IIC2413 – Bases de Datos

PL/pgSQL - Guía

Problema 1: Secuencia

En el siguiente ejercicio mostraremos un ejemplo de creación de tablas auxiliares para lo que encontremos necesario. Eventualmente podríamos necesitar números ascendentes en forma de secuencia, pero en forma de tabla... Por ejemplo, para crear indices haciendo producto cruz con otra tabla. Pero la secuencia puede ser de largo variable. Podríamos optar por tener una tabla en nuestra bases de datos que solo contenga números para este uso, pero esto complica el limite de cuantos números usar al tener que especificar en un WHERE cada vez que queramos utilizarla. Esto es mucho trabajo, la alternativa es crear una función capaz de generar esta tabla automáticamente, es decir buscamos una función `seq(n)` que retorne una tabla de una sola columna con enteros del 1 a n, por ejemplo:

```
SELECT * FROM seq(5);

1
2
3
4
5
```

Ahora deja de leer esto e intenta hacerlo tu mismo. Ten en cuenta lo siguiente: recuerda que la idea es crear una tabla auxiliar que aparece al llamado de `seq`, es decir, si revisáramos las tablas de la base de datos luego de ejecutar `seq`, no debería haber nuevas tablas.

Ok, ahora si la solución:

```
CREATE OR REPLACE FUNCTION seq(n integer)
RETURNS TABLE (number integer) as $
$
DECLARE
i integer;
BEGIN
  CREATE TABLE SEQ(number integer);
  for i in 1 .. n
  LOOP
    INSERT INTO SEQ VALUES(i);
  END LOOP;
  RETURN QUERY SELECT * FROM SEQ;
  DROP TABLE SEQ;
  RETURN;
END;
$$ language plpgsql;
```

Si la vemos, es bastante directo. El parámetro de entrada como dijimos es un `integer` n, y la función retorna una tabla con una sola columna de tipo `integer` también (que llamamos `number`). Luego creamos una tabla auxiliar para luego retornar `SEQ` e insertamos los números de 1 a n mediante un `for` en nuestra tabla. Especificamos el valor de retorno con `RETURN QUERY`, eliminamos la tabla auxiliar para no dejar rastro de tu

creación, y luego retornamos la función con `RETURN`.

Esta es una de las habilidades ventajosas de procedimientos almacenados, nos permiten crear tablas auxiliares mediante funciones que uno puede crear fácilmente en un lenguaje de programación como `Python`.

Propuesto

Ahora intente expandir nuestra definición de `seq`. Tendrá 3 parámetros, el inicio de la secuencia, el final y un incremento. Es decir, entrega una secuencia que parte en cierto valor, luego aumenta el valor según el incremento hasta superar el valor final. Lo anterior es para el caso en que el incremento sea positivo, en caso contrario, debe iniciar en la posición final y disminuir hasta pasar el valor inicial. Con ejemplos:

```
SELECT * FROM seq(1,5,1);

1
2
3
4
5

SELECT * FROM seq(3,8,2);

3
5
7

SELECT * FROM seq(2,4,-1);

4
3
2

SELECT * FROM seq(3,2,-1);
```

Ojo con el caso final. Como en este caso la secuencia disminuye y comenzamos con 2 que ya es menor a 3, entrega la tabla vacía.

Problema 2: Resta Casera

Otro buen ejercicio es realizar a mano operaciones nativas de SQL con procedimientos almacenados. En el siguiente realizamos la operación resta que conocimos en álgebra relacional y que existe como `EXCEPT` en SQL. Para recordar, esta recibe 2 tablas unión-compatibles(mismo número de columnas y de mismos tipos) y retorna aquella tabla que contiene todas las tuplas de la primera que no es encuentran en la segunda. Para simplificar, supongamos que existen 2 tablas A y B con columnas a y b cada una, y que la única consulta SQL que podemos realizar sobre estas tablas es obtener todos los elementos (`SELECT * FROM A/B`). Realizar un procedimiento almacenado que calcule la resta de estas 2 tablas:

