IIC2413 — Bases de Datos

PL/pqSQL

La mayoría de los DBMS ofrecen la posibilidad de programar funciones o procedimientos almacenados en el mismo DBMS, de forma que al correr estos se pueda tomar ventaja de todas las bondades de los DBMS. Esta semana vamos a aprender a programar estos procedimientos. Sqlite3 no soporta esta funcionalidad, así que vamos a tener que recurrir nuevamente a PostgeSQL.
Para minimizar la carga del servidor, asignaremos una base de datos nueva por cada grupo del proyecto. Para entrar a esta base de datos, ingresa vía ssh con tu nombre de usuario `grupoxx` como siempre. Para entrar a la base de datos, ejecuta:

psql grupoxxstoredy

Donde grupoxx es tu nombre de grupo, e y es 1 o 2 (es una base de datos por persona, ponganse de acuerdo para ver quien usa la 1 o la 2). Por ejemplo, el usuario 1 del grupo 29 escribe:

psql grupo29stored1

Antes de comenzar, crea una relación (¡asegúrate que estas en la base de datos stored para que esto no interfiera en tu entrega!)

Personas(run:varchar, nombre:varchar,apellido:varchar)

Ahora estamos listos para crear nuestra primera función!

1. Funciones Básicas

La forma básica de las funciones es:

```
CREATE or REPLACE Function <nombre_función> (<atributos de input>)
RETURNS <tipo_retorno> AS
$$
DECLARE
    <declaracion de variables>
BEGIN
    <sentencias SQL>
END
$$ language plpqsql;
```

Acá tenemos:

- `<nombre_función>` Es el nombre que le quieras dar a la función
- `<atributos de input>` Es una lista de atributos que recibe la función como input, de acuerdo a la sintáxis `(input_1 tipo_1, input_2 tipo_2, . . . , input_n tipo_n)`
- `<declaración de variables>` Es una lista de variables a declarar. Los tipos más comunes son integer, numeric, varchar y record.
- `<sentencias SQL>` contiene una lista de instrucciones SQL, cada una terminando en punto y coma (;)

Hagamos entonces nuestra primera función. Abre en tu computador un archivo de texto, y copia

```
CREATE OR REPLACE FUNCTION
insertar_persona (rut varchar, nombre varchar, apellido varchar)
RETURNS void AS $$
BEGIN
insert into personas values (rut,nombre,apellido);
END
$$ language plpgsql
```

Guardalo como `insertar_persona.sql`, y almacenalo como texto plano[1]. Ahora usa un servidor de ftp para mover este archivo al servidor codd.ing.puc.cl, a la carpeta home/grupoxx. Vuelve a entrar a postgres, a la base de datos grupoxxstoredy. Ejecuta

\i insertar_persona.sql

Con esto le estás diciendo a postgreSQL que ejecute el comando que tenías guardado. PostgreSQL va crear la función, ¡que ya está lista para ser usada! Con esto podemos llenar datos de forma más rápida. Ejecuta

SELECT insertar_persona('unrut ','unnombre','unapellido')

Y verifica que la inserción fue realizada revisando la tabla Persona. ¿Puedes entender que es lo que pasó y para qué sirve tu función?
Ahora intentemos algo más radical.

[1] si queda como .rtf o como text con formato va a tener metadatos que postgres no va a saber interpretar

```
CREATE OR REPLACE FUNCTION
insercion_radical (numero integer)
RETURNS void AS $$
DECLARE
    temp varchar;
BEGIN
    FOR i IN 1..numero LOOP
        temp := to_char(i,'99999999');
        insert into personas values (temp,temp,temp);
    END LOOP;
END
$$ language plpgsql
```

Nuevamente, almacena esto como insercion_radical.sql. Llévalo al servidor con ftp y ejecuta en postgreSQL:

\i insercion_radical.sql

Ahora estamos listos! Ejecuta lo siguiente para tener tu primera tabla con 10.000 tuplas

SELECT insercion_radical(10.000)

Hay hartas cosas que explicar acá.

- El control de flujo en esta función está dado por

  ```
  FOR <var> IN <x>..<y> LOOP
      <sentencias SQL>
  END LOOP;
  ```

  No necesitas haber declarado `<var>`, pero solo es válida dentro del loop (de hecho, al entrar al loop ignoras la declaración anterior, si existiese). En general `<x>` e `<y>` pueden ser números, variables numéricas o expresiones.

