# Implementation of Distributed Database System on Geospatial Data Using GeoSpark

Arizona State University

Namrata Nayak
nnayak3@asu.edu

Nimshi Venkat
nmeripo@asu.edu

Shrinivas Bhat
sbhat10@asu.edu

Varun Chandra Jammula
vjammula@asu.edu

## ABSTRACT

Since the early 1990s, there has been an explosive growth in the adaptation of internet. Due to this rapid growth, companies serve more customers than ever. This growth also led to the development of several companies that work solely on the internet. The need for revenue and competition led to the development of various new applications that include streaming, social networking, searching, e-commerce etc. Depending upon the requirements of the application, some applications generate a lot of data that is used later for analytics, some applications require a lot of data to serve the customers. The need to handle this data effectively and efficiently cannot be achieved through centralized database systems and thus needs a distributed environment. In this project, we set up a cluster of machines with Spark and Hadoop and implemented a practical everyday application to analyze how the system works in a distributed environment.

## General Terms

Algorithm, Measurement, Performance, Scalability, Run time, Memory Utilization, CPU utilization.

## Keywords

Apache Spark, Geospatial operations, GeoSpark, Ganglia, Apache Hadoop, Map Reduce, Cartesian product.

## 1. INTRODUCTION

We used GeoSpark (Jia Yu, 2015), a framework used for processing large-scale spatial data to execute the various spatial join queries. We implemented it on a cluster of machines with Hadoop (Hadoop, n.d.) and Apache (Apache Spark, n.d.) Spark.

This paper contains the analysis of tasks that we performed on the spark cluster using geospark API. The following tasks are included in this report.

1. Create GeoSpark SpatialRDD (Point RDD).
2. Spatial Range Query.
    a) Query the PointRDD.
    b) Build R-Tree index on PointRDD then query this PointRDD.
3. Spatial KNN Query:
    a) Query the PointRDD and find 5 nearest neighbors
    b) Build R-Tree index on PointRDD then query this PointRDD again
4. Spatial Join Query:
    a) Join the PointRDD using Equal grid without R-Tree index
    b) Join the PointRDD using Equal grid with R-Tree index
    c) Join the PointRDD using R-Tree grid without R-Tree index
5. Spatial Join Query using simple Cartesian Product algorithm
6. Apply spatial statistics to spatiotemporal big data to identify statistically significant spatial hot spots using Apache Spark (Data used and applied as per ACM SIGSPATIAL GISCUP-2016) (GISCUP 2016, n.d.)

## 2. ARCHITECTURE

The system consists of three parts:

- Hadoop File System(HDFS) to distribute the data among the machines in the cluster.
- Apache Spark is used to run the application on the cluster.
- GeoSpark is used to work on the Resilient Distributed Datasets(RDD) that are used in the application running on Apache Spark.

