Spatial Analysis on New York City taxi dataset using Apache Spark

Pawan Patil (1211173324), Jasmin George (1213387914),
Prit Sheth (1213203392), and Nagarjuna Kalluri (1212441735)

Abstract—In todays world, applications using Geo-Spatial data have found wide usage across all walks of life, be it in social sciences, geology, ecology, medicine, public safety, disaster management etc. Often, analyzing such data involves large amount of data processing. Such predicament requires the use of modern parallel and distributed data management systems. Here in this project, we are trying to implement an algorithm for hot-spot detection on Geo-Spatial data obtained from New York City Yellow Cab taxi trip records such that we get a hands-on experience on such distributed database management system.

Index Terms—Hadoop, Apache Spark, GeoSpark, MapReduce, Geo-Spatial Data, Parallel and Distributed Database Management Systems

1 Introduction

DATABASE management systems that can process large amounts of data at a faster rate are the need of the hour in todays world. Traditional database management systems are centralized and in spite of their obvious advantages in the form of low network latency, availability, they are very difficult to scale to be able to meet todays expectations. Also, they have a very low fault tolerance, are susceptible to data loss in case of any hardware failures make less desirable for large scale data processing. Instead, distributed systems divides and processes the data on various systems instead of running them on a single node. In doing so, they overcome the issues like scalability, reliability, availability and fault tolerance. Analyzing Geo-Spatial data in itself is a very difficult task due to its multidimensional nature. And analyzing large amounts of such data just becomes even more herculean a task. But distributed database management technologies such as Hadoop, Apache Spark etc make it relatively easy to analyze such Geo-Spatial data. Since Apache Spark cannot inherently handle Geo-Spatial queries, we use an extension for Apache Spark called as 'GeoSpark' [6] with a set of Spatial Resilient Distributed datasets (SRDDs) which helps in loading, processing and analyzing large scale spatial data across machines. In this paper, we integrate Geo-Spark with Apache Spark and Hadoop for running Geo-Spatial queries and then analyze and review them. *Phase I* deals with how the cluster was set up using Hadoop and Spark and running the existing GeoSpark functionalities. *Phase II* deals with creating user defined spatial range functions which can used for running various spatial range queries and lastly *Phase III* deals with coming up with an algorithm which can be used for hot-spot detection. Additionally, the paper also has various sections detailing the system architecture, experimental setup and experimental evaluations.

- *P. Patil, P. Sheth, J. George, and N. Kalluri are students at Arizona State University.*
- *E-mail: {ppatil6, jgeorg24, prsheth1, nkalluri}@asu.edu*

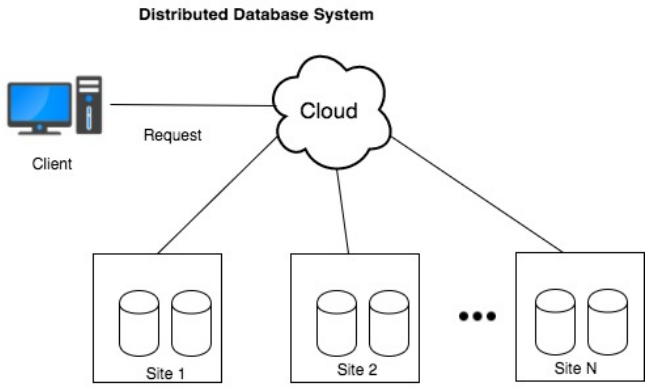

Fig. 1. Distributed Database System

2 System Architecture

This section explains the various distributed database management technologies which can be used for faster computation on large datasets.

2.1 Apache Hadoop

Apache Hadoop [1] is an open source distributed storage and computing software. It is known for its reliability, scalability and fault-tolerance. It provides high performance distributed computing on a cluster of computers and can handle both structured and semi-structured data. Its storage part is known as HDFS (Hadoop Distributed Storage System) which has a master/slave architecture and is a JAVA based file system. It normally consists of a single namenode and multiple datanodes. Namenode usually stores the META information such as names, number of blocks and details of all the datanodes in the cluster etc. where as datanodes store the actual data. Namenode is responsible for opening, closing, renaming files and directories. Generally there will be one datanode for each node in the cluster and they responsible for managing the storage associated with the nodes under them in the cluster. Another important

factor to be taken into consideration is the replication factor which is vital to Hadoop performance optimization. For our implementation, we have used *Hadoop 2.6.5* [2] and set one namenode and 3 datanodes in our cluster.