Deje de leer, inténtelo y vuelva:

```
CREATE OR REPLACE FUNCTION restar()
RETURNS TABLE (number1 integer, numer2 integer) as $$
DECLARE
  r1 record;
  r2 record;
  se_restan bool;
BEGIN
  CREATE TABLE resta(number1 integer, number2 integer);

  FOR r1 IN SELECT * FROM A
  LOOP
    se_restan := FALSE;
    FOR r2 IN SELECT * FROM B
    LOOP
      IF r1 = r2 THEN
        se_restan := TRUE;
      END IF;
    END LOOP;

    IF NOT se_restan THEN
      INSERT INTO resta VALUES(r1.a, r1.b);
    END IF;
  END LOOP;

  RETURN QUERY SELECT * FROM resta;
  DROP TABLE resta;
  RETURN;
END;
$$ language plpgsql;
```

Nuevamente, bastante directo. Iteramos sobre los elementos de A creando para cada uno una variable booleana `se_restan` con la cual verificamos si se encuentra el elemento de A en B. En caso de no estar, lo agregamos tranquilamente a nuestra tabla auxiliar `resta`.

Propuesto

Teniendo los mismos supuestos de antes, es decir, dos tablas A y B con columnas a y b cada una. Escriba un proceso almacenado almacenado que calcule el producto cruz de A y B.

Problema 3: Calcular Distancias

Otro ejercicio útil, sobre todo cuando se trabaja con bases de datos de lugares que contienen coordenadas geográficas, es calcular la distancia entre un punto dado y los lugares de la base de datos y filtrar por una distancia dada. De esta forma, se obtienen todos los lugares que se encuentren por ejemplo, a menos de 10 km el Campus San Joaquín. Para esto, resulta muy útil crear función capaz de generar esta tabla de resultados utilizando procedimientos almacenados. Para calcular la distancia entre dos lugares podemos usar la fórmula de Haversine:

- `Lugar1 = (lat1, lng1)`
- `Lugar2 = (lat2, lng2)`

$distancia = 6371 \cdot acos(cos(radians(lat1)) \cdot cos(radians(lat2)) \cdot cos(radians(lng2)) - radians(lng1) + sin(radians(lat1)) \cdot sin(radians(lat2)))$

En donde `Lugar1` es nuestro punto de referencia/comparación. Realizar un procedimiento almacenado que dado un punto de referencia y una distancia, entregue una tabla con todos los lugares que se encuentran a una distancia menor al valor entregado. *Hint: Revisar SQL Dinámico.*

Para esto, tienen la tabla `Lugares(id integer, nombre varchar(50), lat float, lng float)`. Ahora deja de leer e inténtalo. Una vez que creas tener la solución, revísala:

```
CREATE OR REPLACE FUNCTION
HaversineDistance(lat float, lng float, distancia integer)
RETURNS TABLE (id integer, nombre varchar(50), distancia float) AS $$
BEGIN
RETURN QUERY EXECUTE 'SELECT id, nombre, (6371 * ACOS(
                        COS(RADIANS($1))
                        * COS(RADIANS(lat))
                        * COS(RADIANS(lng))
                        - RADIANS($2)
                        + SIN(RADIANS($1))
                        * SIN(RADIANS(lat)))
                        ) AS distancia
                      FROM Lugares
                      HAVING distancia < $3'
      USING lat, lng, distancia;
      RETURN;
END
$$ language plpgsql
```

Si lo vemos, es bastante directo. EL procedimiento recibe 3 parámetros de entrada y se genera la consulta en SQL que filtra a través del `HAVING` los lugares cuya distancia al lugar de referencia sea menor a `distancia` (en [km]). Finalmente, para ejecutar el procedimiento se debe llamar a `SELECT * FROM HaversineDistance(-33.497579, -70.612246, 10);`, que nos entregará todos los lugares que se encuentren a menos de 10 km del Campus San Joaquín.

Propuesto

Supongamos existe la tabla `Lugares` como en el ejercicio anterior. Escriba una procedimiento almacenado que utilizando esa tabla, cree las siguientes dos tablas en la base de datos: `Distancia(idlugar1 int, idlugar2 int, distancia float)` y `VecinoMasCercano(idlugar int, idvecino int)`. Como los nombren sugieren, `Distancia` indica la distancia entre cada par de lugares de la tabla `Lugares` y `VecinoMasCercano` contiene para cada lugar, su id y el id del lugar más cercano (que no es si mismo). A diferencia de casos anteriores, este procedimiento no debe entregar las tablas si no simplemente dejarlas almacenadas en la base de datos donde se trabaja y asume la existencia de la tabla `Lugares`, no es necesario ingresar su nombre como parametro.