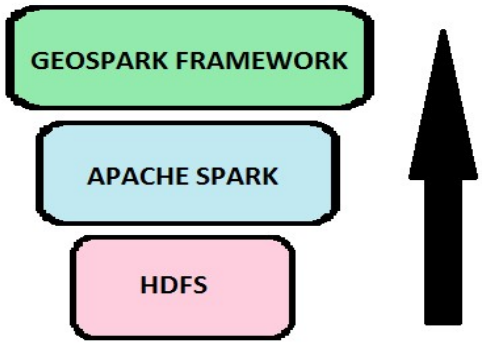


Fig 2.1 Overall Architecture

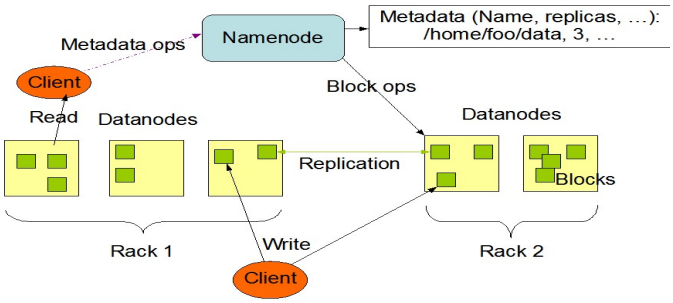


Fig 2.2 HDFS Architecture

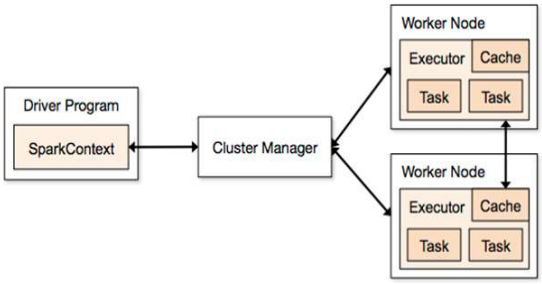


Fig 2.3 Spark Architecture

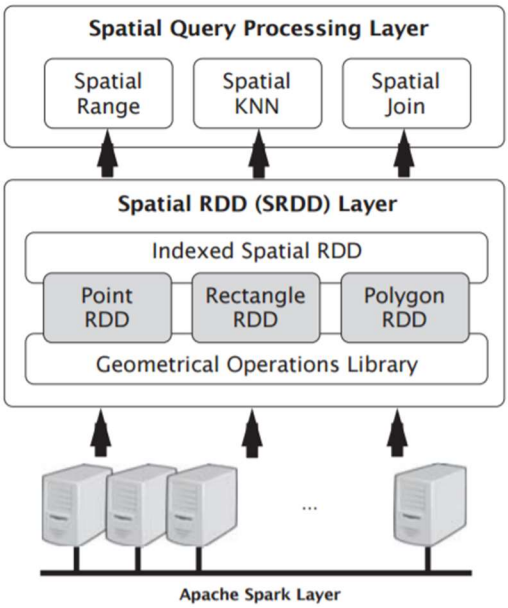


Fig 2.4 GeoSpark Architecture

a) **NameNode**: The NameNode is the centerpiece of an HDFS file system. It keeps the directory tree of all files in the file system, and tracks where across the cluster the file data is kept. It does not store the data of these files itself. (Hadoop Wiki, n.d.)

b) **DataNode**: A DataNode stores data in the HadoopFileSystem. A functional filesystem has more than one DataNode, with data replicated across them. (Hadoop Wiki, n.d.)

The Apache Spark cluster consists of 1 master and 3 slave nodes. We use GeoSpark with Apache Spark and run the application on the cluster. The application is submitted through spark-shell which runs on Scala. The input files and output files required by the spark program are usually loaded from or saved to Hadoop file system.

# 3. IMPLEMENTATION

The project implements distributed dataset operation using Apache Spark and GeoSpark. The project is divided into 3 phases. In each phase efficiency of different API's of GeoSpark and Apache Spark is measured. Distributed execution of queries on different GeoSpark specific RDD's such as SpatialRDD, PointRDD etc. with either indexing or without indexing is performed in phase 1.

Phase 2 will introduce a tool called Ganglia which is used to measure the performance of the cluster in the distributed system. A fair comparison between each task of phase 1 is done using performance recorded by Ganglia. Also, a spatial join query is performed using Cartesian product algorithm, which is used to compare the efficiency of joining PointRDD using Equal grid and R-tree grid with or without R-tree indexing.

## 3.1 Phase 1: GeoSpark Queries

In this phase, distributed execution of the GeoSpark queries is done using setup mentioned in section 4. The steps and the pseudocode of the queries we executed are as follows:

### *3.1.1 Create GeoSpark SpatialRDD:*

SpatialRDD is an extension of Apache spark RDD in a spatial dataset. Spark shell is used to load the GeoSpark jar and run the code. We create a SpatialRDD of geospatial coordinates that will be used later in different queries.

### *3.1.2 Spatial Range Query:*

A range query on the spatial data returns whether a given point is in the window created by a pair of points.

Pseudo Code:

- Given a pair of points of the window, create an Envelope.
- A PointRDD is created with the list of points that has to be range sorted.
- Perform RangeQuery using the Envelope and PointRDD object.
- Collect and aggregate the result in the collection as a list of points.
- We execute the same query with R-tree indexing.

### *3.1.3 Spatial KNN query:*

For every PointRDD object, we try to find nearest 'n' neighbors to the query point. The pseudocode is as follows:

- Create a point for which KNN has to be computed.
- Load PointRDD from the CSV file.
- Run Spatial KNN query to find 'n' nearest neighbors
- Aggregate the result.
- We execute the same query, with R-tree indexing.

### *3.1.4 Spatial Join Query:*

RectangleRDD is a pair of envelope and a PointRDD, on which we perform a spatial join query. This would give a list of points that are present in the envelope.

- Load PointRDD and RectangleRDD from the CSV files.
- Perform join operation on these two RDDs and aggregate the result.
- Perform the operations with and without R-tree indexing, on an Equal grid or R-tree grid object.

