CSE 462: Project 1 (due on March 12, 2015, at 23.55 ET)

This is an individual project: no cooperation is allowed.

Problem 1: SQL (10 points)

You are given the following relational schema:

`EMP(SSN,EmpName)`
`ASSIGN(SSN,CityName,StartYear,EndYear)`
`CITY(CityName,Country)`

Keys are underlined. In `ASSIGN`, `CityName` is a foreign key referencing `CityName` in `CITY`, and `SSN` is a foreign key referencing `SSN` in `EMP`. The domain of the attributes `StartYear` and `EndYear` is Years.

The meaning of ASSIGN is as follows: a tuple (n, c, s, e) represents the fact that the employee with SSN equal to n was assigned to the city c from the year s to the year e (not including e). For example, there could be two assignments in Paris:

`(123,Paris,2000,2001),(555,Paris,2002,2005).`

The first assignment was for one year: 2000; the second, for three years: 2002, 2003, and 2004. It is possible that the same person was assigned to the same city more than once and that more than one person is assigned to the same city at the same time. There are no years beyond 2015 in the database.

Write the following queries in SQL:

Query 1.1 List the names of all the employees and for each employee calculate the average assignment length of this employee. The result should be sorted (descending) by that length.

Query 1.2 Let the maximum assignment length across all times, cities and employees be M. List all cities where there was an assignment of length M.

Query 1.3 List all employees that were assigned to Paris immediately after they were assigned to Moscow, i.e., there was no gap between the two assignments.

Query 1.4 List all employees having an assignment overlapping, perhaps partially, with an assignment of Judy Brown to the same city. An example of partial overlap:

`(123,Paris,2000,2002),(555,Paris,2000,2005),`

Query 1.5 List all employees who were assigned to the same cities as Jim Smith, though perhaps at different times.

Please write all your SQL statements into a text file, named `Project01_SQL.txt`. Please use semicolons to separate each command from the others. Make sure that the resulting file can be executed in Tora without any modification (assuming the relations `EMP`, `ASSIGN` and `POST` are already present).

Problem 2: JDBC (10 points, plus 5 extra credits)

You are given a relation instance named MAP, with the following schema (we use Oracle's syntax here):

```
CREATE TABLE Map
(
City VARCHAR(40) PRIMARY KEY,
Latitude DECIMAL(*,10) NOT NULL,
Longitude DECIMAL(*,10) NOT NULL
);
```

Each row in MAP corresponds to a geographic location, whose latitude and longitude are expressed in *degrees*. For example, the content of MAP may look like this:

City	Latitude	Longitude
Wues	49.79667	6.15556
Wolwelange	49.82861	5.76472
Wolfsmuhle-lès-Ellange	49.53333	6.31667
Wollefsmillen	49.71639	6.49
Wintrange	49.50139	6.35278
Wilwerdange	50.14056	6.02389
...	...	...

Your goal is to develop a Java application able to answer the following queries:

Range queries (5 points) Given the name of a city c and a radius r (expressed in kilometers) return all the cities whose distance from c is strictly less than r. By *distance* we mean the length of the shortest trajectory over the earth's surface that connects the two cities (more details are given below).

Top-k nearest neighbors (5 points) Given the name of a city c and a strictly positive integer k return the k cities that are closest to c.

Shortest round-trip query (extra credit: 5 points) Given the names of two cities, c_1 and c_2, find the name of a third, distinct city c_3 so that the sum of the distances $d(c_1, c_2) + d(c_2, c_3) + d(c_3, c_1)$ is minimized.

The MAP relation will be stored inside an Oracle database; your application will need to access it using JDBC. In order to be graded, your code must meet the following constraints:

1. You must write all your code in a single class, named `Project01_Main`.
2. The package of `Project01_Main` must be `edu.buffalo.cse462`.
3. Class `Project01_Main` must be executable, that is it must implement the method `main(String[] args)`.
4. Invocations to `Project01_Main` must respect a given interface, that is described below.
5. The output generated by `Project01_Main` must respect a given format, described below.
6. You should not use external libraries, except for `ojdbc6.jar`. Your code must be compilable with JDK v7.

Students who fail to comply with these rules will not receive credits, as their projects will not be graded.

Invocation interface and output format

The first two arguments passed to the `main` method of `Project01_Main` are always the username and password for accessing an Oracle account, where the MAP relation is stored. The third one is a string specifying the kind of query to be processed: its possible values are `RANGE_QUERY`, `NN_QUERY` or `MIN_ROUNDTRIP_QUERY`.

