PL/PGSQL Y OTROS LENGUAJES PROCEDURALES EN POSTGRESQL

Guía para el desarrollo de lógica de negocio del lado del servidor

Anthony R. Sotolongo León

Yudisney Vazquez Ortíz

ISBN : 978-1-312-99489-8, Edición 1
La Habana, 2015

CONTENIDOS

ÍNDICE DE EJEMPLOS

PREFACIO

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

Surge en respuesta a la necesidad de contar con bibliografía o 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 2, 3 y 4 ejercicios en los que se deben aplicar los elementos abordados en ellos para su solución.

¿Qué cubre el libro?

Se encuentra dividido en cuatro capítulos:

- **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:** 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:** aborda cómo programar funciones en PL/pgSQL; explicándose su estructura, cómo trabajar con variables y sentencias, las estructuras condicionales y de control que implementa y cómo hacer uso de los 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 detalles 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 PostgreSQL y, cómo realizar con ellos el retorno de valores, la ejecución de consultas y la definición de disparadores.

¿Qué necesita para trabajar con este libro?

Para que este 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 9.3 (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 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 2, 3 y 4; puede ser accedida desde las direcciones http://linux.dell.com/dvdstore/ y http://pgfoundry.org/projects/dbsamples/.

¿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 luego, tendrá las habilidades suficientes para comprender y poder hacer uso de los lenguajes procedurales de desconfianza PL/Python y PL/R.

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 escribirán en Droid Sans a 10 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_clientesmenores() RETURNS void AS
$$
        DELETE FROM customers WHERE age <= 18;
$$ LANGUAGE sql;
```

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

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

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

- Notas: acotaciones sobre lo que se esté discutiendo se escribirán en Roboto, cursivas a 10 puntos y enmarcadas, de la forma:

Para analizar en detalles el acotado a emplear en el cuerpo de una función puede remitirse a las Secciones String Constants y Dollar-quoted String Constants de la Documentación Oficial de PostgreSQL

Errores

De encontrarse cualquier error se le agradecería que lo comunicara a los autores.

AUTORES

Anthony R. Sotolongo León (asotolongo@gmail.com**).** Profesor de la Universidad de las Ciencias Informáticas y miembro de la Comunidad Cubana de PostgreSQL. Obtuvo el grado de Máster en Informática Aplicada en el año 2010 en la Universidad. Ha impartido en el pregrado durante 8 años asignaturas relacionadas con las tecnologías de bases de datos como Sistemas de Bases de Datos I, Sistemas de Bases de Datos II y Optimización de bases de datos. Imparte hace 3 ediciones las asignaturas Introductorio a PostgreSQL, Programación en PostgreSQL y Réplica de datos en PostgreSQL del Diplomado en tecnologías de bases de datos PostgreSQL, del cual es Coordinador. Organiza y coordina eventos relacionados con el gestor en Cuba y ha impartido varias charlas sobre PostgreSQL.

Yudisney Vazquez Ortíz (yvazquezo@uci.cu, yvazquezo@gmail.com**).** Profesora de la Universidad de las Ciencias Informáticas y miembro de la Comunidad Cubana de PostgreSQL. Obtuvo el grado de Máster en Gestión de Proyectos Informáticos en el año 2011 en la Universidad. Ha impartido en el pregrado durante 7 años asignaturas relacionadas con las tecnologías de bases de datos como Sistemas de Bases de Datos I, Sistemas de Bases de Datos II y Optimización de bases de datos. Imparte hace 2 ediciones las asignaturas Seguridad en PostgreSQL y Programación en PostgreSQL del Diplomado en tecnologías de bases de datos PostgreSQL, del cual es miembro de su Comité Académico. Organiza y coordina eventos relacionados con el gestor en Cuba.

COLABORADORES

Ing. Daymel Bonne Solís

Ing. Marcos Luis Ortiz Valmaseda

Ing. Adalennis Buchillón Sorís

1. 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 datos de las aplicaciones informáticas, las operaciones con dichas bases de datos suelen ser comúnmente mediante sentencias SQL, lenguaje por defecto para interactuar con las mismas.

Ver los gestores de bases de datos relacionales puramente para almacenar datos es restringir sus potencialidades de trabajo, puesto que brindan algo más que un lugar para acumular datos, ejemplo de ello son los procedimientos almacenados y los disparadores (*triggers*), 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 controles de flujo o utilización de variables para retener un dato determinado, impidiendo 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 brinda las características de programación del lado del servidor desde sus inicios, que a partir de la versión 7.2 se mejoraron considerablemente y se le agregaron otros lenguajes para la programación (llamados lenguajes de desconfianza), además del conocido Estándar SQL. Paulatinamente se han ido agregando 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 Funciones Definidas por el Usuario (FDU por sus siglas en inglés), también llamadas en otros gestores como procedimiento almacenados. Estas funciones se pueden entender como el conjunto agrupado de operaciones que se ejecutan del lado del servidor y que pueden derivar en acciones sobre los datos; pueden utilizar otras características del gestor como los tipos de datos y operadores personalizados, reglas, vistas, etc. Las Funciones Definidas por el Usuario pueden clasificarse en cuatro tipos según la sección *User-defined Functions* de la Documentación Oficial:

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

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

Es válido aclarar que todo lo que ocurre dentro de una Función Definida por el Usuario en PostgreSQL lo hace de forma transaccional, es decir, se ejecuta todo o nada; de ocurrir algún fallo el propio PostgreSQL realiza un *RollBack* deshaciendo las operaciones realizadas.

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 OR REPLACE 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 OR REPLACE FUNCTION sumar(valor1 int, valor2 int) RETURNS int AS
$$
BEGIN
        RETURN $1 + $2;
END;
$$ LANGUAGE plpgsql;
```

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

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

Para invocar una función se debe ejecutar la sentencia:

SELECT *nombre_función*([*parámetro_función*1,...])

Si dentro de una función se realiza el llamado a otra función basta con colocar su nombre sin necesidad de utilizar la sentencia `SELECT`.

Los siguientes ejemplos ilustran cómo realizar invocaciones de funciones.

Ejemplo 3: Invocación de funciones en PostgreSQL

```
postgres=# SELECT version();
                                                version
-------------------------------------------------------------------------------------------------
 PostgreSQL 9.3.1, compiled by Visual C++ build 1600, 64-bit

(1 fila)
```

version() es una función interna de PostgreSQL. Puede encontrar otras funciones internas en la Documentación Oficial de PostgreSQL en la sección Functions and Operators

Ejemplo 4: Invocación de la suma definida en el ejemplo 2

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

(1 fila)
```

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: presentación 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 capa de datos con el gestor de bases de datos, capa de negocio con el código de las aplicaciones y capa de presentación con la interfaz que se le presenta al usuario.

Ahora, 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 capa de negocio el código del negocio (que puede estar en el gestor de bases de datos), y como 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 cuales se resumen en:

- Rendimiento: si la lógica de negocio se realiza de lado del servidor se evita 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 cambia por algún motivo los cambios se realizan en la base de datos central y pueden ser realizados fácilmente mediante el Lenguaje de Definición de Datos (DDL) para actualizar las funciones implicadas en los cambios.
- Simple modo de garantizar la seguridad en la lógica de negocio: a las funciones se les pueden definir permisos de acceso a través del Lenguaje de Control de Datos (DCL), además de evitar que los datos viajen por la red.

Otra ventaja es que evita la necesidad de reescribir código de negocio, imagine que se tiene una base de datos de la que consumen datos varias aplicaciones escritas en diferentes lenguajes como Pascal, *Java*, PHP o *Python* y, que se tenga que realizar una operación de inserción y actualización de datos; si dicha operación está en el lado del servidor de bases de datos solo sería ordenar su ejecución desde las distintas aplicaciones.

1.3 Usos de la lógica de negocio del lado del servidor

Transacciones que incluyan varias operaciones

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

Ejemplo 5: Función que actualiza el género de una canción

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

Auditorías de datos

La auditoría de datos es la operación de chequear las operaciones realizadas sobre los datos con el objetivo de detectar anomalías u operaciones 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 siguientes módulos desarrollados para esta tarea:

- Pgaudit: disponible en https://github.com/jcasanov/pg_audit.
- Audit-triggers: disponible https://github.com/2ndQuadrant/audit-trigger.

Dichos ejemplos son un conjunto de funciones y disparadores que posibilitan detectar cambios realizados sobre los datos y, son configurables para trabajar sobre las tablas deseadas.

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 siguiente.

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)
```

Una vez invocada la función se muestra el gráfico mostrado en la figura siguiente.

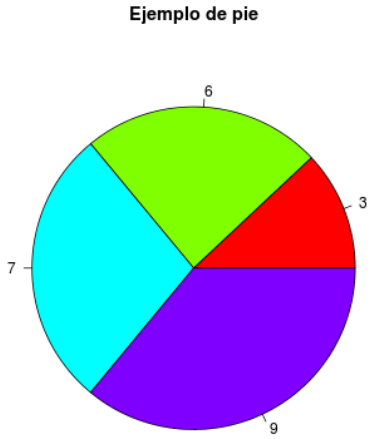

Figura 1: Gráfica generada con PL/R

Importación y exportación de datos

Desde PostgreSQL se puede realizar exportación o importación de datos, por ejemplo, si se necesita exportar el resultado de una consulta a un formato CSV, se puede desarrollar una función con PL/pgSQL que realice dicha actividad como muestra el ejemplo siguiente.