## 3.2 Phase 2: Performance measurement of different queries

In this phase, we developed an algorithm to compute spatial join query using Cartesian product algorithm. We also compared and analyzed the performances of each query of Phase 1 and Phase 2 using a monitoring system named Ganglia.

Ganglia is a scalable monitoring tool that can be used to analyze the performance metrics of nodes in a distributed cluster. Ganglia has a web interface that provides the user with memory, CPU utilization, execution time details of every node in the cluster. We use this interface to analyze the performance of different spatial queries.

### *3.2.1 Spatial join query using Cartesian product Algorithm:*

We implemented a spatial join query using a simple Cartesian product algorithm with the following algorithm.

1. Load the PointRDD and RectangleRDD from their respective datasets.
1. Collect all the envelopes from the RectangleRDD using rawSpatialRDD.
2. For every envelope in the list, do:
    a) SpatialRangeQuery on the PointRDD and the envelope.
    b) Collect all the points as a tuple.
3. Parallelize the collected data into JavaPairRDD and return it.

### *3.2.2 Performance analysis:*

For every task in Phase 1 and task 1 of Phase 2, we compared the execution time, average memory, average CPU utilization of the overall cluster using Ganglia. Setup for Ganglia is mentioned in section 4 and result of each of such comparison is explained in section 5.

## 3.3 Phase 3: Applying spatial statistics to spatiotemporal data

We applied spatial statistics to spatiotemporal big data to identify statistically significant spatial hot spots using Apache Spark.

The input data used is a collection of New York City Yellow Cab taxi trip records for January 2015. The source data is clipped to an envelope encompassing required data.

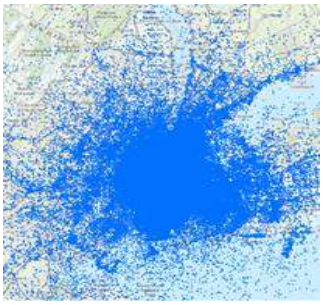

Fig 3.3 Map representing all the data collected based on location. The rectangle covers the area covered in the project.

### *3.3.1 Constraints:*

1. Each cell unit size is 0.01 * 0.01
2. The minimum and maximum geographical coordinates considered in this case are latitude 40.5N – 40.9N, longitude 73.7W – 74.25W
3. The program takes an input file and an output file as the parameters.

### *3.3.2 Algorithm:*

1. Create a three-dimensional cube for the input data
    a) Using mapreduce on each record in the input file we return a list of tuples where, each tuple is in the format (latitude, longitude, day)
    b) The total number of cells is calculated as the product of (number of_days * latitude_range * longitude_range)
2. Calculate the z-score:
    a) Calculate the ordis numerator: 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.
    b) Calculate the ordis denominator: Calculate the sum of square of spatial weight of every neighbor for each cell in the space time cube and the square of the spatial weight of all neighbors.
3. Calculate the z-score for each cell by dividing the numerator with the denominator and save it in a priority queue.
4. Sort the result to find the top 50 coordinates:
    a) 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.
    b) After creating the space time cube, we calculated the z-score for each cell as per the formula.

**Getis-Ord statistics ($G_i^*$)**

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

where,

$x_j$: attribute value for cell j

$w_{i,j}$: spatial weight between cell i and j

n: total number of cells.

The standard deviation (S) is calculated as below:

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

where,

Summation of $x_j^2$: sum of squares of all the points in the space-time cube

n: total number of cells

The mean ($\bar{X}$) is calculated as below:

$$\bar{X} = \frac{\sum_{j=1}^{n} x_j}{n}$$

where,

$x_j$: attribute value for cell j

n: total number of cells

# 4. EXPERIMENTAL SETUP

The following packages and technologies were used for the cluster setup:

- Open JDK 1.7.0
- Hadoop 2.7.3
- Spark 2.1.0
- SSH
- Maven 3
- Eclipse Neon IDE
- Ganglia

The cluster setup includes 1 master and 3 slave machines setup on Ubuntu desktop 14.04 LTS. The configuration of the machines are as follows:

- master – (39.25G, 1 core)
- namrata – (5.78G, 1 core)
- michael – (80.37G, 8 cores)
- varun – (16.61G, 1 core)

Ganglia setup was done using the tutorial from here (Ganglia Tutorial, n.d.).

