Exercise 6

Date of issue: 4th April 2017 Deadline: 11th April 2017

The exercise is based on the *Mondial* database. Its schema can be accessed at `https://files.ifi.uzh.ch/dbtg/dbs/mondial/schema.pdf`

In addition to your written (or printed) submission, we kindly ask you to send your submission also via **email** to your teaching assistant. The written (or printed) submission is still mandatory and has precedence over any digital submission if they differ in content.

1 Geodistance Function

In this task the goal is to implement a user defined function (UDF) to compute the distance (in kilometers) between two GPS coordinates. In the Mondial database GPS coordinates are stored in the form of the custom data type `GeoCoord` that consists of two attributes `Latitude` and `Longitude`.

Let (ϕ_1, λ_1) and (ϕ_2, λ_2) be two GPS coordinates, where ϕ_1, ϕ_2 and λ_1, λ_2 are latitudes and longitudes in degrees, respectively. The geodistance d between this two points is computed as follows:

$$
\begin{aligned}
R &= 6371 \qquad \text{(radius of the earth in kilometers)} \\
c &= \frac{2\pi}{360} \qquad \text{(factor to convert degrees to radians)} \\
a &= \sin^2\left(\frac{c(\phi_2 - \phi_1)}{2}\right) + \cos(c \cdot \phi_1) \times \cos(c \cdot \phi_2) \times \sin^2\left(\frac{c(\lambda_2 - \lambda_1)}{2}\right) \\
d &= R \times 2 \times \text{atan2}(\sqrt{a}, \sqrt{1-a})
\end{aligned}
$$

The four-quadrant inverse tangent function $\text{atan2}(x, y)$ is a variant of the arctangent trigonometric function, where the signs of x and y are used to determine the quadrant of the resulting angle.

Tasks:

1. Develop and **write down** a UDF that implements the above formula and returns distance d. The UDF's signature is

   ```
   geoDist(c1 GeoCoord, c2 GeoCoord)
   RETURNS NUMERIC
   ```

   See PostgreSQL's documentation[1] for a description of the required math functions.

2. You can check your function with the following example query and its expected resulting table.

   ```
   SELECT geoDist(
     (SELECT Coordinates FROM island WHERE Name = 'Kos'),
     (SELECT Coordinates FROM island WHERE Name = 'Samos')
   ) AS Dist;
   ```

[1] `https://www.postgresql.org/docs/9.4/static/functions-math.html`

Dist
103.538090149668

3. Execute the following query and **write down** its result

```
SELECT geoDist(
  (SELECT Coordinates FROM island WHERE Name = 'Guam'),
  (SELECT Coordinates FROM island WHERE Name = 'Korfu')
) AS Dist;
```

2 Computing Nearest Islands

Computing the nearest neighbors of a spatial object (e.g. an island) is a common task. The following UDF receives the name `IName` of an island as single input parameter and returns a table of three attributes. The function returns `IName`, the names of the 3 nearest islands according to the previously defined distance function, and their distance to this island `IName`.

Task: Execute the following function definition, which will be needed in the tasks to follow.

```
CREATE OR REPLACE FUNCTION nearestIslands(IName VARCHAR(35))
RETURNS TABLE(Name VARCHAR(35), Neighbor VARCHAR(35), Dist NUMERIC)
AS $$
BEGIN
  RETURN QUERY
  SELECT i1.Name AS Name, i2.Name AS Neighbor,
         geoDist(i1.Coordinates, i2.Coordinates) AS Dist
  FROM island i1, island i2
  WHERE i1.Name = IName AND i2.Name <> i1.Name
  ORDER BY geoDist(i1.Coordinates, i2.Coordinates), i2.Name
  LIMIT 3;
END; $$ LANGUAGE plpgsql;
```

3 Caching Nearest Islands

Computing nearest neighbors is a common, but expensive operation. The goal of this task is to cache the results of the UDF created in the previous task to speed up repeated invocations of the same nearest neighbor query.

Tasks:

1. Write down a **`CREATE TABLE`** statement that creates a new table called **cachedNearestIslands**, which has the same schema as the table that function `nearestIslands` returns.

2. Develop and **write down** a UDF that takes one parameter and has the following signature:

   ```
   fastNearestIslands(IName VARCHAR(35))
   RETURNS TABLE(Name VARCHAR(35), Neighbor VARCHAR(35), Dist NUMERIC)
   ```

   The UDF should make a lookup in table **cachedNearestIslands** to see if the nearest neighbors of island `IName` are cached.

- If they are cached, a message should be logged on the NOTICE level saying 'Cache hit for island "IName"' – replacing IName with the actual name of the island (cf. PostgreSQL's documentation[2]). Then the corresponding tuples are returned.
- If they are not cached, a message should be logged on the NOTICE saying 'Cache miss for island "IName"' – replacing IName with the actual name of the island. Then the neighbors are computed with UDF `nearestIslands`, stored in the cache and the tuples returned.

3. Execute the following query and **write down** any printed log message and the resulting table

```
SELECT * FROM fastNearestIslands('Guam');
```

4 Cache Refresh

When table **island** is modified, the actual nearest neighbors might change and differ from the cached ones. The goal of this task is to develop a trigger to automatically refresh the cache upon insertions on table **island**.

Tasks:

1. Write a trigger that refreshes the **cachedNearestIslands** table on every *insert* of a new tuple into the **island** relation. Refreshing means that for every cached island (attribute `Name` in **cachedNearestIslands**) (a) a message should be logged on the NOTICE level saying 'Refreshing nearest neighbors for island "Name"' (replacing Name with the actual name of the island), and (b) its nearest islands are recomputed.

2. Execute the following two queries and **write down** any printed log message and the resulting table.

```
INSERT INTO island VALUES ('Test1', 'Test1', 100,
  100, NULL, (144.9,13.1));

SELECT * FROM fastNearestIslands('Guam');
```

[2] https://www.postgresql.org/docs/9.4/static/plpgsql-errors-and-messages.html