Range queries

For range queries (flagged with `RANGE_QUERY`), the fourth and fifth parameters are the name of a city and a decimal number, for example:

```
java edu.buffalo.cse462.Project01_Main <username> <passwd> RANGE_QUERY Wues 4.8
```

The above invocation should return the list of the cities that are within 4.8 kilometers from Wues. The list must be printed on the standard output, one line per city, in ascending order of distance. For each city the actual distance from the target (Wues) must be specified; the distance must be expressed in kilometers and rounded so to have only two digits of precision. A well-formatted output for the above invocation should look like this:

```
Schrondweiler 0.29
Cruchten 1.72
Oberglabach 2.16
Ferme Thibesart 3.57
Stegen 3.89
Leihof 4
Moesdorf 4.23
Pettingen 4.44
Angelsberg 4.49
Maison Burg 4.54
Meysembourg 4.57
```

Top-k nearest neighbors queries

When the third parameter is `NN_QUERY`, the fourth and the fifth ones are the name of a city and a positive integer. For example:

```
java edu.buffalo.cse462.Project01_Main <username> <passwd> NN_QUERY Angelsberg 5
```

The above invocation should return the top-5 cities that are closest to Angelsberg. The list must be printed on the standard output, one line per city, in ascending order of distance. For each city the actual distance from the target must be specified; the distance must be expressed in kilometers and rounded so to have only two digits of precision. A well-formatted output for the above invocation should look like this:

```
Schoos 1.54
Meysembourg 2.1
Oberglabach 2.46
Wickelscheid 2.9
Beringen 3.16
```

Shortest round-trip query

When the third parameter is `MIN_ROUNDTRIP_QUERY`, the fourth and the fifth ones are names of cities. For example:

```
java edu.buffalo.cse462.Project01_Main <username> <passwd>
MIN_ROUNDTRIP_QUERY Schoos Beringen
```

The above invocation should return a single line of text, specifying the minimum round-trip distance (in kilometers, rounded so to have two digits of precision) an the names of the three cities. For example:

```
Schoos Beringen Angelsberg 8.88
```

Testing your code

You will be given a file with the SQL commands to create an instance of the relation MAP inside your Oracle account. This data set contains information about 400 cities of Luxembourg, and was used to generate the examples above. Before submitting your project you should test it on timberlake, making sure that your code compiles and return answers that match those presented above. After uploading the data set into your Oracle account, you should do the following:

- Upload your java code to timberlake, using scp or similar tools.
- Open an ssh console to timberlake (`ssh <username>@timberlake.cse.buffalo.edu`)
- Run the script `/util/oracle/coraenv.sh`
- Compile your code using `javac`
- Run your code and try all the queries discussed above. Your output must match the one presented here.

Computing the distance between two locations

In this section we provide a quick, step-by-step guide to compute the approximate distance between two points on the surface of Earth. Let's assume we are given with two tuples from MAP, $(city_1, latitude_1, longitude_1)$ and $(city_2, latitude_2, longitude_2)$, and we want to compute the distance between $city_1$ and $city_2$, in kilometers:

Step 1 First we need to convert degrees to radiants. Let's denote by ϕ and λ the latitude and the longitude, respectively, expressed in radiants:

$$\phi_1 = latitude_1 \cdot \frac{\pi}{180} \qquad \lambda_1 = longitude_1 \cdot \frac{\pi}{180} \tag{1}$$

$$\phi_2 = latitude_2 \cdot \frac{\pi}{180} \qquad \lambda_2 = longitude_2 \cdot \frac{\pi}{180} \tag{2}$$

Step 2 Next we compute the "haversine" h:

$$h = \sin^2\left(\frac{\phi_1 - \phi_2}{2}\right) + \cos(\phi_1) \cdot \cos(\phi_2) \cdot \sin^2\left(\frac{\lambda_1 - \lambda_2}{2}\right) \tag{3}$$

Step 3 The distance d between $city_1$ and $city_2$, expressed in kilometers, is given by:

$$d = 12{,}742 \cdot \operatorname{atan2}(\sqrt{h}, \sqrt{1-h}) \tag{4}$$

Hint: SQL supports many trigonometric functions. More information is available here: `http://docs.oracle.com/cd/B28359_01/server.111/b28286/functions001.htm`. If you want to check whether your code computes distances correctly, you may test it against this website: `http://www.movable-type.co.uk/scripts/latlong.html`.

Submission Guidelines

Each student must submit two files, named `Project01_SQL.txt` and `Project01_Main.java`. The use of these filenames is mandatory. The first file should contain the answers to Problem 1; the second file should be a compilable Java class, addressing Problem 2. Both files must be submitted using the command `submit_cse462`, available on any departmental machine. You may submit multiple times, only the last submission before the deadline will be graded. By submitting a file, you confirm that you have read and agree to all university's policies and regulations on academic integrity (`http://academicintegrity.buffalo.edu/policies/`). Violations of academic integrity will be penalized accordingly. All files are due on **March 12th, at 23.55 EST**. No late submissions will be accepted.