Hadoop and Spark are installed on all the nodes under the same user and group "hadoop". Password less secure shell (SSH) login from master to every node has been established. Input files and data sets are stored in HDFS as mentioned above. The cluster was setup to run with 4 nodes (1-master and 3-slaves).

# 5. RESULTS

In this project, we conducted several experiments that either required executing a task and analyzing the output or analyzing the performance of the system during these computational tasks. For each of the phases implemented as mentioned in section 3, we produce the output or we compare the performance of the tasks.

## 5.1 Phase 1

In Phase 1, we set up the Hadoop and Spark environment and performed some spatial query operations using the GeoSpark framework on spark cluster.

## 5.2 Phase 2

We used Ganglia (Ganglia Monitoring System, n.d.) – to analyze the performance of cluster for tasks in Phase-1 and Phase-2 and these results were compared.

*5.2.1 Difference between Spatial Range Queries*

a. querying PointRDD
b. Build R-tree index on PointRDD and then query the PointRDD

| Iteration:150 | Execution Time | Average Memory | CPU Utilization (%) |
|---|---|---|---|
| 2a | 92ms | 13.8 G | 71.9 |
| 2b | 81ms | 13.9 G | 68.8 |

Tab 5.1 Performance metrics for query 2a, 2b

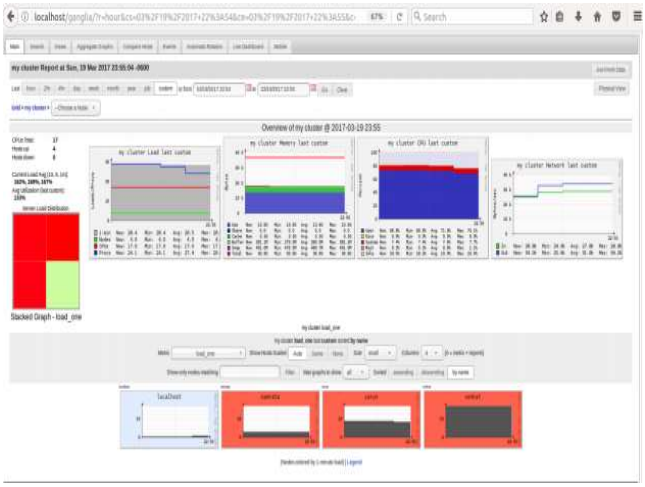

Fig 5.1 Cluster report of query 2a of Phase 1

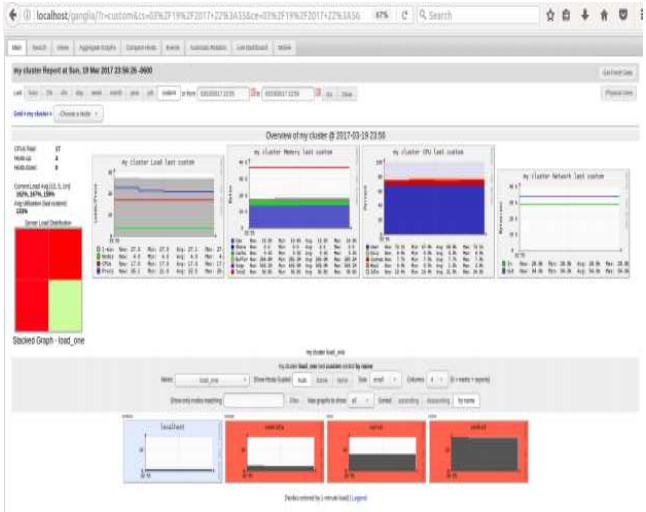

Fig 5.2 Cluster report of query 2b of Phase 1

*5.2.1.1 Conclusion:*

Execution time and CPU utilization were better with R-tree indexing while building PointRDD. And memory utilization was the almost same. Thus, we can conclude that R-tree indexing runs faster because it uses R-tree data structure which is ideal for spatial access methods. R-tree data structure groups nearby objects bound by a minimum rectangle.

