

## CSE 512 – Assignment 1

The required task is to simulate data partitioning approaches on-top of an open source relational database management system (i.e., PostgreSQL). Each student must generate a set of Python functions that load the input data into a relational table, partition the table using different horizontal fragmentation approaches, and insert new tuples into the right fragment.

Input Data. The input data is a Movie Rating data set collected from the MovieLens web site (<http://movielens.org>). The raw data is available in the file ratings.dat.

The ratings.dat file contains 10 million ratings and 100,000 tag applications applied to 10,000 movies by 72,000 users. Each line of this file represents one rating of one movie by one user, and has the following format:

UserID::MovieID::Rating::Timestamp

Ratings are made on a 5-star scale, with half-star increments. Timestamps represent seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970. A sample of the file contents is given below:

```
1::122::5::838985046
1::185::5::838983525
1::231::5::838983392
```

Required Task. Below are the steps you need to follow to fulfill this assignment:

1. Download the virtual machine that has the same environment with the grading machine. This is highly recommended. You can use your own machine. But it is not ensured that you code can work in the grading machine. If you use the provided machine, then **skip Step 2**. Virtual Machine setting: Python 2.7.x. Ubuntu 16.04.
2. Install PostgreSQL.
3. Download ratings.dat file from the MovieLens website, <http://files.grouplens.org/datasets/movielens/ml-10m.zip>

You can use partial data for testing. One testing data file is given on blackboard.

4. Implement a Python function `LoadRatings()` that takes a file system absolute path that contains the ratings.dat file as input. `LoadRatings()` then loads the ratings.dat content into a table (saved in PostgreSQL) named `Ratings` that has the following schema

UserID (**int**) – MovieID (**int**) – Rating (**float**)

5. Implement a Python function `Range_Partition()` that takes as input: (1) the `Ratings` table stored in PostgreSQL and (2) an integer value `N`; that represents the number of partitions. `Range_Partition()` then generates `N` horizontal fragments of the `Ratings` table and store them in PostgreSQL. The algorithm should partition the ratings table based on `N` uniform ranges of the `Rating` attribute.
6. Implement a Python function `RoundRobin_Partition()` that takes as input: (1) the `Ratings` table stored in PostgreSQL and (2) an integer value `N`; that represents the number of partitions. The function then generates `N` horizontal fragments of the `Ratings` table and stores them in PostgreSQL. The algorithm should partition the ratings table using the round robin partitioning approach (explained in class).
7. Implement a Python function `RoundRobin_Insert()` that takes as input: (1) `Ratings` table stored in PostgreSQL, (2) `UserID`, (3) `ItemID`, (4) `Rating`. `RoundRobin_Insert()` then inserts a new tuple to the `Ratings` table and the right fragment based on the round robin approach.
8. Implement a Python function `Range_Insert()` that takes as input: (1) `Ratings` table stored in PostgreSQL (2) `UserID`, (3) `ItemID`, (4) `Rating`. `Range_Insert()` then inserts a new tuple to the `Ratings` table and the correct fragment (of the partitioned ratings table) based upon the `Rating` value.

### ***Frequently Asked Questions:***

- Partition numbers start from 0, if there are 3 partitions then range\_part0, range\_part1, range\_part2 are partition table names for range partitions and similar numbering should be done for round robin partitions.
- Do not change partition table names prefix given in assignment\_tester.py
- Do not hard code input file name.
- Do not close the connection inside the implemented function.
- Do not hard code database name.
- Table schema should be equivalent to what has been described in point 4.
- Use Python **2.7.x** version.

### ***Question with respect to Partitioning:***

The number of partitions here refer to the number of tables to be created.

For rating values in [0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5]

Case N = 1,

One table containing all the values.

Case N = 2,

Two tables,

Partition 0 has values [0, 2.5]

Partition 1 has values (2.5, 5]

Case N = 3,

Three tables,

Partition 0 has values [0, 1.67]

Partition 1 has values (1.67, 3.34]

Partition 2 has values (3.34, 5]

Uniform ranges means a region is divided uniformly, I hope the example gives a clear picture.

### ***Hint***

1. Do not use global variables in your implementation. Meta-data table is allowed.
2. You are not allowed to modify the data file.
3. Pass all the test cases cannot ensure that your answer is correct. It means there is no compile error in your code. To fully verify your implementation, you need to check content of the corresponding tables in the database.
4. Two insert functions can be called many times at any time. They are designed for maintaining the tables in the database when insertions happen.

### ***Submission***

1. **Only** submit the Interface.py file. Do **not** change the file name. Do **not** put it into a folder or upload a zip.
2. Multiple submissions are allowed. Only the **latest** submission will be graded. **No** late submission is accepted.

***Please make use of Discussion Board extensively to clear any doubts. It is your responsibility to check all the threads in Discussion board. Loss of points caused by missing information in Discussion board cannot be argued.***