Phase 1

- We used GeoSpark SpatialRDD – an extension of RDD for spatial dataset
- Point RDD represents geospatial data in terms of latitude and longitude
- An envelope represents the diagonal corners of a rectangular geographical area
- PointRDD is constructed from the csv data, using Envelope and SpatialRDD
- We executed 'Spatial Range Query', 'Spatial KNN Query' and 'Spatial Join Query' on the PointRDD

Phase 2

Spatial Join Query using the simple Cartesian Product algorithm

- Implemented the SpatialJoinQuery over a dataset of rectangles and points.
- The implementation details are as follows:
  - Load the PointRDD and RectangleRDD from their respective datasets
  - Collect all the envelopes from the RectangleRDD using rawSpatialRDD
  - For every envelope in the list, do:
    - SpatialRangeQuery on the PointRDD and the envelope
    - Collect all the points as tuple
  - Parallelize the collected data into JavaPairRDD and return it

Phase 2: Inference

Phase-1 Tasks

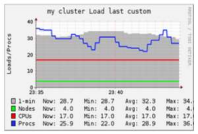
Query type	With R-tree Indexing	Without R-tree Indexing
Spatial Range Query	Better performance	Poor performance
Spatial KNN Query	Poor performance	Better performance
Spatial Join Query	Better performance	Poor performance

Phase-2 vs Phase-1 Tasks

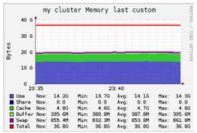
Equal grid with R-tree indexing	Equal grid without R-tree indexing	R-tree grid without R-tree indexing	Cartesian Product
Low execution time, least memory and low CPU usage	Low execution time, low memory and low CPU usage	Low execution time, low memory and low CPU usage	High execution time, high memory and High CPU usage

Cluster Analysis for Phase-2

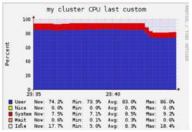
Load on the cluster



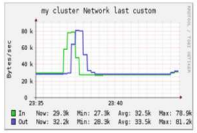
Memory utilization of cluster



CPU utilization of cluster



Network utilization of cluster



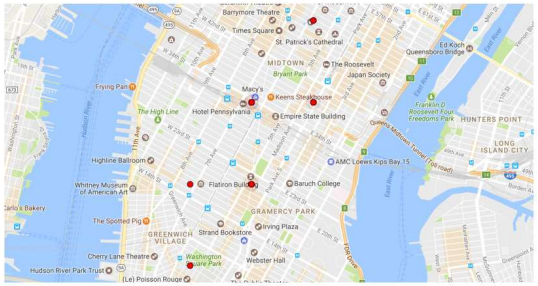
Phase 3

Identifying top 50 hotspots

- Applying spatial statistics to spatio-temporal big data in order to identify statistically significant spatial hot spots
- We have three java classes for this task.
  - MathUtility.java - deals with all the mathematical calculations for calculating Getis-Ord statistic, the class
  - Cell.java – POJO class that represents a cell.
  - HotSpots.java - handles the steps required to calculate Getis-Ord statistics and saves top fifty hot spot cells in the output file
- Each cell unit size is considered 0.01 * 0.01 as per the problem statement. The total number of cells is calculated as the product of (number of_days * latitude_range * longitude_range)

Phase 3

Hotspots marked on a map



Phase 3

After creating the space time cube, we calculated the z-score for each cell as per the formula for calculating the Getis-Ord statistics

$$G_i^* = \frac{\sum_{j=1}^{n} w_{i,j}x_j - \bar{X}\sum_{j=1}^{n} w_{i,j}}{S\sqrt{\frac{\left[n\sum_{j=1}^{n} w_{i,j}^2 - \left(\sum_{j=1}^{n} w_{i,j}\right)^2\right]}{n-1}}}$$

$$S = \sqrt{\frac{\sum_{j=1}^{n} x_j^2}{n} - (\bar{X})^2} \qquad \bar{X} = \frac{\sum_{j=1}^{n} x_j}{n}$$

- We collected the top 50 G score values which basically represents 50 top hotspots for pickup of Yellow Taxi in the month of January 2015

Phase 3- Implemented functions

Method name	Implementation details
collectData()	In this function, we use MapReduce on each record in the input file to return a list of tuples where, each tuple is in the format (latitude, longitude, day)
createSpaceTImeCube()	In this function, we create 3-D array of the space time cube using the list of tuples that was returned by the collectData() function
getOrdisNumerator()	In this function, we calculate the sum of spatial weight of neighbors for each cell in the space time cube and also the total attribute cost of all neighbors
getOrdisDenominator()	In this function, we calculate the sum of square of spatial weight of every neighbor for each cell in the space time cube and also the square of the spatial weight of all neighbors
computeOrdisValue()	In this function, we calculate the z-score for each cell and save it in a priority queue
getTopkHotSpots()	In this function, we call the above mentioned methods to obtain the priority queue which contains the z-score for each cell in the space time cube
saveResults()	Once the z-score is calculated for each point, we create a priority queue and compare the z-score for each cell and store the latitude, longitude, day and z-score of the top 50 most significant cells in the output file provided by the user

Phase 3 - Results

Top 10 hotspots in the format *cell_x, cell_y, time_step, zscore*

1	40.75	-73.99	13	66.27568493387267
2	40.75	-73.99	14	65.83416700256043
3	40.75	-73.99	21	65.23760925425758
4	40.75	-73.99	12	64.56516153666243
5	40.75	-73.99	8	64.05531655395988
6	40.75	-73.99	15	63.701163363929155
7	40.75	-73.99	22	63.693730841163415
8	40.75	-73.99	7	63.53073692086209
9	40.75	-73.99	29	63.22756822910162
10	40.75	-73.99	20	62.96403965875844