- Existen otras formas de iteración, como `WHILE` y `LOOP`

- La línea `temp := to_char(i,'99999999');` dice que la variable `temp` ahora corresponde a lo que retorna la función `to\_char`. Esta función toma un número n y un string f , y devuelve el número n como un string, en el formato que indicaste con f. En general postgreSQL tiene miles de funciones como esta, que ya están hechas. ¡Consulta la documentación!

Como control de flujo puedes escribir

```
IF <condicion booleana> THEN
    <sentencias SQL>
ELSE
    <sentencias SQL>
END IF;
```

En este caso la condición booleana es cualquier comparación que puedas escribir en SQL.

1.1. Recorriendo los resultados de una consulta

La capacidad para recorrer, en el entorno mismo del DBMS, los resultados de las consultas es quizá lo más importante en las funciones.
Para eso necesitamos una variable de tipo `RECORD`. Este es un tipo abstracto que sirve para contener los resultados de una tupla.
La forma básica de recorrer los resultados de una consulta es entonces:

```
    FOR <record> IN <consulta SQL> LOOP
        <sentencias SQL>
    END LOOP;
```

Esto funciona de la siguiente manera: El sistema ejecuta la `<consulta SQL>`, y va iterando tupla a tupla la respuesta: la primera tupla de la respuesta queda guardada en la variable `<record>`, cuando se terminan de ejecutar las `<sentencias SQL>` se pasa a la segunda iteración, donde la segunda tupla de la respuesta a `<consulta SQL>` pasa a la variable `<record>`, y así sucesivamente.
A modo de ejemplo, crea una tabla en postgres

PersonasCompleto(run:varchar, nombrecompleto:varchar)

La siguiente función se usa para copiar a PersonasCompleto el rut y la concatenación del nombre y el apellido de cada persona almacenado en la tabla Persona

```
CREATE OR REPLACE FUNCTION
transferencia ()
RETURNS void AS $$
DECLARE
    tupla RECORD;
    concat varchar;
BEGIN
    FOR tupla IN SELECT * FROM Personas LOOP
        concat = tupla.nombre || tupla.apellido;
        insert into personascompleto values (tupla.rut, concat);
    END LOOP;
END
$$ language plpgsql
```

2. Trucos avanzados

Puedes adivinar qué computa esta función?

```
CREATE OR REPLACE FUNCTION fib (
numero integer
) RETURNS integer AS $$
BEGIN
IF numero < 2 THEN
RETURN numero;
END IF;
RETURN fib(numero - 2) + fib(numero - 1);
END;
$$ LANGUAGE plpgsql;
```

Exacto, ¡son los números de Fibonacci! Esto no tiene nada que ver con bases de datos, dices. Pero ahora trata de ejecutar esta función en el servidor. Intentalo para n = 10. Ahora para n = 20. Ahora para n = 30... Se demora muchísimo, y si te acuerdas de introducción a la programación, tiene toda la razón en demorarse, por que nuestro algoritmo recursivo está mal escrito. Por ejemplo, para n = 30 tenemos que llamar a fib(29) y fib(28), pero ambas requieren el cómputo de fib(27), y por lo tanto las ejecuciones de fib para números menores terminan ejecutándose miles de veces.
Bueno, aprovechémonos de que tenemos una base de datos, y tratemos de guardar en una tabla temporal los resultados de fibonacci. No debería tomar mucho espacio.
Crea una relación

Fib_cache(num integer, fib integer)

Esta relación almacenará cada número num junto al número fibonacci correspondiente a num (es decir, fib(num)).
Ahora observa

```
CREATE OR REPLACE FUNCTION fib_cacheando(
  numero integer
) RETURNS integer AS $$
DECLARE
ret integer;
BEGIN
if numero < 2 THEN
    RETURN numero;
END IF;
SELECT INTO ret fib
FROM   fib_cache
WHERE  num = numero;
IF ret IS NULL THEN
```

```
ret := fib_cacheando(numero - 2) + fib_cacheando(numero - 1);

INSERT INTO fib_cache (num, fib)
VALUES (numero, ret);
END IF;
RETURN ret;

END;
$$ LANGUAGE plpgsql;
```