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

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

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.

1.4 Modelo de datos para el trabajo en 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 http://linux.dell.com/dvdstore/ y http://pgfoundry.org/projects/dbsamples/.

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 manteniendo, así como el evitar reescritura de código.

PostgreSQL como sistema de gestión de bases de datos relacional permite la programación del lado del servidor, principalmente mediante 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 el lector podrá utilizarlas para potenciar las funcionalidades de este gestor.

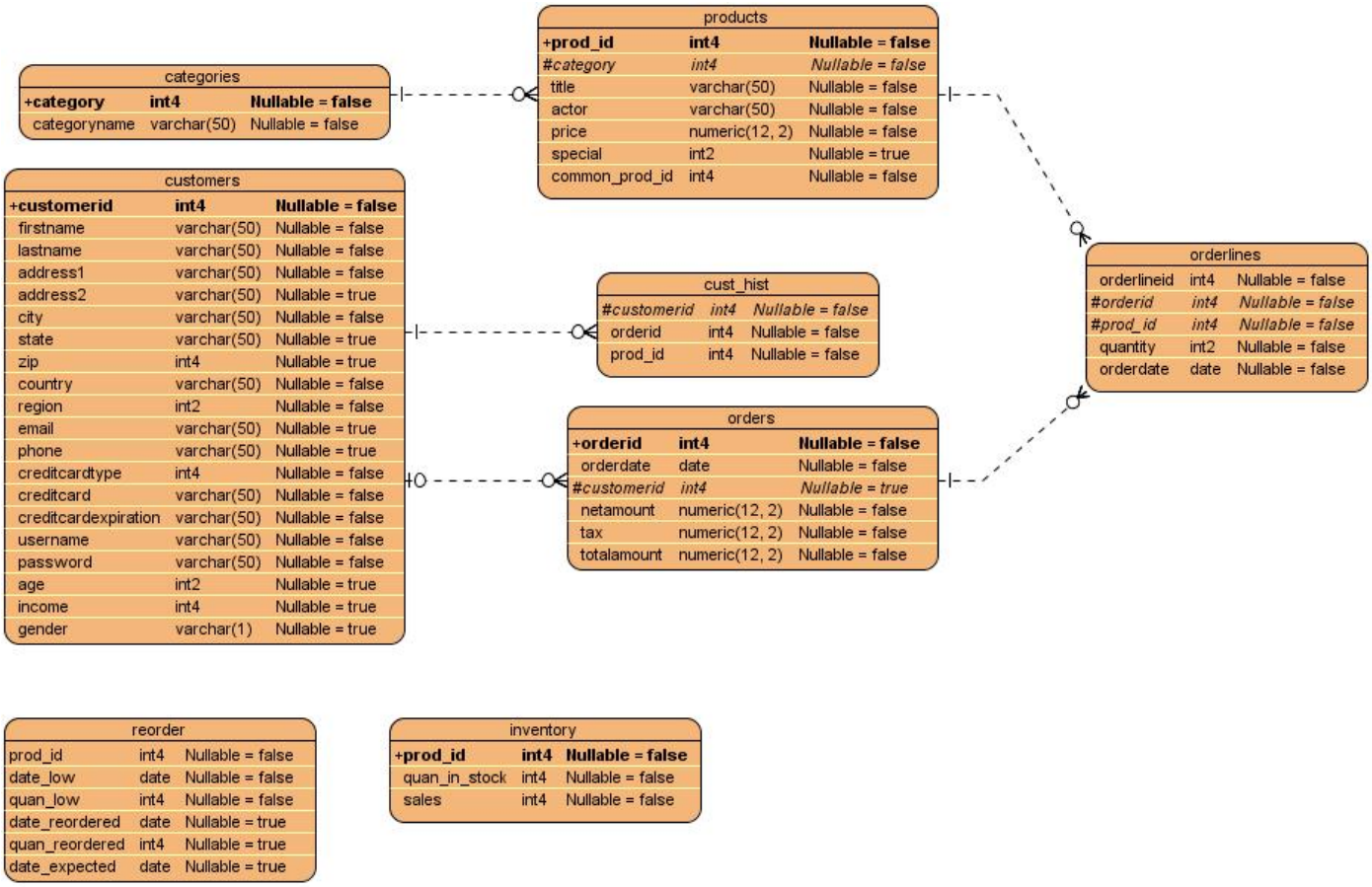

Figura 2: Modelo de datos de la base de datos Dell Store 2

2. 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 sencillos la sintaxis básica para la creación de funciones, el empleo de parámetros y los tipos existentes para el retorno de valores, de forma que se pueda posteriormente enfrentar el trabajo con lenguajes procedurales de mayores potencialidades para la extensión del gestor.

Para estudiar en detalle el lenguaje SQL puede consultar libros como *Understanding the New SQL*, *A Guide to the SQL Standard* y la propia Documentación de PostgreSQL.

2.2 Extensión de PostgreSQL con funciones SQL

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 en dicha lista; por lo que cualquier colección de comandos SQL puede ser empaquetada y definida como una función.

Además de consultas 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…) y algunos de utilidad (VACUUM…).

2.3 Sintaxis para la definición de una función SQL

Para la definición de una función 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;
```

Para analizar la sintaxis ampliada de definición de una función puede remitirse a la Documentación Oficial de PostgreSQL en el capítulo Reference, epígrafe SQL Commands

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.

Por ejemplo, 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 detalles de los tipos de retorno vea el epígrafe 2.5 Retorno de valores.

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

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 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.

Para analizar en detalles el acotado a emplear en el cuerpo de una función puede remitirse a las Secciones String Constants y Dollar-quoted String Constants de la Documentación Oficial de PostgreSQL. En este libro se empleará el $$ para acotar el cuerpo de las funciones

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 vieja, 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 intentarse definir la función, PostgreSQL arroja un error especificando que ya existe una función con el mismo nombre y parámetros.

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() existente 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.

2.4 Parámetros de funciones SQL

Los parámetros de una función especifican aquellos valores que el usuario define sean empleados para su procesamiento posterior en el cuerpo de la función 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*

El modo del 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 la función.

- OUT: parámetro de salida, forma parte del resultado de la función y no se incluye en la invocación de la función.
- INOUT: parámetro de entrada/salida, puede ser empleado indistintamente para que forme parte de la lista de parámetros de entrada y que sea parte 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.

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

Los parámetros pueden ser referenciados en el cuerpo de la función usando sus nombres (a partir de la versión 9.2 de PostgreSQL) 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.

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 que en la sentencia de inserción se hace referencia a los parámetros mediante sus nombres, especificados en la definición de los parámetros.

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

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

Note que puede perfectamente omitirse el nombre de los parámetros y solamente especificar el tipo de dato, en cuyo caso se hace referencia a ellos mediante el signo $ y el número que ocupan en el listado de parámetros definidos.

Ejemplo 13: Empleo de parámetros con iguales nombres que columnas en tabla empleada 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;
```

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 los parámetros con el nombre de la función.

Ejemplo 14: Empleo de parámetros de tipo compuesto

```
CREATE FUNCTION insertar_producto(categoria categories, id integer, titulo varchar, actor varchar, precio float, especial integer, id_comun integer) RETURNS void AS
$$
    INSERT INTO products VALUES (categoria.category, id, titulo, actor, precio, especial, id_comun);
$$ 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 el parámetro accediendo al elemento *category*.

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
dell=# SELECT mostrar_nombrecompleto(20);
  mostrar_nombrecompleto
- - - - - - - - - - - - - - - - - -
        (IAYPUX,YELMUQZEHW)
(1 fila)
-- Invocación de la función empleando otra forma
dell=# SELECT first, last FROM mostrar_nombrecompleto(20);
+-------------+-------------+
|    first    |    last     |
+-------------+-------------+
|   IAYPUX    | YELMUQZEHW  |
+-------------+-------------+
(1 fila)
```

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 se debe especificar el tipo de retorno de la función haciendo uso de la cláusula RETURNS.

2.5 Retorno de valores

El tipo de retorno de las funciones especifica el tipo de dato que debe devolver la función 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.

Retorno de tipos de datos básicos

Las funciones SQL más simples no tienen parámetros o retornan 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(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.

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

```
CREATE FUNCTION incrementar_precio(prod integer) RETURNS numeric AS
$$
        UPDATE products SET price = price + 0.05 * price WHERE prod_id = prod;
        SELECT price FROM products WHERE prod_id = prod;
$$ LANGUAGE sql;
-- El mismo ejemplo empleando la cláusula RETURNING en el UPDATE
CREATE FUNCTION incrementar_precio(prod integer) RETURNS numeric AS
$$
        UPDATE products SET price = price + 0.05 * price WHERE prod_id = prod
        RETURNING price;
$$ LANGUAGE sql;
```

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 función, que da respuesta a la misma necesidad, se garantiza el retorno de la función mediante la cláusula RETURNING en el UPDATE.

