CS236

Qizhong Mao
862002283

Krupa Hegde
862006278

Project Report

Part 1:

A Python script was used for sampling 10% of the data, and saving the sampled data to CSV file. The sampled CSV file was loaded as a DataFrame. An R-Tree index was built on the DataFrame on longitude and latitude. Since R-Tree creates partitions on the data, *mapPartitions* was used to access each partition. We could compute the MBR by iterating all points in each partition, updating the minimum and maximum longitude and latitude accordingly. We tests with a bunch of number of partitions, and decided to use 64 partitions.

To plot the points and MBRs, we output all the points and MBRs in CSV format, where points are represented by 2 float values, and MBRs are represented by 4 float value. Python's matplotlib was used to plot the points and rectangles.

Part 2:

Task1:

Solution:

To retrieve the restaurants located inside the 5th road ring of the city, first build an R-Tree index over the latitude, longitude coordinates. Figure Task1.1 shows Simba syntax to build an R-Tree index. *pois* is a DataFrame which has values of the POI dataset. Using the columns of *lat*, *lon* in the dataframe Simba builds an R-Tree.

```
import simba.simbaImplicits._

pois.index(RTreeType, "rtPoints", Array("lon", "lat"))  // R-Tree index over longitude and latitude

println("Index done")
```

Fig Task1.1

By doing a range query (search in a rectangle) over the points with given latitude and longitude, the data is filtered by checking if the description part of the POI dataset has keyword "amenity=restaurant". Resulting DataFrame is written to a file. Fig Task 1.2 shows the Simba and Spark SQL implementation for range query and filter data.

```
val results = pois
  .range(Array("lon", "lat"), Array(minLon, minLat), Array(maxLon, maxLat))  // Search in rectangle
  .filter($"desc".contains("amenity=restaurant"))  // Filter by restaurant
  .orderBy("id")  // Order by ID
  .collect()
```

Fig Task1.2

Task2:

Solution:

There are several steps that need to be done for this task.

1. All objects within 2000 meters of (-322357, 4463408) in a circle are filtered.
2. From the timestamp of each point, a new column *weekday* with the weekday (Mon, Tue, Wed, Thu, Fri, Sat and Sun) is created for further filtering, and a new column *date_hour* with the date and hour of the day is created, also for future filtering.
3. Filter the above DataFrame by selecting only weekdays, that is *weekday* is not "Sat" or "Sun", to get all points only in weekdays.
4. Find all distinct points (so we don't count the same object twice) for each hour in the whole dataset, by grouping by key *date_hour*.
5. Further group the points by key *hour* (0 - 23) among all days to find the number of distinct points in each hour.

Fig Task2.1 shows the Simba implementation of circleRange Query and fig Task 2.2 shows the sample output

```
val tmpResults = points
  .circleRange(Array("lon", "lat"), Array(-322357.0, 4463408.0), 2000.0)  // Search in circle within 2000 m
  .withColumn("weekday", date_format(col("time"), "EE"))  // Get weekday
  .withColumn("date_hour", date_format(col("time"), "yyyy-MM-dd HH"))  // Get hour of day
  .filter($"weekday" =!= "Sat" && $"weekday" =!= "Sun")  // Filter by weekdays (not weekends)
```

Fig Task2.1

Number of people near the Tiananmen Square are:

Task 3:

Solution:

To answer this question, first we need to know the start points and stop points. For start points, we would like to find the point with the earliest (minimum) timestamp for a given set of trajectory ID (*tid*) and object ID (*oid*), and obtain the longitude and latitude of the point. Similarly, we can obtain the stop points with the latest (maximum) timestamp.

Next, we create 2 DataFrames that refer to all trajectories starting and stopping in the big MBR, respectively. The joined results of these 2 DataFrame are all the trajectories starting and ending in the big MBR. Some trajectories may have starting point or stopping point in the big MBR, but not both of the points.

Then, for each quadrant (top left, bottom left, top right and bottom right), we filter the trajectories starting and stopping in that quadrant, respectively. Therefore, for each quadrant, we have 2 DataFrames, one for trajectories with starting point in the quadrant, the other for trajectories with stopping point in the quadrant. And we take the natural join with these 2 DataFrames per quadrant, finding the trajectories starting and stopping in that quadrant, and union all the results here. We would get all the trajectories starting and stopping in the same quadrant. In case some trajectory's start or stop point lies on the boundings such that the trajectory may be counted multiple times, we only take the distinct trajectories (*tid*, *oid*) and count them.