¿Que crees que está pasando? Vamos paso por paso.

```
CREATE OR REPLACE FUNCTION fib_cacheando(
  numero integer
) RETURNS integer AS $$
DECLARE
ret integer;
BEGIN
if numero < 2 THEN
    RETURN numero;
END IF;
```

Todo bien hasta acá, tomamos los casos bases 0 y 1

```
SELECT INTO ret fib
FROM   fib_cache
WHERE  num = numero;
```

Usamos `SELECT INTO <variable> <atributo> FROM...` para guardar en `<variable>` el valor de la consulta `SELECT <atributo> FROM...`. En este caso, buscamos en la tabla fib_cache si acaso tenemos la tupla del número de fibonacci correspondiente a `num`.

```
IF ret IS NULL THEN
ret := fib_cacheando(numero - 2) + fib_cacheando(numero - 1);

INSERT INTO fib_cache (num, fib)
VALUES (numero, ret);
END IF;
```

Si `ret` es nulo, significa que no tenemos este valor, y hay que obtenerlo utilizando una llamada recursiva. Si `ret` no es nulo, significa que ya lo tenemos, y no tenemos que hacer nada más.

```
RETURN ret;

END;
$$ LANGUAGE plpgsql;
```

Finalmente retornamos el valor de `ret`. Prueba ahora calcular fib_cacheando(30). ¡Mira como vuela esa función!

3. Retornando Tablas

Crea otra relación

Vuelo(ciudad_origen:varchar, ciudad_destino:varchar, horas:integer)

La idea de esta relación es que una tupla (ciudad origen, ciudad destino, N) indique que existe al menos un vuelo directo desde la ciudad origen a la ciudad destino, y que el vuelo más corto demora N horas. Llena esta tabla con unas cuantas tuplas para ir probando las funciones que debes hacer.
Queremos crear una función que reciba una ciudad C y retorne una tabla con todas las ciudades en las que existe un vuelo directo desde C, junto con el tiempo. Para esto, tenemos que decirle a la función que debe retornar una tabla (si, esto se puede hacer en SQL, pero lo hacemos acá por que es un buen ejercicio para empezar).
Veamos primero como retornar una tabla cualquiera:

```
CREATE OR REPLACE FUNCTION
retornar_vuelos ()
RETURNS TABLE (ciudad_origen varchar(50), ciudad_destino varchar(50),
                                                    horas integer) AS $$
BEGIN
RETURN QUERY Select * from Vuelo;
RETURN;
END
$$ language plpgsql
```

Nada muy impactante... lo único que hay que tomar en cuenta es que `RETURN QUERY` no retorna la función, solo asigna el valor de retorno a la tabla en cuestión (por eso retornamos explícitamente después).
Una vez que cargues esta función en PostgreSQL, prueba llamándola como hemos hecho hasta ahora:

SELECT retornar_vuelos();

¿Algún Problema? ¿No se supone que retornaba una tabla? Efectivamente, y por eso es que si queremos que la selección haga explícito todos sus atributos, tenemos que consultar lo que retorna la función como si fuera una tabla!

SELECT * from retornar_vuelos();

3.1. SQL dinamico

¿Qué pasa cuando quiero que mi consulta se vea modificada de acuerdo al input de la función? En ese caso tenemos dos opciones:

- Generar la consulta como una concatenación de strings (alternativa penca)
- Usar SQL dinámico (alternativa PUC)

Al igual que con lenguajes de programación, la idea de SQL dinámico es "preparar" una consulta antes de saber los parámetros, y una vez que tengamos el valor de los parámetros, ejecutar la consulta, mediante el comando `EXECUTE`. Veamos el ejemplo de los vuelos directos.

```
CREATE OR REPLACE FUNCTION
vuelos_directos (c_origen varchar)
RETURNS TABLE (ciudad_destino varchar(50), horas integer) AS $$
BEGIN
RETURN QUERY EXECUTE 'SELECT ciudad_destino, horas
        FROM Vuelo
        WHERE ciudad_origen = $1'
    USING c_origen;
    RETURN;
END
$$ language plpgsql
```

La idea es almacenar en variables (`$1`, `$2`, etc) los valores que necesitamos, y luego especificar las variables de donde sacamos estos valores con `USING`.