Empleo y retorno de tipos de datos compuestos

Cuando se escriben funciones que utilicen 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 determinado pasándosele como parámetro un producto

```
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 al 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 es que no se necesita conocer previamente un atributo en particular y, se le puede aumentar el precio o lo que se desee hacer 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 aumentar 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 * 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: Empleo de la función mostrar_cliente para devolver 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 aún, 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 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 de una consulta (que devuelve el resultado enumerando los valores de las columnas separados por coma); el problema de utilizar esta forma surge cuando se ponen como parte de la lista del SELECT más de una función que retorna un conjunto de valores, lo que implica que el resultado puede no tener mucho sentido, este es uno de los motivos por el 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 pudiera implementarse una función como la mostrada en el ejemplo 22.

Ejemplo 22: Empleo de la función listar_productos para devolver el listado de productos existente

```
CREATE FUNCTION listar_productos() RETURNS SETOF products AS
$$
        SELECT * FROM products;
$$ 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 desde el 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: Empleo de la función mostrar_productos para devolver 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. Este tipo se emplea cuando el resultado:

- No es la estructura completa de una tabla, por ejemplo cuando se quieren obtener solamente un conjunto de los atributos existentes, como en el caso anterior.
- Se obtiene de concatenaciones entre tablas, por ejemplo cuando se desea obtener 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. El mismo posee la ventaja de que fue añadido 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 implementarse de la forma mostrada en el ejemplo 24 empleándose RETURNS TABLE.

Ejemplo 24: Función mostrar_productos para devolver nombre y precio de los 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.

2.6 Resumen

Las funciones SQL son la forma más sencilla de agregar código al núcleo de 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, el tipo de retorno de la función y el cuerpo de la misma, en el que se listan las sentencias SQL que conformarán la función.

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).

2.7 Para hacer con SQL

1. Implemente funciones que permitan la inserción de datos en cada una de las tablas:
    a. categories
    b. products
    c. customers
    d. orderlines
    e. orders
2. Desarrolle una función que dado el identificador de un producto devuelva toda la información referente a él.
3. Cree una función que dado el identificador de un cliente devuelva su nombre y apellidos, edad, género, correo electrónico, teléfono, ciudad y país.
    a. Emplee parámetros de salida.
    b. ¿Qué tipo de dato pudiera emplearse para no tener que declarar tantos parámetros de salida?
4. Obtenga 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. Elabore 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. Implemente una función que devuelva todos los clientes de sexo femenino y menores de 30 años.
    a. ¿De qué formas puede definirse el tipo de retorno de esta función? ¿Cuál es la diferencia entre ellas?
7. Desarrolle 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. Cree 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. Obtenga una función que muestre por categoría el total de productos existentes.

3. 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 y disparadores, 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 de 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 (*triggers*). Todo esto le otorga mayor potencialidad al combinar las ventajas de un lenguaje procedural y la facilidad de SQL.

3.2 Estructura de PL/pgSQL

Al ser PL/pgSQL un lenguaje estructurado por bloques, su definición 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.

Esta 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 OR REPLACE FUNCTION sumar(int, int) RETURNS int AS
$$
BEGIN
        RETURN $1 + $2;
END;
$$ LANGUAGE plpgsql;
-- Invocación de la función
dell=# SELECT sumar(2, 3);
 sumar
- - - - - - - -
     5
(1 fila)
```

Nótese que en este lenguaje procedural, al igual que en SQL, se mantienen las sentencias terminadas con punto y coma (;) al final de cada línea y para retornar el resultado se emplea la palabra reservada RETURN (para más detalles vea el epígrafe 3.6 Retorno de valores).

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

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

Cada sentencia en el bloque de sentencias puede ser un sub-bloque, empleado para realizar agrupaciones lógicas o para crear variables para un grupo de sentencias. Las variables de bloques externos pueden ser accedidas en un sub-bloque calificándolas con la etiqueta del bloque al que pertenecen. El ejemplo 26 muestra el tratamiento de variables en bloques anidados y el acceso a variables externas mediante su calificación con el nombre del bloque al que pertenecen.

Ejemplo 26: Empleo de variables en bloques anidados

```
CREATE FUNCTION incrementar_precio_porciento(id integer) RETURNS numeric AS
$$
<<principal>>
DECLARE
    incremento numeric := (SELECT price FROM products WHERE prod_id = $1) * 0.3;
BEGIN
    RAISE NOTICE 'El precio después del incremento será de % pesos', incremento;
    -- Muestra el incremento en un 30%
    <<excepcional>>
    DECLARE
        incremento numeric := (SELECT price FROM products WHERE prod_id = $1) * 0.5;
    BEGIN
        RAISE NOTICE 'El precio después del incremento excepcional será de % pesos', incremento; -- Muestra el incremento en un 50%
        RAISE NOTICE 'El precio después del incremento será de % pesos', principal.incremento; -- Muestra el incremento en un 30%
    END;
    RAISE NOTICE 'El precio después del incremento será de % pesos', incremento;
    -- Muestra el incremento en un 30%
    RETURN incremento;
END;
$$ LANGUAGE plpgsql;
--Invocación de la función
dell=# SELECT * FROM incrementar_precio_porciento(1);
NOTICE: El precio después del incremento será de 7.797 pesos
NOTICE: El precio después del incremento excepcional será de 12.995 pesos
```

```
NOTICE: El precio después del incremento será de 7.797 pesos
 incrementar_precio_porciento
------------------------------
                        7.797
(1 fila)
```

3.3 Trabajo con 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 a la variable es el valor con que se mantendrá, de no ser especificada 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.

Pueden declarase variables de la forma mostrada en los siguientes ejemplos:

```
impuesto CONSTANT numeric := 0.3;   -- variable de tipo numeric con un valor constante
                                    -- de 0.3
incremento integer DEFAULT 31;      -- variable de tipo entero con valor 31
region varchar := 'Occidente';      -- variable de tipo varchar con el valor Occidente
                                    -- asignado
fecha date NOT NULL:= '2007-11-24'; -- variable de tipo 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.

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

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 FOR y del que sus campos pueden ser accesibles calificándolos, como cualquier otro tipo de dato compuesto (ver epígrafe Empleo y retorno de tipos de datos compuestos).

Para emplearlo se debe tener en cuenta que:

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

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

*nombre_tabla***%ROWTYPE**

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

variable nombre_tabla

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 les permite definir sus propios tipos de datos 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 de una función un resultado compuesto por elementos de varias tablas.

Dicho resultado consiste que 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* [,…]]),

Un ejemplo de su empleo pudiera ser:

```
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 del comando SELECT o FOR.

RECORD no es un tipo de dato verdadero sino un contenedor. Este tipo de dato no es el mismo concepto que cuando se declara una función para que 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, mientras que la de la variable puede ser cambiada en tiempo de ejecución.

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

Parámetros de funciones

En PL/pgSQL al igual que en SQL (ver epígrafe 2.4 Parámetros de funciones SQL), se hace referencia a los parámetros de las funciones mediante la numeración $1, $2, $n o mediante un alias, que puede crearse de 2 formas:

- Nombrar el parámetro en el comando CREATE FUNCTION (forma preferida).

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

Es válido aclarar que el comando ALIAS se emplea no sólo 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 o record o algún elemento de un arreglo.
- expresión: debe ser devolver un único valor.

Ejemplos de asignaciones son los siguientes:

```
pais := 'Cuba';

impuesto := price*0.2;

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, lo que puede hacerse añadiendo 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 STRICT no es especificado, 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. De ser especificado, si la consulta devuelve más de una fila se genera un error en tiempo de ejecución. No obstante, en el caso de las consultas INSERT, UPDATE o DELETE,

aun cuando no sea especificado el STRICT se genera el error si el resultado tiene más de una fila, ya que como estas no tienen la opción ORDER BY no se puede determinar cuál de las filas del resultado se debería devolver.

Los ejemplos siguientes muestran el empleo de estos comandos para capturar una fila del resultado.

Ejemplo 28: Captura de una fila resultado de consultas SELECT, INSERT, UPDATE o DELETE

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

Ejecución de comandos dinámicos

Existen escenarios donde se hace inevitable generar comandos dinámicos en las funciones PL/pgSQL, o sea, comandos que involucren diferentes tablas o tipos de datos cada vez que sean ejecutados. Para ello 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 ser de tipo RECORD, fila o lista de variables simples separados por coma, con las especificaciones explicadas previamente para el empleo de la cláusula INTO (ver epígrafe Ejecución de consultas que arrojan resultados con una única fila), que de no ser especificada son descartadas las filas resultantes.
- expresión USING: suministra valores a ser insertados en el comando.

La sentencia ejecutada con este comando es planeada cada vez que es ejecutado 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 literal, pero es empleada cuando pueden haber parámetros nulos, retornando una cadena nula y no derivando en un error de EXECUTE 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, ya que los parámetros no son convertidos a texto.

Los siguientes ejemplos demuestran los elementos previamente analizados.

