PostgreSQL

Procedimientos Almacenados

Procedimientos Almacenados

SQL

SQL es un lenguaje de base de datos normalizado, y ampliamente utilizado por los DBMS relacionales. Es un lenguaje declarativo, en donde las instrucciones indican lo que se quiere, pero no la forma de cómo conseguirlo (como ocurre con los lenguajes procedimentales).

SELECT nombre_e FROM estudiantes
WHERE carrera = 'Ingeniría de Sistemas'
ORDER BY nombre;

Devuelve el nombre de los estudiantes de Ingeniería de Sistemas ordenados por nombre.

Como es un lenguaje declarativo, sus instrucciones se escriben como si fueran frases, donde se indica el resultado que se desea obtener.

Procedimientos Almacenados

El manejo del lenguaje SQL mediante sus instrucciones LDD y LMD tiene limitaciones, ya que no permite operaciones como control de flujo (estructuras de decisión), ciclos iterativos (estructuras repetitivas), o simplemente almacenar información en variables para usarlas posteriormente. Esto limita la potencialidad del lenguaje ya que no se cuentan con estas ventajas.

PostgeSQL (y muchos de los DBMS), permite el desarrollo de la programación lógica mediante funciones definidas por el usuario (procedimientos almacenados en otros DBMS). Las funciones definidas por el usuario se pueden clasificar en 4 tipos:

- Función en SQL: sólo con operaciones SQL
- Función en lenguaje procedimental: con PL/pgSQL o PL/Python
- Funciones internas: funciones predefinidas enlazadas en el servidor
- Funciones en C: funciones en C o C++ que se pueden cargar

Procedimientos Almacenados

¿Qué son los procedimientos almacenados?

Son programa que se almacena físicamente en la base de datos. Una de las principales ventajas es que su ejecución se realiza directamente en el servidor de bases de datos, teniendo un acceso directo a los datos y sólo necesita enviar el resultado al emisor de la petición.

En PostgreSQL un procedimiento almacenado se puede escribir en varios lenguajes de programación:

1. **PL/pgSQL**
2. PL/Perl
3. PL/Tcl
4. PL/Python

El único de ellos disponible en la instalación normal de PostgreSQL es PL/pgSQL

Procedimientos Almacenados

Usos de los procedimientos almacenados

- Transacciones que incluyen varias operaciones
- Auditoría de datos
  - pgAudit (https://www.pgaudit.org)
  - Triggers
- Consumo de características de otros lenguajes de programación
- Importación y exportación de datos

Procedimientos Almacenados

Como módulos adicionales, existen otros lenguajes como:

1. PL/Java
2. PL/PHP
3. PL/R
4. PL/Ruby
5. PL/Sheme
6. Pl/Sh

Pero deben descargarse e instalarse por separado

Objetivos de PL/pgSQL

Los principales objetivos cuando se creó este lenguaje fueron:

1. Poder ser utilizado para crear funciones y disparadores (triggers)
2. Añadir estructuras de control al lenguaje SQL
3. Poder realizar cálculos complejos
4. Heredar todos los tipos, funciones y operadores definidos por el usuario
5. Poder ser definido como un lenguaje “confiable”
6. Fácil de usar

PL/pgSQL

PL/pgSQL es un lenguaje estructurado en bloques. Al menos debe existir un bloque principal para el procedimiento almacenado, y dentro de él se pueden definir sub-bloques.

Un bloque se define de la siguiente forma:

```
[ << etiqueta >> ]
[ DECLARE
        declaraciones de variables ]
BEGIN
        codigo
END [ etiqueta ];
```

Definición de una función PL/pgSQL

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

PL/pgSQL

Opciones más importantes:

- argmodo . Puede ser IN, OUT o INOUT. Por defecto se usa IN, en caso de no definirse
- argtipo . Los tipos que podemos utilizar son todos los disponibles en PostgreSQL y todos los definidos por el usuario
- Declaraciones de variables. Las declaraciones de variables se pueden realizar de la siguiente manera
  ($n = orden de declaración del argumento.)

