
CREATE FUNCTION f3_11bii(fecha date, inicio integer, fin integer) RETURNS
 SETOF orders AS
$$
```

```
DECLARE
   c1 refcursor;
   cont integer := 0;
   res orders;
BEGIN
   OPEN c1 FOR SELECT orderid FROM orders WHERE orderdate = $1 ORDER BY
    orderid;
   MOVE ABSOLUTE ($2 - 1) FROM c1;
   WHILE cont <= ($3 - $2) LOOP
      MOVE c1;
      DELETE FROM orders WHERE CURRENT OF c1;
      cont := cont + 1;
   END LOOP;
   CLOSE c1;
   OPEN c1 FOR SELECT * FROM orders WHERE orderdate = $1 ORDER BY orderid;
   LOOP
      FETCH c1 INTO res;
      EXIT WHEN not found;
      RETURN NEXT res;
   END LOOP;
   CLOSE c1;
   RETURN;
END;
$$ LANGUAGE plpgsql;

-- Eliminar las órdenes 2 y 3 del 25-06-2004
SELECT * FROM f3_11bii('2004-06-25',2,3)
```

3_11.c.

```
CREATE FUNCTION f3_11c() RETURNS SETOF customers AS
$$
DECLARE
   c1 refcursor;
   res customers;
BEGIN
   OPEN c1 FOR SELECT c.* FROM customers c JOIN orders o ON (c.customerid =
    o.customerid) GROUP BY c.customerid HAVING count(*) > 4;
   LOOP
      FETCH c1 INTO res;
      EXIT WHEN not found;
      UPDATE orders SET tax = tax - (totalamount * 0.01) WHERE customerid =
       res.customerid;
      RETURN NEXT res;
   END LOOP;
```

```
   CLOSE c1;
   RETURN;
END;
$$ LANGUAGE plpgsql;

-- Rebajar impuesto a clientes con más de 10 órdenes
SELECT * FROM f3_11c()
```

3_11.d.

```
CREATE FUNCTION f3_11d(p_age smallint, p_tax smallint) RETURNS SETOF record
 AS
$$
DECLARE
   c1 CURSOR FOR SELECT c.customerid, c.firstname, c.lastname, c.age,
    c.country, avg(o.tax) FROM customers c JOIN orders o USING (customerid)
    WHERE c.age > $1 GROUP BY c.customerid;
   res record;
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO res;
      EXIT WHEN not found;
      UPDATE orders SET tax = tax - $2 WHERE customerid = res.customerid;
      RETURN NEXT res;
   END LOOP;
   CLOSE c1;
   RETURN;
END;
$$ LANGUAGE plpgsql;

-- Rebajar impuesto en 5 a clientes con más de 32 años
SELECT * FROM f3_11d(32::smallint,5::smallint) AS (customerid integer,
 firstname character varying(50),lastname character varying(50),
 age smallint,country character varying(50),avg numeric)
```

3_12.a.

```
CREATE FUNCTION f3_12a(prod refcursor) RETURNS refcursor AS
$$
BEGIN
   OPEN prod FOR SELECT * FROM products ORDER BY prod_id;
   RETURN prod;
END;
$$ LANGUAGE plpgsql;

-- Comenzar transacción
```

```
BEGIN;

-- Invocar función que retorna un cursor que apunta a los 10000 productos
SELECT f3_12a('prod');

-- Cargar los primeros 5000 productos
FETCH 5000 IN "prod";

-- Cargar los productos del 5001 al 10000
FETCH 5000 IN "prod";

-- Cerrar cursor
CLOSE "prod";

-- Confirmar transacción
COMMIT;
```

3_12.b.

```
CREATE FUNCTION f3_12b(cust refcursor) RETURNS refcursor AS
$$
BEGIN
   OPEN cust FOR SELECT firstname, lastname FROM customers ORDER BY firstname;
   RETURN cust;
END;
$$ LANGUAGE plpgsql;

-- Comenzar transacción
BEGIN;

-- Invocar función que retorna un cursor que apunta a los clientes
SELECT f3_12b('cust');

-- Cargar todos los clientes
FETCH ALL IN "cust";

-- Cerrar cursor
CLOSE "cust";

-- Confirmar transacción
COMMIT;
```

3_12.c.

```
CREATE FUNCTION f3_12c(total_pedidos integer, monto numeric, prod refcursor)
 RETURNS refcursor AS
$$
DECLARE
   c1 CURSOR FOR SELECT prod_id FROM orderlines GROUP BY prod_id HAVING
    count(*) > $1;
   id_prod integer;
```

```
BEGIN
   OPEN c1;
   LOOP
   FETCH c1 INTO id_prod;
   EXIT WHEN not found;
   UPDATE products SET price=price-$2 WHERE prod_id=id_prod;
   END LOOP;
   CLOSE c1;
   OPEN prod FOR SELECT p.* FROM products p JOIN orderlines o USING (prod_id)
    GROUP BY prod_id HAVING count(*) > $1;
   RETURN prod;
END;
$$ LANGUAGE plpgsql;

-- Comenzar transacción
BEGIN;

-- Invocar función que rebaja el precio en 1.00 a los productos con más de 16
-- pedidos asociados y retorna un cursor que apunta a dichos pedidos
SELECT f3_12c(16, 1.00, 'prod');

-- Cargar todos los productos
FETCH ALL IN "prod";

-- Cerrar cursor
CLOSE "prod";

-- Confirmar transacción
COMMIT;
```

Ejercicios del Capítulo 4

4_1.