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 por parámetro el nombre de la tabla a eliminar

EXECUTE 'DROP TABLE IF EXISTS ' || $1 || ' CASCADE';

-- Consulta dinámica que actualiza un campo de la tabla products, recibiendo por
-- parámetros la columna a actualizar, el nuevo valor y su identificador

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 lo 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 que permiten la ejecución de comandos basados en ciertas condiciones como se muestra en los ejemplos del 30 al 34.

Tabla 1: Estructuras condicionales implementadas por PL/pgSQL

Tipo	Sintaxis	Uso

IF-THEN	**IF** *expresión_booleana* **THEN**
	sentencias;
	END IF;
		Forma más simple del IF, las sentencias son ejecutadas si la condición es verdadera.

IF-THEN-ELSE	**IF** *expresión_booleana* **THEN**
	sentencias;
	ELSE
	sentencias;
	END IF;
		Añade al tipo anterior la cláusula ELSE para especificar sentencias alternativas a ejecutar cuando no se cumpla la condición definida.

IF-THEN-ELSIF	**IF** *expresión_booleana* **THEN**
	sentencias;
	[ELSIF *expresión_booleana* **THEN**
	sentencias;
	…]
	[**ELSE**
	sentencias;]
	END IF;
		Empleada cuando hay más de 2 alternativas a evaluar, 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** *expresión* [,*expresión* [...]] **THEN** *sentencias*; [**WHEN** *expresión* [,*expresión* [..]] **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** *expresión_booleana* **THEN** *sentencias*; [**WHEN** *expresión_booleana* **THEN** *sentencias*; ...] [**ELSE** *sentencias*;] **END CASE**;	Permite la ejecución de condicionales basadas en una expresión booleana. Cada expresión 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 en la estructura; al igual que en la anterior, de no haber una expresión evaluada de verdadera se ejecuta la cláusula ELSE de existir.

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

```
-- Insertar un nuevo producto si tiene un precio superior a los 25 pesos
IF price > 25.0 THEN
        INSERT INTO products VALUES ($1, $2, $3, $4, $5, $6, $7);
END IF;
```

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

```
-- Insertar un nuevo producto si tiene un precio superior a los 25 pesos si menor es
-- mostrar una notificación especificando que el producto es muy barato
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;
```

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

```
-- Insertar un nuevo producto si tiene un precio entre los 25 y los 50 pesos, si es menor
-- muestra una notificación indicando que el producto es muy barato, en caso contrario
-- que es muy caro
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;
```

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

```
-- Evaluar si el producto es de categoría infantil (2, 3) u otra
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

```
-- Evaluar si el producto es de categoría infantil (2, 3), drama (5, 7, 8) u otra
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, este tipo de sentencias son muy 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 pues con ambas puede lograrse lo mismo. 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 y de control utilizadas en el ejemplo 35.

Tabla 2 Estructuras iterativas implementadas en 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 LOOPs anidados.
EXIT	**EXIT** [*etiqueta*] [**WHEN** *expresión_booleana*];	Termina la ejecución de un LOOP o un bloque; en caso de especificarse la etiqueta termina el ciclo o bloque etiquetado, siendo la siguiente sentencia la especificada después del END asociado a la estructura

		terminada; de no hacerse termina el LOOP más cercano siendo la siguiente sentencia el END LOOP. De especificarse la cláusula WHEN el EXIT ocurre de cumplirse la condición.
CONTINUE	**CONTINUE** [*etiqueta*] [**WHEN** *expresión_booleana*];	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 sólo si la condición es verdadera, en caso contrario el control pasa a la sentencia después del CONTINUE.
WHILE	[<<*etiqueta*>>] **WHILE** *expresión_booleana* **LOOP** *sentencias*; **END LOOP** [*etiqueta*];	Repite una secuencia de sentencias mientras la condición evaluada sea verdadera, la expresión booleana es chequeada antes de cada entrada en el ciclo LOOP.
FOR	[<<*etiqueta*>>] **FOR** *nombre* **IN** [**REVERSE**] *expresión*.. *expresión* [**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 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.

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

```
-- Cambie el precio de un producto pasado por parámetro, duplicándolo el total de veces
-- pasado como segundo parámetro, retornar el precio final haciendo uso del 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)
-- Cambie el precio de un producto pasado por parámetro, duplicándolo el total de veces
-- pasado como segundo parámetro, retornar el precio final haciendo uso del WHILE
CREATE 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;
```

El uso del FOR puede analizarse en el Ejemplo 37.

Como se muestra en los ejemplos 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.

3.6 Retorno de valores

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

La cláusula RETURN es 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 en una función (excepto en los casos mencionados anteriormente), ya que genera un error en tiempo de ejecución.
- Lo que se devuelve debe tener el mismo tipo de datos que el declarado en la cláusula RETURNS en el encabezado de la función.

El ejemplo 36 muestra su empleo en el cuerpo de una función.

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

```
-- Función que dado el identificador de un producto devuelve su título, empleo de la
-- cláusula RETURN para devolver un dato escalar
CREATE FUNCTION devolver_producto(integer) RETURNS varchar AS
$$
DECLARE
        prod varchar;
BEGIN
        SELECT title INTO prod FROM products WHERE prod_id = $1;
        RETURN prod;
END;
$$ LANGUAGE plpgsql;
-- La misma función anterior pero empleando parámetros de salida, note que en este caso -- la cláusula RETURN no necesita tener una expresión asociada
CREATE FUNCTION devolver_producto(integer, OUT varchar) RETURNS varchar AS
$$
BEGIN
        SELECT title INTO $2 FROM products WHERE prod_id = $1;
        RETURN;
END;
$$ LANGUAGE plpgsql;
-- Función que devuelve dado el identificador de un producto todos sus datos, empleo de
```

```
-- la cláusula RETURN para devolver un dato compuesto, en este ejemplo hay que
-- garantizar que el resultado devuelva una sola tupla, en caso contrario la función
-- devuelve la primera del resultado

CREATE FUNCTION datos_producto(integer) RETURNS products AS

$$

DECLARE

        prod products;

BEGIN

        SELECT * INTO prod FROM products WHERE prod_id = $1;

        RETURN prod;

END;

$$ LANGUAGE plpgsql;

-- Función que devuelve dado el identificador de un producto su nombre, empleo de la
-- cláusula RETURN para devolver de un dato compuesto un elemento

CREATE FUNCTION datos_producto_titulo(integer) RETURNS varchar AS

$$

DECLARE

        prod RECORD;

BEGIN

        SELECT * INTO prod FROM products WHERE prod_id = $1;

        RETURN prod.title;

END;

$$ LANGUAGE plpgsql;

-- La misma función anterior pero utilizando un tipo de dato compuesto que contiene
-- solamente el nombre y el precio del producto

CREATE TYPE mini_prod AS (nombre varchar, precio numeric);

CREATE FUNCTION datos_producto_mini_prod(integer) RETURNS mini_prod AS

$$
```

```
DECLARE
        prod mini_prod;
BEGIN
        SELECT * INTO prod FROM products WHERE prod_id = $1;
        RETURN mini_prod;
END;
$$ LANGUAGE plpgsql;
```

La cláusula RETURN NEXT o RETURN QUERY es empleada cuando la función devuelve un conjunto de datos. Tiene 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 es declarada 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.
- Ambos pueden ser usados en una misma función, concatenándose ambos resultados.
- No culminan la ejecución de la función, simplemente añaden cero o más filas al resultado de la función, puede emplearse un RETURN sin argumento para salir de la función.
- De declararse parámetros de salida se puede especificar el RETURN NEXT sin una expresión, en cuyo caso la función debe declararse para que retorne SETOF record.

Ejemplo 37: Empleo de RETURN NEXT|QUERY para devolver un conjunto de valores resultante de una consulta

```
-- Función que devuelve todos los productos registrados en la base de datos que su
-- identificador sea mayor que el pasado por parámetro, note el empleo de un FOR para
-- iterar por el resultado de la consulta
CREATE FUNCTION datos_producto_setof(integer) RETURNS SETOF products AS
$$
DECLARE
        resultado products;
```

```
BEGIN
        FOR resultado IN SELECT * FROM products where prod_id > $1 LOOP
                RETURN NEXT resultado;
        END LOOP;
        RETURN; -- Opcional
END;
$$ LANGUAGE plpgsql;
-- La misma función pero empleando el RETURN QUERY
CREATE FUNCTION datos_producto_setof(integer) RETURNS SETOF products AS
$$
BEGIN
        RETURN QUERY SELECT * FROM products where prod_id > $1;
        RETURN; -- Opcional
END;
$$ LANGUAGE plpgsql;
-- La misma función pero empleando RETURN QUERY y devolviendo RECORD. Para
-- ejecutarla se debe especificar qué estructura debe tener el resultado de la forma
-- SELECT * FROM datos_producto_record(1000) AS (prod_id int, category integer, title
-- character varying(50), actor character varying(50), price numeric(12,2), special smallint,
-- common_prod_id integer)
CREATE FUNCTION datos_producto_record(integer) RETURNS SETOF record AS
$$
BEGIN
        RETURN QUERY SELECT * FROM products where prod_id > $1;
        RETURN; -- Opcional
END;
$$ LANGUAGE plpgsql;
```

3.7 Mensajes

En ejemplos analizados previamente se ha hecho uso de mensajes utilizando la cláusula RAISE con la opción NOTICE. Además de esta opción, la cláusula RAISE permite las opciones DEBUG, LOG, INFO, WARNING y EXCEPTION, esta última utilizada por defecto:

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

Para mayor detalles sobre las opciones de RAISE puede analizar en la Documentación Oficial la sección Errors and Messages

El ejemplo 38 muestra casos donde es utilizado RAISE con varios de los niveles de mensajes que permite.