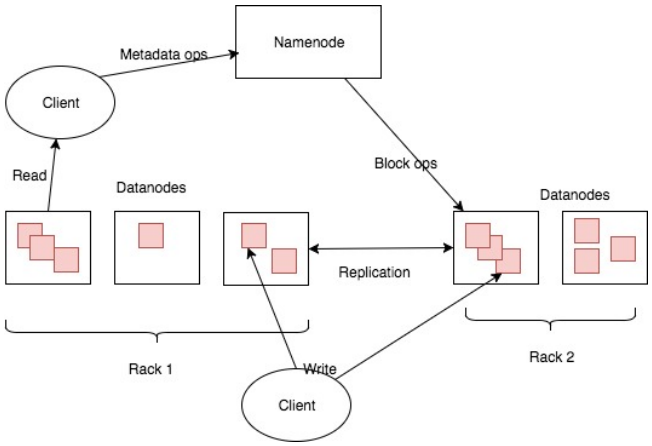

Fig. 2. Hadoop Architecture [1]

2.2 Apache Spark

Apache Spark is an powerful open source cluster computing framework used for faster computation. It is extends Hadoop MapReduce to include more types of computations like stream processing and interactive queries. Unlike Hadoop, Spark uses in-memory cluster computing which increases the applications processing speed when using large datasets. It also provides an interface for programming cluster with implicit parallelism. The fundamental data structure used is called an RDD (Resilient Distributed Datasets) which is set of immutable distributed objects. Since RDDs are immutable, the old results are stored as it is along the newly modified ones. RDDs reduce network and I/O costs. Iterative operations in Spark are supported by allowing users to access datasets multiple times in a loop and Interactive operations are supported by allowing users to store intermediate results on distributed storage.

RDD primarily supports two types of operations:

- Transformation: map, flatMap, filter, reduceByKey, groupByKey, aggregateByKey
- Action: reduce, count, collect, foreach, countByKey.

Structured and semi-structured data are supported by Spark using a data abstraction called as Data frames. For our implementation, we have used *Apache Spark 2.2.0* [4].

2.3 Scala

Scala is JAVA based scalable programming language. Everything in scala is an object. It supports both functional and object oriented approaches.Since it is based on Java, its executables can be run on Java Virtual Machine (JVM). Its flexibility and ability to use existing java libraries makes it efficient to use for scalable, distributed and parallel processing. In our implementation, we have used Scala for algorithm implementations.

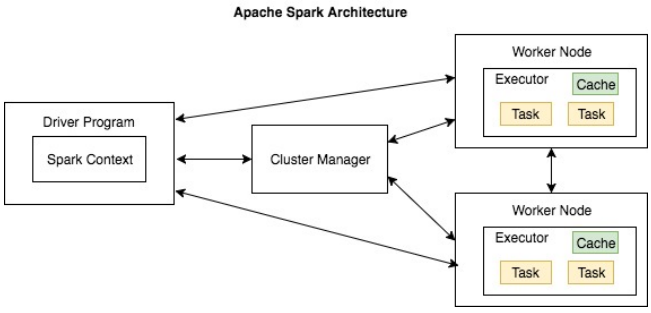

Fig. 3. Spark System Architecture [3]

2.4 Ganglia

Ganglia is a monitoring system for high performance computing systems. Ganglia is scalable and distributed and it works perfectly for distributed computing systems like clusters. [5] The software is used for analyzing either live or recorded statistics like CPU load averages or network utilization for multiple nodes as well as individual node. In our implementation, we have Ganglia to monitor various spatial operations done on the cluster and the results have been represented in the graphs.

3 Geo Spatial Analysis

Geo-Spatial analysis is used for applying statistical analysis on spatial data. It has become more prominent in the recent years to have accurate analysis of spatial data for various industries ranging from medicine, disaster management, crisis management, public safety, sales analysis, natural resources, ecology, geology, weather monitoring etc. But, processing Geo-Spatial information is not that easy due to the unstructured and multidimensional nature of its data. Hence, running this kind of data on a centralized system might prove a daunting task, but with modern distributed computing systems, it is very much manageable. Apache Spark can handle the sheer volume of such data, but it cannot inherently handle Geo-Spatial queries on its own. For this reason, we use an extension for Apache Spark called as Geo-Spark to run our spatial queries.

In our implementation, we run spatial queries on New York City taxi data to find out regions of the city which had a very high frequency of taxi pickups during a set period of time.