*5.2.2 Difference between Spatial KNN Queries*

a. querying PointRDD and finding 5 nearest neighbors
b. Build R-tree index on PointRDD and then query the PointRDD again

| Iteration:150 | Execution Time | Average Memory | CPU Utilization (%) |
|---|---|---|---|
| 3a | 94ms | 13.8 G | 71.6 |
| 3b | 143ms | 14.2 G | 72.9 |

Tab 5.2 Performance metrics for query 3a, 3b

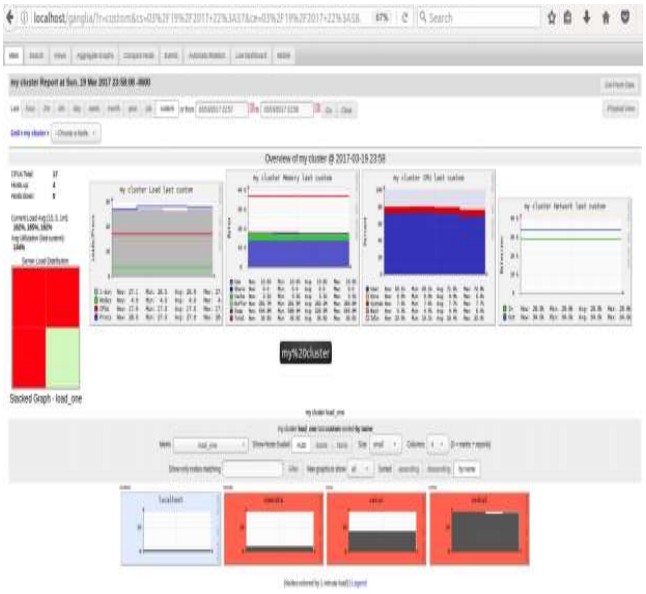

Fig 5.3 Cluster report of query 3a of Phase 1

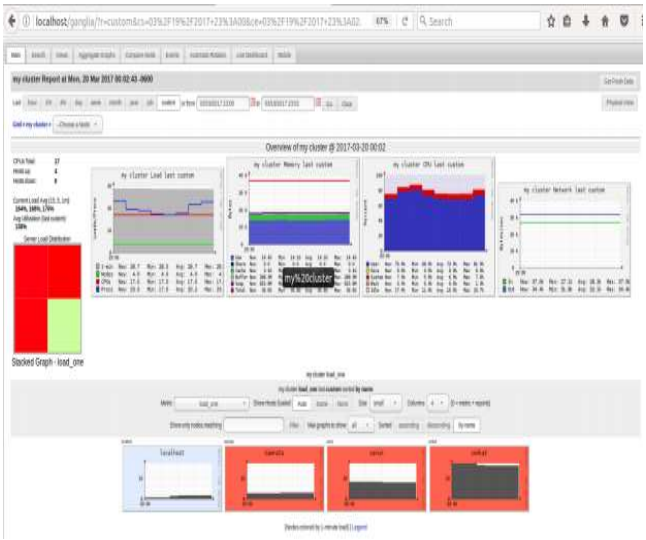

Fig 5.4 Cluster report for query 3b of Phase 1

*5.2.2.1 Conclusion:*

Execution time and CPU utilization were better without R-tree indexing while building PointRDD. Since finding nearing 5 neighbors becomes a burden with indexing hence performance was little poor.

*5.2.3 Difference between Spatial Join Queries:*

Create a RectangleRDD and join it with PointRDD

a. join PointRDD using Equal grid without R-tree index
b. join PointRDD using Equal grid with R-tree index
c. join PointRDD using R-tree grid without R-tree index and Task 1 of phase 2 i.e. Spatial Join Query using simple Cartesian product algorithm

| Iterations:10 | Execution Time | Average Memory | CPU Utilization (%) |
|---|---|---|---|
| 4a | 159163ms | 13.6 G | 74.5 |
| 4b | 45385ms | 13.9 G | 73.8 |
| 4c | 56589ms | 14.0 G | 76.2 |
| Phase 2 - T1 | 915393ms | 14.1 G | 83.0 |