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

```
-- Función que devuelve dado el identificador de un producto su título, empleo de la
-- cláusula RETURN para devolver un dato escalar, uso de FOUND y EXCEPTION para
-- determinar si lo encontró
CREATE FUNCTION producto(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 la cláusula RETURN para devolver un dato escalar, uso de FOUND, LOG y
-- WARNING para determinar si lo encontró y registrar el error en el log de PostgreSQL
CREATE OR REPLACE FUNCTION producto(integer) RETURNS varchar AS
$$
DECLARE
        prod varchar;
BEGIN
        SELECT title INTO prod FROM products WHERE prod_id = $1;
        IF not found THEN
                RAISE LOG 'Producto % no encontrado', $1;
                RAISE WARNING 'Producto % no encontrado, es posible que no se haya
                insertado aún', $1;
        END IF;
        RETURN prod;
END;
$$ LANGUAGE plpgsql;
-- Invocación de la función
dell=# SELECT producto(1000000);
WARNING: Producto 1000000 no encontrado, es posible que no se haya insertado aún
 producto
----------
(1 fila)
```

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

- Una asignación de un SELECT INTO que haya arrojado un resultado efectivo almacenado en la variable.

- Las operaciones UPDATE, INSERT y 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.

Existen otros casos donde se activa la variable FOUND, puede ver la sección Obtaining the Result Status de la Documentación Oficial

---

También puede utilizarse un bloque de excepciones para hacer tratamiento de las mismas como se muestra en el ejemplo 39.

Ejemplo 39: Bloque de excepciones

```
-- Función que valida si una consulta pasada por parámetro se puede ejecutar en la base
-- de datos
CREATE OR REPLACE 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
```

3.8 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 se resuelve un problema en que haya que ejecutar 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 automáticamente 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 registro nuevo 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 restricciones 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 el respectivo disparador que llame a dicha función; esto posibilita a varios disparadores utilizar la misma función disparadora 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 40 muestra una función disparadora simple que notifica que ha sido invocado un disparador.

Ejemplo 40: Implementación de una función disparadora

```
-- Función disparadora que notifica que ha sido invocado un disparador
CREATE OR REPLACE 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, la sintaxis ampliada se puede ver en la Documentación Oficial en la sección *SQL Commands*.

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.

Un ejemplo de disparador para la invocación de la función definida anteriormente puede ser el mostrado en el ejemplo 41.

Ejemplo 41: Disparador que invoca la función disparadora del ejemplo 40

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

El ejemplo 41 especifica que la función disparadora previamente definida se ejecutará después de haberse insertado un registro en la tabla *categories*.

En el ejemplo 42 se define un disparador que invocará la función disparadora cuando ocurra más de un evento, a diferencia del ejemplo anterior que sólo la invocaba cuando se realizaba una inserción.

Ejemplo 42: Disparador que invoca la función disparadora cuando se realiza una inserción, actualización o eliminación sobre la tabla categories

```
CREATE TRIGGER trigger_dos
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 está trabajando cuando se invocó el disparador.
- NEW: variable de tipo RECORD que almacena los nuevos valores de la fila que se está insertando o modificando.
- OLD: variable de tipo RECORD que almacena los valores antiguos de la fila que se está modificando o eliminando.

Existen otras variables especiales que se pueden analizar en detalle en la sección Triggers on data changes de la Documentación Oficial

FOR EACH en la definición del disparador

Una vez 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 43 muestra una consulta que ejecuta el disparador definido sobre la tabla *categories* en el ejemplo anterior.

Ejemplo 43: Ejecución de una consulta que activa el disparador creado sobre la tabla categories

```
-- Ejecución de una consulta de actualización 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 44 y se ejecuta la misma sentencia de actualización, se mostraría otro mensaje indicando la ejecución una sola vez.

Ejemplo 44: Definición de la función disparadora y su disparador asociado que se dispara cuando se realiza alguna modificación sobre la tabla categories

```
-- Función disparadora que notifica que ha sido invocado un disparador para modificación
CREATE OR REPLACE FUNCTION trigger_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 anterior una vez realizada una actualización o
-- eliminación sobre la tabla categories
CREATE TRIGGER trigger_modificacion
AFTER UPDATE or DELETE
ON categories
FOR EACH STATEMENT
EXECUTE PROCEDURE trigger_para_modificacion();
-- Invocación del disparador una vez ejecutada una actualización sobre la tabla categories
dell=# UPDATE categories SET categoryname=upper(categoryname) WHERE category >=
```

```
        15;

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 afecte al menos una fila. El ejemplo 45 demuestra lo anterior.

Ejemplo 45: Ejecución de trigger_modificacion en lugar de trigger_uno debido a que no se actualiza ninguna tupla sobre la tabla categories con la consulta ejecutada

```
-- Invocación de trigger_modificacion una vez ejecutada una actualización sobre la tabla
-- categories

dell=# UPDATE categories SET categoryname=upper(categoryname) WHERE category >=
        15;

NOTICE: Un disparador se ha invocado para modificación

UPDATE 0
```

Note que en este ejemplo el disparador que se dispara es trigger_modificacion en lugar de trigger_uno porque el primero utiliza la sentencia FOR EACH STATEMENT en lugar de FOR EACH ROW.

El evento TRUNCATE solo puede ser utilizado con la sentencia FOR EACH STATEMENT

El 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 46 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 46: Disparador que permite modificar un nuevo registro antes de insertarlo en la base de datos

```
-- Función disparadora que pone en mayúscula la primera letra del nombre
CREATE OR REPLACE FUNCTION mayuscula_nombre() RETURNS trigger AS
$$
DECLARE
        resultado text;
BEGIN
        -- Validando que el nombre comience con mayúscula
        IF ascii(substring(NEW.firstname from 1 for 1) ) >= 65 AND
        ascii(substring(NEW.firstname from 1 for 1)) <= 90 THEN
                RAISE NOTICE 'Formato correcto';
                RETURN NEW;
        ELSE
        -- Si no comienza con mayúscula se arregla antes de insertarlo
                resultado:= upper(substring(NEW.firstname from 1 for
                1)) || substring(NEW.firstname from 2 for length(NEW.firstname));
                RAISE NOTICE 'Formato incorrecto. % es el correcto', resultado;
        -- Se asigna el nuevo valor con la corrección a la variable especial NEW
                NEW.firstname := resultado;
        -- Se retorna el nuevo valor
                RETURN NEW;
        END IF;
END;
$$ LANGUAGE plpgsql;
-- Definición del disparador
```

```
CREATE TRIGGER trigger_modificar_insercion
BEFORE INSERT
ON customers
FOR EACH ROW
EXECUTE PROCEDURE mayuscula_nombre();
--Ejecución de una inserción sobre la tabla customers con la primera inicial del nombre en  -- mayúscula
dell=# INSERT INTO customers(firstname, lastname, address1, address2, city, state, country, region, creditcardtype, creditcard, creditcardexpiration, username, password) VALUES ('Anthony', 'Sotolongo', 'dir1', 'dir2', 'Cienfuegos', 'Cienfuegos', 'Cuba', 2, 1, 'una tarjeta crédito', 'algo', 'asotolongo', 'mipasss');
NOTICE: Formato correcto
UPDATE 0
INSERT 0 1
--Ejecución de una inserción sobre la tabla customers sin la primera inicial del nombre en  -- mayúscula
dell=# INSERT INTO customers(firstname, lastname, address1, address2, city, state, country, region, creditcardtype, creditcard, creditcardexpiration, username, password) VALUES ('luis', 'Sotolongo', 'dir', 'dir2', 'Cienfuegos', 'Cienfuegos', 'Cuba', 2, 1, 'una tarjeta crédito', 'algo', 'lsotolongo', 'mipasss');
NOTICE: Formato incorrecto. Luis es el correcto
INSERT 0 1
```

Este comportamiento puede ser utilizado para evitar que se eliminen datos de alguna tabla específica. Por ejemplo, de ser necesario garantizar que no se eliminen datos de la tabla *orders*, pudiera desarrollarse un disparador en el que se defina que antes de eliminar un registro de esa tabla se ejecute determinada función disparadora que retorne NULL. El ejemplo 47 muestra una implementación para ello.