```
CREATE FUNCTION f4_1() RETURNS numeric AS
$$
   resultado = plpy.execute("SELECT sum(price*quantity) AS monto_total FROM
    products JOIN orderlines ON (orderlines.prod_id=products.prod_id) WHERE
    actor='VIVIEN COOPER'")
   return resultado[0]['monto_total']
$$ LANGUAGE plpythonu;
```

4_2.

```
CREATE FUNCTION f4_2() RETURNS TABLE (producto text, precio numeric) AS
$$
   resultado = plpy.execute("SELECT title, price FROM products ORDER BY
    price DESC LIMIT 100")
```

```
    for tupla in resultado:
        yield (tupla['title'], tupla['price'])
$$ LANGUAGE plpythonu;
```

4_3.

```
CREATE FUNCTION f4_3(ruta_nombre_archivo text) RETURNS text AS
$$
    import csv
    resultado = plpy.execute("SELECT title, price FROM products ORDER BY
     price DESC LIMIT 100")
    with open(ruta_nombre_archivo, 'w') as csvfile:
        encabezado = ['producto', 'precio']
        archivo = csv.DictWriter(csvfile, fieldnames=encabezado)
        archivo.writeheader()
        for tupla in resultado:
            archivo.writerow({'producto': tupla['title'], 'precio':
             tupla['price']})
    return ruta_nombre_archivo
$$ LANGUAGE plpythonu;

SELECT f4_3('/tmp/archivo.csv');
```

4_4.

```
CREATE FUNCTION f4_4(caracter text) RETURNS TABLE (nombre_producto text) AS
$$
    miplan = plpy.prepare("SELECT title FROM products WHERE title LIKE $1",
     ["text"])
    resultado = plpy.execute(miplan, [caracter])
    for tupla in resultado:
        yield (tupla['title'])
$$ LANGUAGE plpythonu;

SELECT f4_4('%O');
```

4_5.

```
CREATE FUNCTION f4_5(ruta_nombre_archivo text, caracter text) RETURNS text AS
$$
    import csv
    resultado = plpy.execute("SELECT title, price FROM products ORDER BY price
     DESC LIMIT 100")
    miplan = plpy.prepare("SELECT title FROM products WHERE title LIKE $1",
     ["text"])
    resultado = plpy.execute(miplan, [caracter])
```

```
   with open(ruta_nombre_archivo, 'w') as csvfile:
      encabezado = ['producto']
      archivo = csv.DictWriter(csvfile, dialect=csv.excel,
       fieldnames=encabezado)
      archivo.writeheader()
      for tupla in resultado:
         archivo.writerow({'producto': tupla['title']})
   return ruta_nombre_archivo
$$ LANGUAGE plpythonu;

SELECT f4_5('/tmp/archivo.csv', '%O');
```

4_6.

```
CREATE FUNCTION f4_6() RETURNS trigger AS
$$
   import smtplib
   from email.MIMEText import MIMEText
   if TD["event"] == "UPDATE" and TD["new"]['price'] == 0:
      plpy.notice('Enviando correo por modificacion de precio del producto: '
       + TD["new"]['title'])
      #enviando correo
      mens = MIMEText('Enviando correo por modificacion de precio del
       producto: ' + TD["new"]['title'])
      # Conex con el server GMAIL
      mens['Subject'] = 'Producto modificado con precio 0'
      mens['From'] = "usuario@gmail.com"
      mens['To'] = "admin@dellstore.com"
      serversmtp = smtplib.SMTP("smtp.gmail.com", 587)
      serversmtp.ehlo_or_helo_if_needed()
      serversmtp.starttls()
      serversmtp.ehlo_or_helo_if_needed()
      serversmtp.login("usuario@gmail.com", "contrasena")
      # enviando
      serversmtp.sendmail("usuario@gmail.com","admin@dellstore.com",
       mens.as_string())
      # cerrando
      serversmtp.close()
   return "OK"
$$ LANGUAGE plpythonu;


-- Disparador
CREATE TRIGGER tgr_plpython
AFTER UPDATE
```

```
ON products
FOR EACH ROW
EXECUTE PROCEDURE f4_6();

-- Actualización sobre producto con identificador 1 para disparar la función
UPDATE products SET price = 0 WHERE prod_id=1;
```

4_7.

```
CREATE FUNCTION f4_7(precio numeric) RETURNS TABLE (producto text) AS
$$
   consulta<-paste(c("SELECT title FROM products WHERE price >", precio),
    collapse = " ")
   resultado <- pg.spi.exec(consulta)
   return (resultado)
$$ LANGUAGE plr;

SELECT f4_7(29);
```

4_8.

```
CREATE FUNCTION f4_8() RETURNS integer AS
$$
   png(paste("clientesxpaises", "png", sep="."))
   resultado <- pg.spi.exec("SELECT count(*) FROM customers GROUP BY country
    ORDER BY country")
   paises <- pg.spi.exec("SELECT DISTINCT country FROM customers ORDER BY
    country")
   pie (as.matrix(resultado), col=rainbow(length(as.matrix(resultado))),
    labels=as.matrix(paises), main="Cantidad de clientes por paises")
   dev.off()
$$ LANGUAGE plr;

SELECT f4_8();
```

4_9.

```
CREATE FUNCTION f4_9() RETURNS numeric AS
$$
   resultado <- pg.spi.exec("SELECT price::numeric FROM products WHERE
    special  = 1 ")
   mediana <- median(resultado[,1])
   return (mediana)
$$ LANGUAGE plr;

SELECT f4_9();
```