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

PL/pgSQL - Código

Un ejemplo de una declaración de una función

```
CREATE OR REPLACE FUNCTION ejemplo() RETURNS integer AS $$
BEGIN
        RETURN 104;
END;
$$ LANGUAGE plpgsql;
```

La función se usa de la siguiente forma

```
SELECT ejemplo();
```

PL/pgSQL - Código

Usando un argumento

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

La función se usa de la siguiente forma

```
SELECT ejemplo(205);
```

PL/pgSQL - Código

También se pudo haber escrito de la siguiente forma

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

```
CREATE OR REPLACE FUNCTION ejemplo(integer) RETURNS integer AS
$$
DECLARE numero ALIAS for $1;
BEGIN
        RETURN numero;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL - Código

Usando dos argumentos y declarando algunas variables

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

PL/pgSQL - Código

Declaración de una función con parámetros de salida

```
CREATE OR REPLACE FUNCTION ver_estudiante(cod int, OUT ce int,
   OUT nom varchar, OUT car varchar) AS $$
BEGIN
   ce := (select cod_e from estudiantes where cod_e=cod);
   nom := (select nom_e from estudiantes where cod_e=cod);
   car := (select nom_carr from estudiantes natural join carreras
                where cod_e=cod);
END;
$$ LANGUAGE plpgsql;
```

RETURNS es omitido, sería redundante con los parámetros de salida.

PL/pgSQL - Código

Usando sentencias IF … THEN

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

PL/pgSQL - Código

Retornando conjunto de registros

```
CREATE FUNCTION permutacion(INOUT a int,
INOUT b int,INOUT c int) RETURNS SETOF RECORD AS $$
BEGIN
RETURN NEXT;
        SELECT b,c INTO c,b; RETURN NEXT;
        SELECT a,b INTO b,a; RETURN NEXT;
        SELECT b,c INTO c,b; RETURN NEXT;
        SELECT a,b INTO b,a; RETURN NEXT;
        SELECT b,c INTO c,b; RETURN NEXT;
END;
$$ LANGUAGE plpgsql;
```

```
select permutacion(1,2,3);
```

PL/pgSQL - Código

```
CREATE TABLE test001 (id integer ,value text);

INSERT INTO test001 VALUES (1,'a'), (1,'b'),(1,'c');
```

```
CREATE OR REPLACE FUNCTION show_data() RETURNS SETOF test001
AS $$
DECLARE
        sql_result test001%rowtype;
BEGIN
        FOR sql_result in EXECUTE 'SELECT * from test001' LOOP
                RETURN NEXT sql_result;
        END LOOP;
END;
$$ LANGUAGE plpgsql
```

```
Select * from show_data();
```

PL/pgSQL - Código

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT); INSERT
INTO foo VALUES (1, 2, 'three'), (4, 5, 'six'),(-1,9,’ten’);
```

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

```
Select * from getAllFoo();
```

PL/pgSQL - Código

Función con estructura desconocida

```
CREATE OR REPLACE FUNCTION consulta(query TEXT)
RETURNS SETOF RECORD
AS $$
DECLARE
        registro RECORD;
BEGIN
        FOR registro IN EXECUTE query LOOP
        RETURN NEXT registro;
END LOOP ;
END;
$$ LANGUAGE plpgsql;
```

```
select * from consulta('select cod_e,nom_e,nom_carr from
        estudiantes natural join carreras') as
        ("ce" int,"ne" varchar,"nc" varchar);
```

PL/pgSQL - Código

Estructura repetitiva con arreglos

```
CREATE FUNCTION findmax(int[]) RETURNS int8 AS $$
DECLARE  max int8 := 0;
         x int;
BEGIN    FOREACH x IN ARRAY $1
         LOOP     IF x > max THEN
                  max := x;
                  END IF;
         END LOOP;
         RETURN max;
END;
$$ LANGUAGE plpgsql;
```

```
select findmax(ARRAY[1,2,3,4,5, -1]);
```

PL/pgSQL - Código

```
CREATE OR REPLACE FUNCTION fibonacci_seq(num integer)
RETURNS SETOF integer AS $$
DECLARE  a int := 0;
         b int := 1;
BEGIN    IF (num <= 0) THEN RETURN;
         END IF;
         RETURN NEXT a;
         LOOP     EXIT WHEN num <= 1;
                  RETURN NEXT b;
                  num = num - 1;
                  SELECT b, a + b INTO a, b;
         END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
select fibonacci_seq(7);
```

Triggers

Un procedimiento Trigger es creado con la instrucción CREATE FUNCTION declarando la función sin argumentos y un tipo de retorno de trigger y debe retornar NULL o un tipo de datos record/row con la misma estructura de la tabla asociada al trigger.