Tab 5.3 Performance metrics for query 4a, 4b, 4c and Phase 2 Task 1

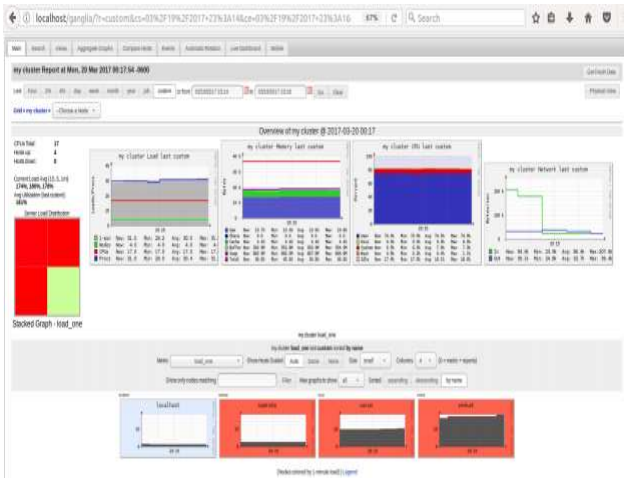

Fig 5.5 Cluster report of query 4a of Phase 1

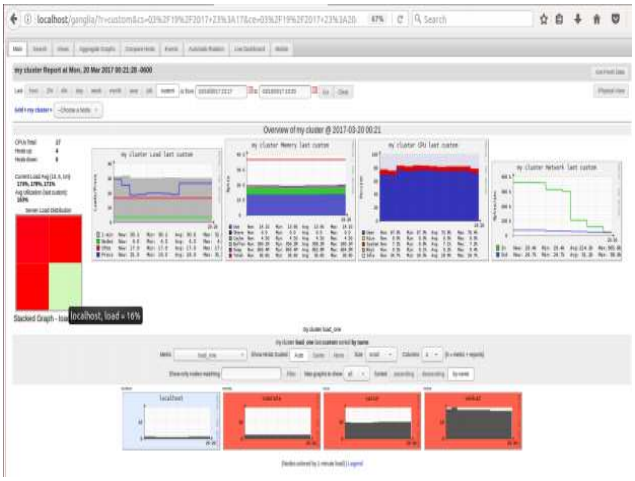

Fig 5.6 Cluster report of query 4b of Phase 1

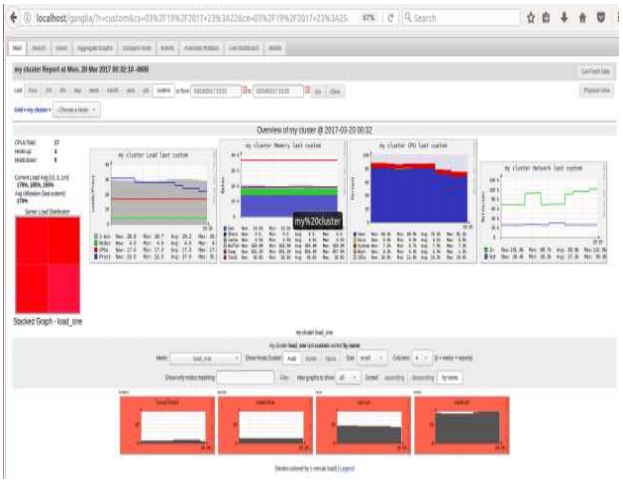

Fig 5.7 Cluster report of query 4c of Phase 1

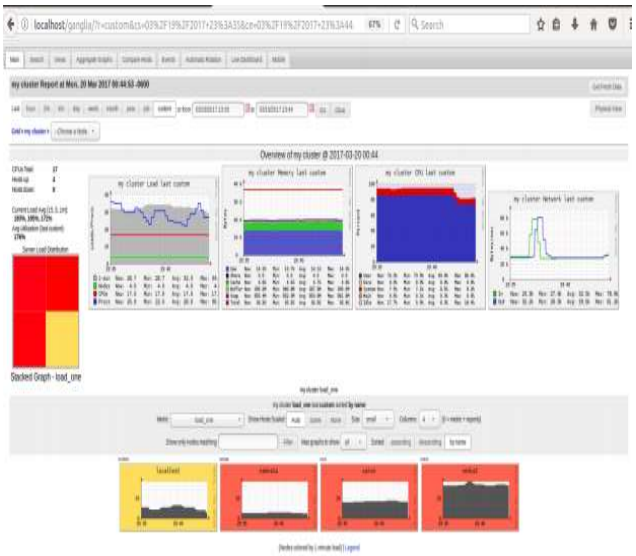

Fig 5.8 Cluster report of task 1 of Phase 2

*5.2.3.1 Conclusion:*

Execution time and CPU utilization in case of Equal Grid with R-tree indexing is better compared to without indexing. R-tree grid without indexing has better execution time and CPU utilization when compared to the Equalgrid with indexing.

Spatial join with simple Cartesian product algorithm (Task-1, Phase2) has the worst performance among all the algorithms.