Ejemplo 47: Disparador que no permite eliminar de una tabla

```
-- Función disparadora que impide eliminar registros de una tabla
CREATE OR REPLACE 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 trigger_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 registro de cambios en la base de datos, algo así como un *log* 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 48 muestra cómo se almacenan en una tabla llamada *registro* los cambios de modificación o eliminación realizados sobre la tabla *customers,* donde, además, se almacenará la operación y la fecha en que fue realizada.

Ejemplo 48: Forma de realizar auditorías sobre la tabla customers

```
-- Tabla registro donde se almacenarán las operaciones realizadas sobre customers
CREATE TABLE registro (operacion text, fecha timestamp, antiguo text, nuevo text);
-- Nota: pueden utilizarse tipos de datos con mejores descripciones para el antiguo y nuevo valor como JSON y HSTORE
```

```
--Función disparadora para registrar las operaciones
CREATE OR REPLACE FUNCTION registro_trigger() RETURNS trigger AS
$$
BEGIN
        IF TG_OP = 'UPDATE' THEN
                RAISE NOTICE 'Operación UPDATE';
                INSERT INTO registro VALUES (TG_OP, now(), OLD::text, NEW::text);
        END IF;
        IF TG_OP = 'DELETE' THEN
                RAISE NOTICE 'Operación DELETE';
        INSERT INTO registro VALUES (TG_OP, now(),OLD::text,'');
        END IF;
        RETURN null;
END;
$$ LANGUAGE plpgsql;
--Disparador
CREATE TRIGGER trigger_registro
AFTER DELETE or UPDATE
ON customers
FOR EACH ROW
EXECUTE PROCEDURE registro_trigger();
-- Ejemplo de operación DELETE
dell=# DELETE FROM customers WHERE customerid = 10000;
NOTICE: Operación DELETE
DELETE 1
-- Ejemplo de operación UPDATE
dell=# UPDATE customers SET firstname = lower(firstname) WHERE customerid =
```

```
12000;
NOTICE: Operación UPDATE
UPDATE 1
```

Disparadores por columnas y condicionales

Desde la versión 9.0 de PostgreSQL se pueden definir disparadores por columnas sobre sentencias UPDATE y definir alguna condición para que se ejecute el disparador. Esto puede acarrear ventajas de rendimiento al no ser necesario programar en la función disparadora lógica de negocio para que se realice determinada acción; significa, que si se establecen estas condiciones en la definición del disparador no es necesaria 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 un disparador 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 de la tabla *products* se almacene en la tabla *registro*; el ejemplo 49 muestra una implementación para ello.

Ejemplo 49: Función disparadora y disparador por columnas para controlar las actualizaciones sobre la columna price de la tabla products

```
-- Función disparadora para registrar las actualizaciones sobre la columna precio
CREATE OR REPLACE 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 trigger_registro_update
AFTER UPDATE OF price
ON products
FOR EACH ROW
EXECUTE PROCEDURE registrar_update()
```

Los disparadores condicionales permiten especificar condiciones en la definición del disparador, pudiendo escribirse cualquier expresión con resultado booleano 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 50.

Ejemplo 50: Definición de un disparador condicional para chequear que se actualice el precio

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

Note que en este caso 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.

Ejemplo 51: Disparador condicional para chequear que actor comienza con mayúsculas

```
CREATE TRIGGER trigger_registrar_update
```

```
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 de PostgreSQL se permite realizar disparadores sobre las vistas siempre y cuando el disparador se defina haciendo uso de:

- 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 actividad de DDL 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 52 describe cómo hacer un disparador sobre la vista inventario_de_productos para que sea actualizada.

Ejemplo 52: Función disparadora y disparadores necesarios para actualizar una vista

```
-- Definición de la vista inventario_de_productos
CREATE VIEW inventario_de_productos AS (
        SELECT inventory.quan_in_stock, inventory.sales, products.title,
        inventory.prod_id
        FROM inventory, products
        WHERE products.prod_id=inventory.prod_id);
-- Definición de la función disparadora
CREATE OR REPLACE 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 las ventas y el 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 el 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 trigger_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('Braveheart') WHERE
        prod_id = 12;
NOTICE: Disparador sobre la vista inventario_de_productos
NOTICE: Actualizando el nombre del producto: ACADEMY ALASKA por BRAVEHEART
UPDATE 1
```

Disparadores sobre eventos

Los disparadores sobre los eventos DDL son agregados a PostgreSQL desde la versión 9.3, los mismos permiten ejecutar alguna rutina de función cuando se ejecuta algún 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: ddl_command_start, ddl_command_end y sql_drop (detalles *en Overview of Event Trigger Behavior* de la Documentación Oficial).
- WHEN: filtro para que se ejecute o no la función disparadora, *variable_filtro* debe ser TAG y *valor_filtro* debe ser cualquiera de los comando DDL listados en la sección *Event Trigger Firing Matrix* de la Documentación, por ejemplo CREATE TABLE, DROP TABLE, etc.

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

- TG_EVENT: variable de tipo texto con el evento.
- TG_TAG: variable de tipo texto con 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 53.

Ejemplo 53: 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_evento (
        evento text,
        fecha timestamp);
-- Definición de la función disparadora
CREATE OR REPLACE FUNCTION registrar_evento() RETURNS event_trigger AS
$$
```

```
BEGIN
        RAISE NOTICE 'Evento: % , Horario: %', TG_TAG, now();
        INSERT INTO registro_evento VALUES (TG_TAG, now());
END;
$$ LANGUAGE plpgsql;
-- Definición del disparador
CREATE EVENT TRIGGER trigger_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: 2014-01-08 18:02:49.941-0
```

3.9 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 las variables, estructuras de control y condicionales, paso de mensajes, disparadores, etc. Además, permite 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 de la función; su cuerpo está estructurado por 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).

3.10 Para hacer con PL/pgSQL

1. Implemente una función que permita la inserción datos en la tabla *categories*, la misma debe mostrar un mensaje de error si la categoría que se desea insertar ya existe.

2. Desarrollo 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. Elabore una función que dada una fecha devuelva todos los datos de órdenes que se realizaron antes de ella, de no haber, muestre un mensaje especificándolo.
4. Obtenga una función que devuelva el nombre de los productos que en el inventario ya no queden en el almacén (inventory.quan_in_stock - inventory.sales=0), si no hay productos en esta situación emita un mensaje indicándolo.
5. Cree una función que dada una categoría de un producto calcule el promedio del precio si la categoría es "Games" o "Music", de lo contrario calcule el valor mínimo del precio.
6. Implemente 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 tuplas a actualizar.
7. Elabore una función que permita mostrar todos los datos de una tabla pasada por parámetro dinámicamente.
8. Dado el identificador de una categoría, se 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. Realice un mecanismo que registre en los *logs* de PostgreSQL los nuevos productos que se inserten de ser el precio superior a las 200 unidades.
10. Implemente un mecanismo que registre en una tabla llamada eliminados [create table eliminados (productos text)] los productos eliminados.
11. Dada una vista que muestre la información de la consulta siguiente:

```
SELECT products.title, categories.categoryname
FROM public.categories, public.products
WHERE products.category = categories.category;
```

    a. Realice un mecanismo que si se modifica un título de un producto de esa vista el mismo se propague a la tabla correspondiente.

4. 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 la operaciones con los datos almacenados en la 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 (para consultar sobre el tema puede remitirse en la Documentación Oficial a la sección *C-Language Function*). En este capítulo se detallarán sobre lenguajes procedurales de desconfianza, útiles para hacer dichas actividades, que aunque 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, algo así como escribir un código en *Python* dentro de una función. 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* para el gestor de bases de datos 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.

Para la instalación de PL/Python a partir de 9.1 en adelante que se creó en PostgreSQL el mecanismo de extensiones, este lenguaje se puede instalar como una extensión con el comando *create extension plpythonu*. También se puede instalar desde la línea de comandos con createlang plpythonu nombre_basededatos, para versiones previas.

Para el empleo correcto de PL/Python en Debian/Ubuntu el sistema debe tener instalado el paquete postgresql-plpython-9.1 o superior

4.2.1 Escribir funciones en PL/Python

Una función en PL/Python se crea de igual forma que el resto de las funciones en PostgreSQL, haciendo uso del comando CREATE FUNCTION. El cuerpo de la función es código en *Python*. El retorno de la función se realiza con la cláusula clásica de *Python* para retornar valores de una función *return*, también puede ser empleado *yield* y, sino se provee una salida se devuelve *none* que se traduce a PostgreSQL como null.

El ejemplo siguiente muestra es una función para devolver la cadena “Hola Mundo”.

Ejemplo 54: Hola Mundo con PL/Python

```
CREATE FUNCTION holamundo() RETURNS text AS
$$
        return "Hola Mundo"
$$
LANGUAGE plpythonu;
-- Invocación de la función
dell=# SELECT holamundo();
  holamundo
- - - - - - - - - -
Hola Mundo
(1 fila)
```

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

4.2.2 Parámetros de una función en PL/Python

Los parámetros de una función PL/Python se pasan normalmente como en cualquier otro lenguaje procedural y, una vez dentro de la función deben ser llamados por su nombre. El ejemplo 55 muestra esto.

Ejemplo 55: Paso de parámetros en funciones PL/Python