Ésta función se asocia a una acción sobre una tabla, la cual puede se un INSERT, UPDATE o DELETE y se puede definir como:

1. Que ocurra antes de cualquier INSERT, UPDATE o DELETE
2. Que ocurra después de cualquier INSERT, UPDATE o DELETE
3. Que se ejecute una sola vez por comando SQL (statement-level-trigger)
4. Para que se ejecute por cada línea afectada por un comando SQL (row-level-trigger)

Triggers

La instrucción para definir un trigger en una tabla es

CREATE TRIGGER nombre { BEFORE | AFTER } { INSERT | UPDATE | DELETE [OR ...] } ON tabla [FOR [EACH] { ROW | STATEMENT }] EXECUTE PROCEDURE nombre de función (argumentos)

Por supuesto, la función ya debe estar previamente creada para definir un trigger y asociarlo a una tabla

Triggers

Características y reglas importantes al momento de defiinr un trigger

1. El procedimiento almacenado debe estar previamente creado
2. El procedimiento no puede tener argumentos y debe devolver el tipo “trigger”
3. Un mismo procedimiento puede ser usado por varios trigger en diferentes tablas
4. Los procedimientos usados por triggers que se ejecute una sola vez por comando SQL (statement-level) tiene que devolver siempre NULL
5. Los procedimientos usados por triggers que se ejecute por cada línea afectada por un comando SQL (row-level) puede devolver una sola fila de tabla

Triggers

Características y reglas importantes al momento de defiinr un trigger

6. Procedimientos que se ejecutan una vez por fila afectada por el comando SQL (row-level) y antes de ejecutarse el SQL puede
   - retornar NULL para saltarse la operación en la fila afectada
   - ó, devuelve una fila de tabla
7. Procedimientos que se ejecuten después del SQL, ignora el valor de retorno, así que puede retornar NULL sin problema
8. Todo procedimiento almacenado debe devolver NULL o un valor record con la misma estructura de la tabla sobre la que opera
9. Si una tabla tiene asociados varios triggers para un mismo evento, éstos se ejecutan en orden alfabético por nombre del trigger. Si es de tipo antes / row-level, la fila retornada es la entrada del siguiente. Si uno de ellos retorna NULL, la operación será anulada para la fila afectada

Triggers

Características y reglas importantes al momento de defiinr un trigger

10. Los procedimientos pueden ejecutar sentencias SQL que a su vez activan otros trigger (triggers en cascada). Pese a que no hay límite del uso de trigger anidados, se recomienda evitar usar demasiado la recursión de llamados, sobre todo el llamado infinito

Triggers

Cuando se define un trigger se definen ciertas variables como:

1. NEW de tipo record que contiene la nueva fila de la tabla para las operaciones INSERT/UPDATE en disparadores de tipo row-level. La variable es NULL de tipo statement-level
2. OLD de tipo record y contiene la antigua fila de la tabla para las operaciones UPDATE/DELETE en disparadores de tipo row-level. La variable es NULL de tipo statement-level
3. TG_NAME de tipo name, contiene el nombre del disparador que esta usando la función actual
4. TG_WHEN de tipo text, contiene BEFORE o AFTER dependiendo de cómo el disparador se está usando
5. TG_LEVEL de tipo text, contiene ROW o STATEMENT dependiendo de cómo el disparador se está usando la función

Triggers

Cuando se define un trigger se definen ciertas variables como:

6. TG_OP de tipo text, contiene el valor INSERT, UPDATE o DELETE
7. TG_RELID de tipo oid, el identificador del objeto de la tabla que ha activado el disparador
8. TG_TABLE_NAME de tipo name, nombre de la tabla que activó el disparador
9. TG_TABLE_SCHEMA de tipo name, nombre del schema de la tabla que activó el disparador
10. TG_NARGS de tipo integer, numero de argumentos dados al procedimiento en la sentencia create trigger
11. TG_ARGV[] de tipo text array, argumentos de la sentencia create trigger. El índice inicia en 0

Triggers

```
CREATE TABLE numeros(
  numero bigint NOT NULL,
  cuadrado bigint,
  cubo bigint,
  raiz2 real,
  raiz3 real,
  PRIMARY KEY (numero)
);
```