3.1 GeoSpark Architecture

GeoSpark is an in-memory cluster computing framework. It is built on top of Apache Spark and can handle large volumes of Geo-Spatial data.

GeoSpark has 3 layers,

- *Apache Spark layer* - It provides basic Spark functionalities like loading data and regular RDD operations
- *Spatial RDD layer* - It provides new RDD objects to support geometrical and spatial objects. The three new RDD are,
  - *PointRDD*
  - *Rectangle RDD*
  - *PolygonRDD*

- *Spatial Query Processing layer* - It executes spatial queries with the support of spatial indexing (e.g. R-Tree) on the given data.

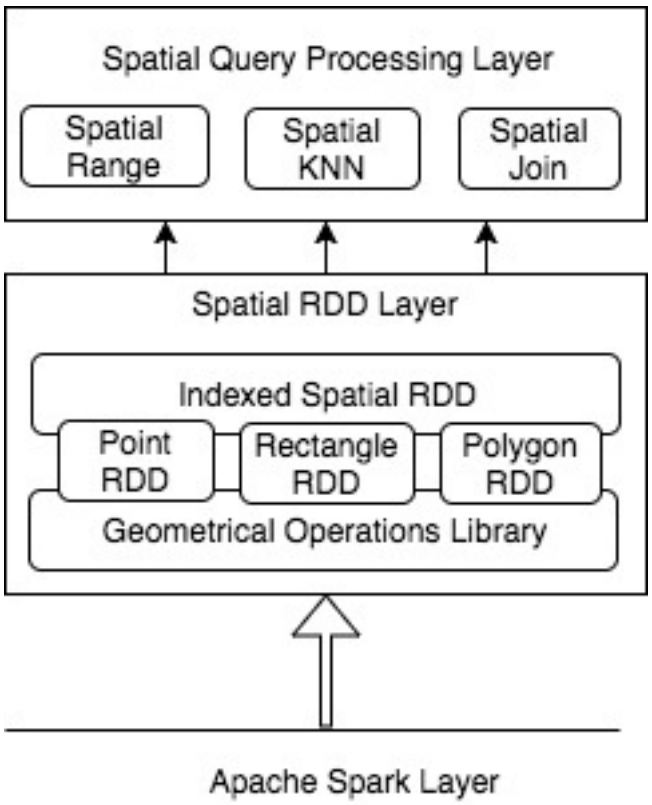

Fig. 4. Geo Spark Overview

A Spatial query is a special type of query that can be run on geographical and spatial data. It allows us to use different kinds of geometrical data types such as point, rectangle, line and polygon etc. The various queries executed are,

4 Project Phase 1

In this phase, we explored the GeoSpark framework using a cluster setup with Apache Spark and Hadoop. The experimental setup included three virtual machines running on Ubuntu 16.04, one master and two workers. Each machine was configured with Apache Spark and Hadoop. Bi-directional passwordless SSH was enabled between these machines and localhost so that the nodes can communication with each other.

TABLE 1
System Configuration

Machine	Processor	Disk Space	RAM
Master	2.8 GHz	15 GB	4 GB
Worker 1	2.8 GHz	15 GB	4 GB
Worker 2	2.8 GHz	15 GB	4 GB

With the above mentioned configurations, Spark and Hadoop were started on each machine. The point dataset and the rectangle query dataset was preloaded to HDFS. The GeoSpark jar was loaded into the Apache Spark scala shell of master node. The operations of GeoSpark like the Create GeoSpark SpatialRDD (PointRDD), Spatial Range Query, Spatial KNN query and Spatial Join query were executed from the scala shell and the results were analysed. This phase helped in understanding the concepts of distributed database and spatial queries.