```
CREATE FUNCTION hola(nombre text) RETURNS text AS
$$
        return 'Hola %s' % nombre
$$
LANGUAGE plpythonu;
-- Invocación de la función
dell=# SELECT hola('Anthony');
  hola
- - - - - - - - - -
Hola Anthony
(1 fila)
```

Al igual que en PL/pgSQL pueden definirse parámetros de salida. El ejemplo 56 muestra su empleo.

Ejemplo 56: Empleo de parámetros de salida

```
CREATE OR REPLACE FUNCTION saludo(INOUT nombre text, OUT mayuscula text) AS
$$
        mayúscula = 'HOLA %s' % nombre.upper()
        return ('Hola '+ nombre, mayuscula)
$$
LANGUAGE plpythonu;
-- Invocación de la función
dell=# SELECT saludo('Sandra');
  saludo
- - - - - - - - - -
```

```
Hola Sandra, HOLA SANDRA

(1 fila)
```

Pasando tipos compuestos como parámetros

Los tipos de datos compuestos de PostgreSQL también pueden pasarse como parámetro a PL/Python y son automáticamente convertidos en diccionarios dentro de *Python*. A continuación un ejemplo de cómo se utilizan.

Ejemplo 57: Paso de parámetros como tipo de dato compuesto

```
-- Creación de un tipo de dato compuesto

CREATE TYPE mini_prod AS (nombre varchar, precio numeric);

-- Función que chequea que determinado product comience o no con A

CREATE OR REPLACE 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('Braveheart', 1.10));

  parametros_mi_tipo
- - - - - - - - - - - - - - -
     No comienza con A

(1 fila)
```

Pasando arreglos como parámetros

También se pueden pasar como parámetros arreglos de PostgreSQL los cuales son convertidos a listas o tuplas. El ejemplo 58 muestra cómo emplearlos.

Ejemplo 58: Arreglos pasados por parámetros

```
-- Función que devuelve el primer elemento de un arreglo pasado por parámetro
CREATE OR REPLACE FUNCTION arreglos(a character varying[]) RETURNS character varying AS
$$
        return a[0]
$$
LANGUAGE plpythonu;
-- Invocación de la función
dell=# SELECT arreglos(array['Anthony','Sotolongo']);
 arreglos
----------
 Anthony
(1 fila)
```

4.2.3 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, int	int	int
bigint, oid	long	int
null	none	none

4.2.4 Retorno de valores de una función en PL/Python

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

Devolviendo arreglos

Devolver una lista o tupla en *Python* es similar a un arreglo en PostgreSQL; el ejemplo 59 lo muestra.

Ejemplo 59: Retornando una tupla en Python

```
-- Función que retorna un arreglo
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)
```

Devolviendo tipos compuestos

Devolver 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 usurario; el ejemplo 60 muestra el caso.
- Diccionario: que debe tener la misma la cantidad y tipos de datos que el tipo compuesto definido, además, los nombres de las columnas deben coincidir; el ejemplo 61 muestra el caso.
- Objeto: donde la definición de la clase debe tener los atributos similares al del tipo de datos compuesto por el usuario.

Ejemplo 60: Devolviendo valores de tipo compuesto como una tupla

```
-- Función que retorna un dato compuesto como tupla
CREATE FUNCTION retorna_tipo_lista() RETURNS mini_prod AS
$$
        nombre = 'Una novia para David'
```

```
        precio = 2.80
        return (nombre, precio) # como tupla
$$ LANGUAGE plpythonu;
-- Invocación de la función
dell=# SELECT retorna_tipo_lista();
 retorna_tipo_lista
- - - - - - - - - - - - - - - - - -
 ("Una novia para David", 2.8)
(1 fila)
```

Note que en este y los siguientes ejemplos se emplea el tipo de dato compuesto mini_prod creado en el ejemplo 57.

También puede retornarse como con la forma RETURN [nombre, precio] para hacerlo una lista

Ejemplo 61: Devolviendo valores de tipo compuesto como un diccionario

```
-- Función que retorna un dato compuesto como diccionario
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 ejemplo 62.

- Un generador (*yield*), ver ejemplo 63.
- Un iterador.

Ejemplo 62: Devolviendo conjunto de valores como una lista o tupla

```
-- Función que retorna un conjunto de datos como lista o tupla
CREATE FUNCTION mini_producto_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_producto_conjunto();
  nombre | precio
- - - - - - - - - - - - - - - - - - -
 Hello Hemingway | 4.31
 Hello HemingwayHello Hemingway | 8.62
(2 filas)
```

Ejemplo 63: Devolviendo conjunto de valores como un generador

```
-- Función que retorna un conjunto de datos como un generador
CREATE FUNCTION mini_producto_conjunto_generador() RETURNS SETOF record AS
$$
        lista=[('Clandestinos', 5.00), ('Memorias del subdesarrollo', 5.10), ('Suite   Habana',
            3.90)]
        for producto in lista:
                yield ( producto[0], producto[1] )
$$ LANGUAGE plpythonu;
-- Invocación de la función
dell=# SELECT * FROM mini_producto_conjunto_generador() AS (a text, b numeric);
```

```
 a | b
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
Clandestinos | 5.00
Memorias del subdesarrollo | 5.10
Suite Habana | 3.90
(3 filas)
```

4.2.5 Ejecutando consultas en la función PL/Python

PL/Python permite realizar consultas sobre la base de datos a través de un módulo llamado plpy, importado por defecto. Tiene varias funciones importantes como son:

- plpy.execute(*consulta* [, max-rows]): permite ejecutar una consulta con una cantidad finita de tuplas en el resultado, especificado en el segundo parámetro (opcional). Devuelve un objeto similar a una lista de diccionarios, al que se puede acceder con la forma resultado[0]['micolumna'], para acceder al primer registro y a la columna 'micolumna'. Ver ejemplo 64 para su uso.
- plpy.prepare(consulta [, argtypes]): permite preparar planes de ejecución para determinadas consultas para luego ser ejecutadas por la función execute(plan [, arguments [, max-rows]]). Ver ejemplo 65 para su uso.
- plpy.cursor(*consulta*): añadido a partir de la versión 9.2 del gestor.

Ejemplo 64: Devolviendo valores de la ejecución de una consulta desde PL/Python con plpy.execute

```
-- Función que retorna los 10 primeros productos haciendo uso de plpy.execute
CREATE FUNCTION obtener_productos() RETURNS TABLE (id int, titulo text, precio numeric) AS
$$
        resultado = plpy.execute("SELECT * FROM products LIMIT 10")
        for tupla in resultado:
                yield (tupla['prod_id'], tupla['title'], tupla['price'])
$$
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)
```

Ejemplo 65: 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)
```

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), plpy.error(msg). Para más información sobre

estas funciones puede dirigirse a la sección Utility Functions del capítulo PL/Python - Python Procedural Language de la Documentación Oficial

4.2.6 Mezclando

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.

Ejemplo 66: Guardar en un XML el resultado de una consulta

```
CREATE OR REPLACE 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;
```

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

4.2.7 Realizando disparadores con PL/Python

PL/Python permite escribir funciones disparadoras, para eso cuenta con un variable diccionario TD, que posee varios pares llave/valor útiles para su trabajo; algunos se describen a continuación:

- TD["event"]: almacena como un texto el evento que se está ejecutando (INSERT, UPDATE, DELETE o TRUNCATE).
- TD["when"]: almacena como un texto el momento de ejecución del *trigger* (BEFORE, AFTER o INSTEAD OF).
- TD["new"], ["old"]: almacenan el registro nuevo y viejo respectivamente, en dependencia del evento que se ejecute.
- TD["table_name"]: almacena el nombre de la tabla que dispara el *trigger*.

Se 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 67 muestra un disparador que verifica al insertar si el título es "HOLA MUNDO" y si es así no lo permite insertar.

Ejemplo 67: Disparador en PL/Python

```
-- Función disparadora
CREATE FUNCTION trigger_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 trigger_plpython
BEFORE INSERT OR UPDATE
ON products
FOR EACH ROW
EXECUTE PROCEDURE trigger_plpython();
-- Invocación de la función
dell=# INSERT INTO products(prod_id,category,title,actor,price,special,common_prod_id)
dell=# VALUES (20001, 13, 'HOLA MUNDO', 'Anthony Sotolongo', 12.0, 0, 1000);
NOTICE: Comenzando el disparador
CONTEXT: PL/Python function "trigger_plpython"
NOTICE: Se abortó la inserción por tener el título: HOLA MUNDO
CONTEXT: PL/Python function "trigger_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. Es desarrollado por Joseph E. Conway desde el 2003 y compatible con PostgreSQL desde su versión 7.4. Soporta casi todas las funcionalidades de R desde el gestor. Este lenguaje es orientado específicamente a realizar operaciones estadísticas y el cual es muy potente en esta rama, cuenta con disimiles paquetes (conjunto de funcionalidades) para su trabajo y se utiliza en varias áreas de la informática, medicina, bioinformática, entre otras.

Para la instalación de PL/R a partir de la versión 9.1 en adelante, en la que fue añadido en PostgreSQL el mecanismo de extensiones, se puede emplear el comando *create extension plr*. También se puede instalar desde la línea de comandos con createlang plr nombre_basedatos, para versiones previas puede consultar el sitio oficial del lenguaje (http://www.joeconway.com/plr/).