Triggers

```
CREATE OR REPLACE FUNCTION proteger_datos() RETURNS TRIGGER
AS $proteger_datos$
  DECLARE
  BEGIN
   /* Esta funcion es usada para proteger datos en un tabla
      No se permitira el borrado de filas si la usamos
      en un disparador de tipo BEFORE / row-level  */
   RETURN NULL;
  END;
$proteger_datos$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER proteger_datos BEFORE DELETE
    ON numeros FOR EACH ROW
    EXECUTE PROCEDURE proteger_datos();
```

Triggers

```
CREATE OR REPLACE FUNCTION rellenar_datos() RETURNS TRIGGER AS
$rellenar_datos$
  DECLARE
  BEGIN
   NEW.cuadrado := power(NEW.numero,2);
   NEW.cubo := power(NEW.numero,3);
   NEW.raiz2 := sqrt(NEW.numero);
   NEW.raiz3 := cbrt(NEW.numero);
   RETURN NEW;
  END;
$rellenar_datos$ LANGUAGE plpgsql;
```

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

Triggers

```
SELECT * from numeros;

INSERT INTO numeros (numero) VALUES (2);

INSERT INTO numeros (numero) VALUES (3);

UPDATE numeros SET numero = 4 WHERE numero = 3;

DELETE FROM numeros;

DROP TRIGGER proteger_datos ON numeros;

DROP TRIGGER rellenar_datos ON numeros;
```

Triggers

```
CREATE OR REPLACE FUNCTION proteger_y_rellenar_datos() RETURNS
TRIGGER AS $proteger_y_rellenar_datos$
  DECLARE
  BEGIN
    IF (TG_OP = 'INSERT' OR TG_OP = 'UPDATE' ) THEN
       NEW.cuadrado := power(NEW.numero,2);
       NEW.cubo := power(NEW.numero,3);
       NEW.raiz2 := sqrt(NEW.numero);
       NEW.raiz3 := cbrt(NEW.numero);
       RETURN NEW;
    ELSEIF (TG_OP = 'DELETE') THEN
       RETURN NULL;
    END IF;
  END;
$proteger_y_rellenar_datos$ LANGUAGE plpgsql;
```

Triggers

```
CREATE TRIGGER proteger_y_rellenar_datos BEFORE INSERT OR
UPDATE OR DELETE
   ON numeros FOR EACH ROW
   EXECUTE PROCEDURE proteger_y_rellenar_datos();
```

```
INSERT INTO numeros (numero) VALUES (5);

INSERT INTO numeros (numero) VALUES (6);

UPDATE numeros SET numero = 10 WHERE numero = 6;

DELETE FROM numeros where numero =10;

SELECT * from numeros;
```

Triggers

```
CREATE TABLE cambios(
  timestamp_ TIMESTAMP WITH TIME ZONE default NOW(),
  nombre_disparador text,
  tipo_disparador text,
  nivel_disparador text,
  comando text
);
```

Triggers

```
CREATE OR REPLACE FUNCTION grabar_operaciones() RETURNS
TRIGGER AS $grabar_operaciones$
  DECLARE
  BEGIN
    INSERT INTO cambios (
              nombre_disparador,
              tipo_disparador,
              nivel_disparador,
              comando)
        VALUES (
              TG_NAME,
              TG_WHEN,
              TG_LEVEL,
              TG_OP
             );
    RETURN NULL;
  END;
$grabar_operaciones$ LANGUAGE plpgsql;
```

Triggers

```
CREATE TRIGGER grabar_operaciones AFTER INSERT OR UPDATE OR
DELETE
   ON numeros FOR EACH STATEMENT
   EXECUTE PROCEDURE grabar_operaciones();
```

```
INSERT INTO numeros (numero) VALUES (100);

SELECT * from numeros ;

SELECT * from cambios ;

UPDATE numeros SET numero = 1000 WHERE numero = 100;

DELETE FROM numeros where numero =1000;
```

BIBLIOGRAFÍA

PostgreSQL 12.0 Documentation. The PostgreSQL Global Development Group
Copyright © 1996-2019

PostgreSQL 12.0 Server Programming. Second Edition
Dar, Krosing, Mlodgenski, Roybal.
Packt Publishing Ltd. 2015