IIC2413 — Bases de Datos

PL/pqSQL – Segunda Parte

1. 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!

2. 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();

2.1. 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)

Por qué es mejor SQL dinámico lo discutiremos en clases. Pero veamos en que consiste.
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`.

2.2. Ejercicio

Imagina ahora que tienes una instancia de la tabla Vuelo con miles de datos, y te interesan todos los lugares a los que puedes llegar volando desde Santiago, sin importar el número de escalas. ¿Puedes pensar en un algoritmo para implementar esto usando una función? Escríbelo como una función en PLpgSQL. Puede que te sea útil pensar en como reutilizábamos una base de datos temporal para computar los números de Fibonacci.