## 5.3 Phase3

In this phase, we solved the GISCUP-2016 problem. The input file which contains the data set is a CSV file. The data is loaded into memory and then mapped based on the <X, Y, date>. After mapping is done, we reduce the key, value pairs by adding all the values with same key. Later, space time cube is created from the data obtained from reduce stage. After creating the space time cube, we compute the Getis-Ord value for each cell in the space time cube and determine the top 50 values and store them in a priority queue. The results for the top 50 hotspots are then saved to the output file provided by the user. The top 50 hotspots for pickup in the NY area for January are as follows:

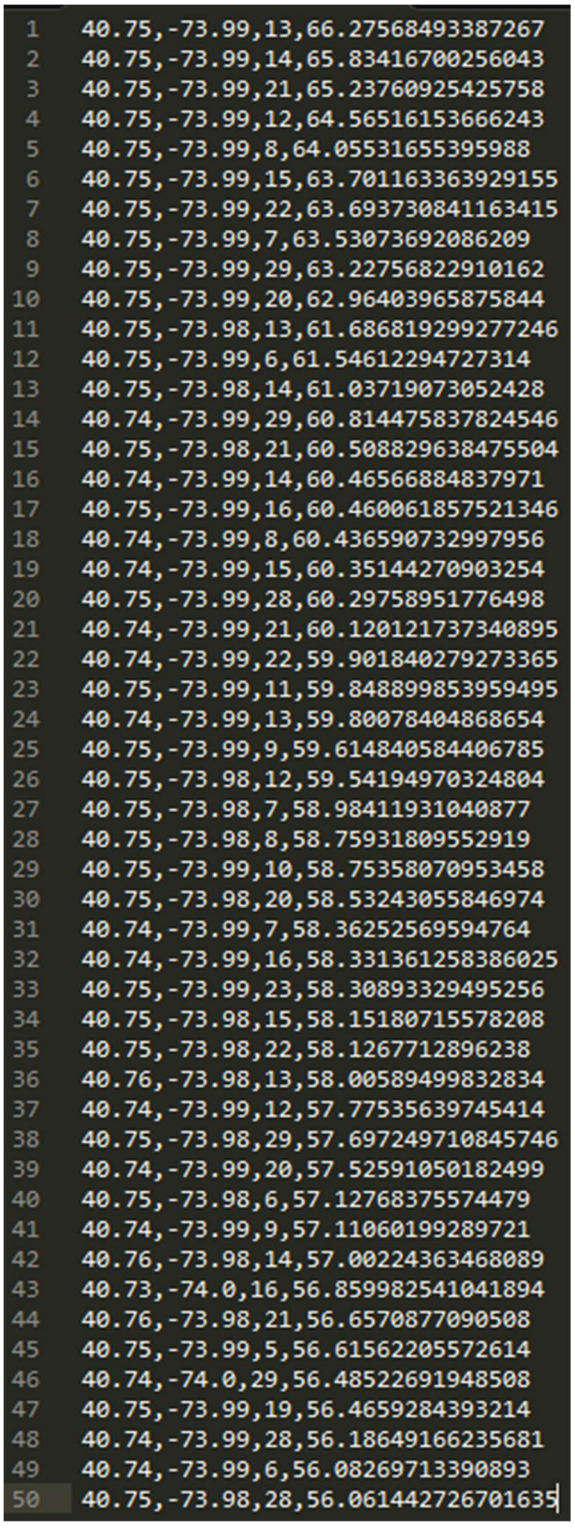


Fig 5.3 Top50 hotspots for Yellow Cab Taxi in NY in January of 2015

## 6. ACKNOWLEDGEMENT

We thank Dr. Mohamed Sarwat our course instructor for his help and guidance throughout the semester.

## 7. REFERENCES

*Apache Spark*. (n.d.). Retrieved from Apache Spark: http://spark.apache.org/

*Ganglia Monitoring System*. (n.d.). Retrieved from http://ganglia.sourceforge.net/

*Ganglia Tutorial*. (n.d.). Retrieved from Ganglia Tutorial: https://www.digitalocean.com/community/tutorials/introduction-to-ganglia-on-ubuntu-14-04

*GISCUP 2016*. (n.d.). Retrieved from GISCUP 2016: http://sigspatial2016.sigspatial.org/giscup2016/

*Hadoop*. (n.d.). Retrieved from Hadoop: http://hadoop.apache.org/

*Hadoop Wiki*. (n.d.). Retrieved from Hadoop Wiki: https://wiki.apache.org/hadoop/NameNode

Jia Yu, J. W. (2015). GeoSpark: A Cluster Computing Framework for. *Proceeding of the ACM International.*