Se debe tener instalado en los sistemas Debian/Ubuntu el paquete postgresql-9.1-plr o superior

4.3.1 Escribir funciones en PL/R

Una función en PL/R se define igual que las demás funciones en PostgreSQL, con la sentencia CREATE FUNCTION. El cuerpo de la función es código R y tiene la particularidad de que en este

lenguaje las funciones deben nombrarse de forma diferente aunque sus atributos 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 escribirla.

El ejemplo 68 muestra cómo sumar dos valores con PL/R.

Ejemplo 68: 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;
```

4.3.2 Pasando parámetros a una función PL/R

Los parámetros de una función PL/R se pasan normalmente como en cualquier lenguaje procedural y:

- Pueden nombrarse, debiendo ser llamados por dicho nombre dentro de la función.
- De no ser nombrados pueden ser accedidos con *argN*, siendo n el orden que ocupan en la lista de parámetros.

El ejemplo 69 muestra la implementación del ejemplo 68 sin nombrar los parámetros.

Ejemplo 69: 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

En una función PL/R cuando se recibe un arreglo como parámetro automáticamente se convierte a un vector c(...).

La función implementada en el ejemplo 70 calcula la desviación estándar de un arreglo pasado por parámetro.

Ejemplo 70: Cálculo de la desviación estándar desde PL/R

```
CREATE OR REPLACE 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

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 71 muestra cómo hacerlo.

Ejemplo 71: Pasando un tipo de dato compuesto como parámetro

```
-- Crear tipo de dato compuesto
CREATE TYPE mini_prod AS (nombre varchar, precio numeric);
-- Función que recibe un tipo de dato compuesto como parámetro
CREATE OR REPLACE 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('Suite Habana', 0)));

  compuesto
- - - - - - - - - - -
Precio incorrecto

(1 fila)
```

4.3.3 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, float4, float8, cash, numeric	numeric
int, int4	integer
bytea	objetc
otro	caracter

4.3.4 Retornando valores de una función en 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, que debe coincidir con el tipo de dato definido en la declaración de la función.

Devolviendo arreglos

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

Ejemplo 72: Retorno de valores con arreglos desde PL/R

```
-- Función que realiza un resumen estadístico de un arreglo pasado por parámetro
```

```
CREATE OR REPLACE 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)
```

Devolviendo 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, con los atributos en orden respecto a los tipos de datos.

Ejemplo 73: Retorno de valores con tipos de datos compuestos desde PL/R

```
CREATE OR REPLACE 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)
```

Devolviendo conjuntos

Se pueden devolver conjuntos de datos de algún tipo compuesto, TABLE o RECORD. A continuación se muestran dos ejemplos haciendo uso de ambos tipos.

Ejemplo 74: Retorno de conjuntos

```
-- Devolviendo conjuntos haciendo uso de TABLE
CREATE OR REPLACE FUNCTION devolver_varios_table() RETURNS TABLE(a text, b 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();
  a | b
- - - - - - - - - - - - - - - - - -
Fresa y Chocolate | 25.01
La película de Ana | 10.65
Conducta | 60.00
(3 filas)
-- Devolviendo conjuntos haciendo uso de RECORD
CREATE OR REPLACE FUNCTION devolver_varios_record() RETURNS SETOF RECORD AS
$$
        nombres <- c("Fresa y Chocolate", "La película de Ana", "Conducta")
        precios <- c(25.01, 10.65, 60)
        dataframe <- data.frame(nombre = nombres, precio = precios)
        return (dataframe)
```

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

4.3.5 Ejecutando consultas en la función 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 ejemplo 75.
- 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 los argumentos para la consulta preparada.
- pg.spi.cursor_open y pg.spi.cursor_fetch: empleados para el trabajo con cursores.

Ejemplo 75: Retorno del resultado de una consulta desde PL/R usando pg.spi.exec

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

ACADEMY AFFAIR | 14.99

-- More --
```

4.3.6 Mezclando

Como se ha mostrado estos lenguajes de desconfianza son utilizados para hacer rutinas externas al servidor de bases de daos, a continuación se muestra un ejemplo de cómo generar un gráfico de barras en un archivo .png con el resultado de una consulta.

Ejemplo 76: Función que genera una gráfica de barras con el resultado de una consulta en PL/R

```
CREATE FUNCTION barras_simple(nombre text, consulta text, texto text, ejex text[])
RETURNS integer AS

$$

        png(paste(nombre,"png", sep="."))

        resultado <- pg.spi.exec(consulta)

        barplot(as.matrix(resultado),beside=TRUE,main=texto,col=rainbow(length(as.
        matrix(resultado))), names.arg=c(ejex))

        dev.off()

$$

LANGUAGE plr;

-- Invocación de la función

dell=# SELECT * FROM barras_simple('grafica_barras',

dell=# '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',

dell=# 'Cantidad producto x Categoría',
```

```
dell=# array(SELECT categoryname FROM categories ORDER BY categoryname LIMIT 4 )::text[]);
```

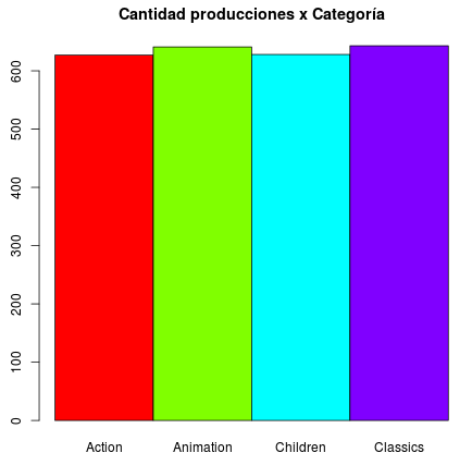

Figura 3: Gráfica de barras generada con el resultado de una consulta en PL/R

4.3.7 Realizando disparadores con PL/R

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

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

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

El retorno del disparador puede ser NULL, una fila en forma de data.frame(pg.tg.new, pg.tg.old) o alguno que contenga las mismas columnas de la tabla que lo invocó. Cuando se retorna NULL significa que el resultado de la operación realizada va a ser ignorado.

El ejemplo 77 muestra el empleo de disparadores desde PL/R.

Ejemplo 77: Utilizando disparadores en PL/R

```
-- Función disparadora
CREATE FUNCTION trigplrfunc() 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 trigplrfunc();
-- 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 Para hacer con PL/Python y PL/R

1. Desarrolle una función en PL/Python que devuelva el monto total(price* quantity) del producto donde el actor es "VIVIEN COOPER".
2. Construya una función que permita devolver los 100 productos más caros.
3. Elabore una función en PL/Python que permita exportar los datos del ejercicio 2 a un archivo CSV.
4. Realice una función en PL/Python para devolver los productos que su título comience con un caracter pasado por parámetro, prepare dicha consulta antes de ejecutarla.
5. Conciba una función en PL/Python que permita exportar los datos del ejercicio 4 a un archivo Excel.
6. Implemente un mecanismo basado en disparadores en PL/Python que permita:
    a. Llevar un registro de las categorías eliminadas.
    b. Controlar que si se agrega o modifica el precio de una producción y este precio es 0, le envíe un correo al administrador del sistema (suponga un correo admin@dellstore.com) notificando la situación.
7. Logre una función en PL/R que devuelva los productos que su precio sea mayor a uno pasado por parámetro, prepare dicha consulta antes de ejecutarla.
8. Confeccione funciones en PL/R que permitan obtener gráficos de:
    a. Pastel con el resultado de la consulta del ejercicio 7.
    b. Histograma de todos los productos del actor "PENELOPE GUINESS".
9. Realice una función en PL/R que permita calcular la mediana de todos los productos realizados por el actor "JUDY REYNOLDS".

4.5 Resumen

Los llamados lenguajes de “desconfianza” permiten escribir lógica de negocio en el servidor de bases de datos PostgreSQL, escritas 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 de la función y; su cuerpo estará compuesto por las características del lenguaje

especificado. Se realizó una homologación con los tipos de datos de PostgreSQL, así como la ejemplificación de la implementación de disparadores.

Se debe tener cuidado en el uso de estos lenguajes pues desde ellos se pueden acceder a recursos del servidor más allá de los datos almacenados. Se recomiendan utilizar en entornos controlados y para actividades que no se puedan realizar desde los lenguajes nativos como lo son el SQL y el PL/pgSQL. El rendimiento de los mismos no suele ser el mejor.

RESUMEN

En *PL/pgSQL y otros lenguajes procedurales en PostgreSQL* se destacan las ventajas de la programación del 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 hacerlo, mediante Funciones en SQL y Funciones en Lenguajes Procedurales. En los capítulos propuestos se analizaron en detalle la forma de implementar este tipo de funciones y su uso en algunos escenarios, abarcando con varios 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 2, 3 y 4, se debe haber alcanzado una habilidad básica para la programación del lado del servidor PostgreSQL con características disponibles hasta su versión 9.3.

BIBLIOGRAFÍA

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. 2013. *PostgreSQL 9.3.0 Documentation.* California : s.n., 2013.