Because we already have the number of trajectories starting and stopping in the big MBR, subtracting by the number of trajectories starting and stopping in the same quadrant, we can get the number of trajectories starting and stopping in different quadrants.

Total number of trajectories starting and ending in the larger rectangle:

Number of trajectories starting and ending in same quadrants:

Number of trajectories starting and ending in different quadrants:

Task 4:
Solution:
Since this task requires a subset of data that happened between February and June, we used a Python script to do the sampling based on the timestamp of each point.

Because the task needs to find neighbors for every point in the same dataset, a join query is needed on the same dataset, which is a self-join. However, Spark does not allow self join on the same dataset (at least in memory dataset), the only solution is to load the input file twice to make 2 identical datasets.

Then the 2 identical datasets use distance join within 100 meters based on their longitude and latitude. To count the total number of nearby neighbors, a group by is applied for count aggregation. Note here every point will surely have itself as a nearby neighbor during the join, thus for the count, we need to subtract by 1 to remove itself.

Finally, we order the results in descending order, and select the top 20 points as the result.

Simba implementation of distance join is shown in Fig Task 4.1

```
val results = points1.as("ps1")
  .distanceJoin(points2.as("ps2"), Array("lon", "lat"), Array("lon", "lat"), args(3).toDouble)  //
  .select($"ps1.*")  // Select only attributes from points1
  .groupBy("tid", "oid", "lon", "lat", "time")  // Group by for count aggregation
  /* Count number of neighbors and minus 1. This removes the same point as its neighbor. */
  .agg(count("*").minus(lit(1)).as("cnt"))
  .orderBy($"cnt".desc)  // Sort in descending order
  .limit(20)  // Get top 20 results
  .collect()
```

Fig- Task4.1

Task 5:
Solution:
For this query, we need both trajectories and POI's data. First trajectories points data is filtered to select data with week days and 2008 year then same is done for year 2009. Month is extracted from the timestamp. On the trajectories and POIs DataFrames, R-Tree index is built.

Using distance join with distance these 2 DataFrames are joined. Result is counted by grouping id, year, month, latitude and longitude. It is ordered in descending order and top 10 points are selected for the output.

Fig Task5.1 shows simba implementation for filtering data and extracting month. Fig Task 5.2 shows distance join on POIs and trajectories data.

```
//filter data to select 2009 and only the week days
val pointsTemp2009 = points.withColumn("Year",year(points("time")))
          .withColumn("month",month(points("time"))) //extract month from timestamp
          .withColumn("weekday",date_format(col("time"),"E"))
          .withColumnRenamed("lon","long").withColumnRenamed("lat","latt")
          .filter($"weekday" =!= "Sat" && $"weekday" =!= "Sun")
          .filter($"year"==="2009" )
```

Fig Task5.1

```
//join POIs and trajectories using distance join
val dataJoined2008=pois.distanceJoin(trajpoints2008,Array("lat","lon"),Array("latt","long"),100.0).distinct()
val dataJoined2009=pois.distanceJoin(trajpoints2009,Array("lat","lon"),Array("latt","long"),100.0).distinct()

//count points by grouping them
val result2008=dataJoined2008.select("id","year","month","lon","lat")
               .groupBy("lon", "lat","year","month","id").count()
               .orderBy(desc("count")).limit(10) // sort in descending order to select top 10
```

Fig Task5.2

Part 3

For this Task 4 and Task 5 are run on different number of cores and the time is measured. Because of the memory issue, for task 4, less amount of data is taken in input. Sampling is done through the python code.

Table below shows the number of cores and time taken for task4.

Cores	Time taken in milliseconds
*	109631
1	113299

The code for the task4 accepts an argument for distance. It can be set to 500. Due to less compatibility of the laptop for distance=500 query is not providing the result.

The table below shows the number of cores and time taken for Task5.

Cores	Time taken in milliseconds
*	313617
1	758825

Note: I have only 2 cores in my laptop hence only 2 results added.

Task 5 with distance modified to 500 m- time taken is 129739 milliseconds.
Task 5 with 2008 and 2009 computed together: time taken is 140572 milliseconds.