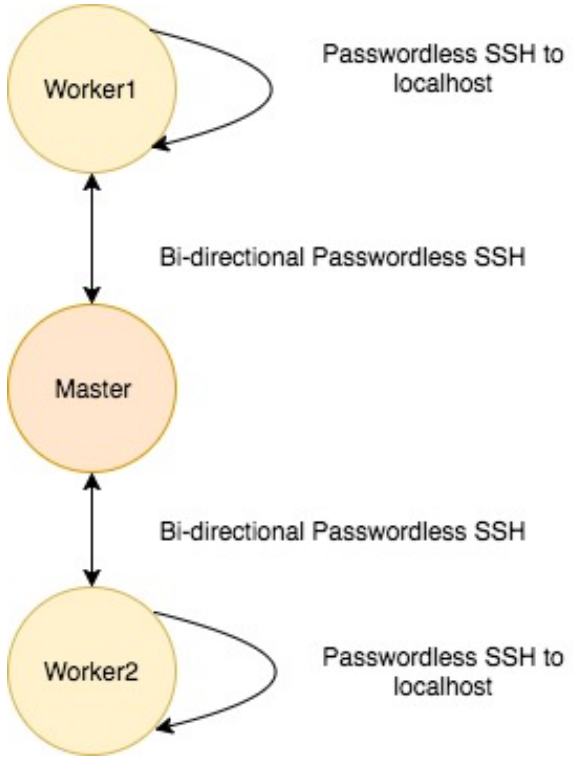

Fig. 5. Password-less SSH Topology

5 Project Phase 2

In this phase, we implemented two user defined functions for running four spatial queries - Range query, Range join query, Distance query and Distance join query. The two user defined functions are ST_Contains and ST_Within. The implementation details of these functions are

ST_Contains : This function takes as input a pointString(single coordinate) and a queryRectangle(two coordinates of recangle space). This function checks whether the point is within the given rectangle or not. The logic implemented was to check whether the x and y coordinate values of the point given was within the x and y coordinate values of the rectangle points.

ST_Within : This function takes as input two pointStrings(single coordinate) and a distance. This function checks whether the two points are within the given distance. The distance between the points were calculated using the Eucledian distance.

5.1 Range query

Given a query rectangle R and a set of points P, range query finds all the points within R. Range query uses ST_Contains to find the points within the rectangle.

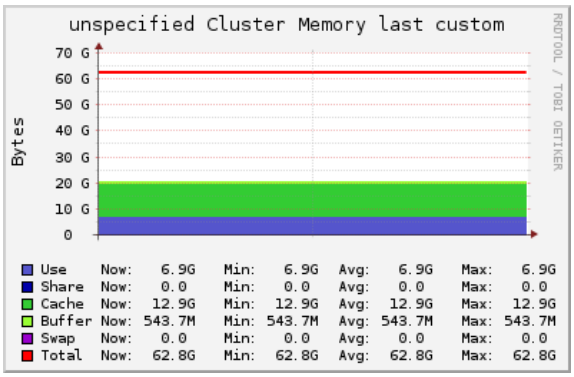

Fig. 6. Range query Cluster CPU Usage

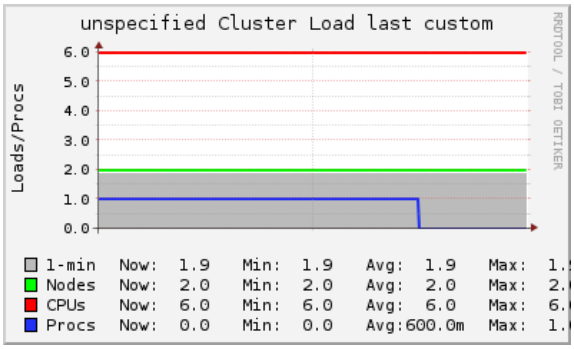

Fig. 7. Range query Network usage

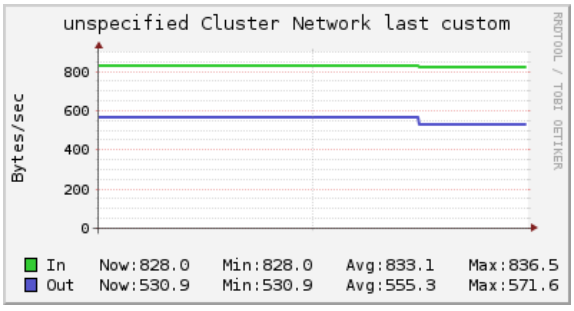

Fig. 8. Range query Server Load

5.2 Range join query

Given a set of Rectangles R and a set of Points S, range join query finds all (Point, Rectangle) pairs such that the point is within the rectangle. Range join query uses ST_Contains to find the points within the rectangle.

5.3 Distance query

Given a point location P and distance D in km, distance query finds all points that lie within a distance D from P. Distance query uses ST_Within to find the points which are within the given distance.

5.4 Distance join query

Given a set of Points S1 and a set of Points S2 and a distance D in km, distance join query finds all (s1, s2) pairs such that s1 is within a distance D from s2 (i.e., s1 belongs to S1 and s2 belongs to S2). Distance join query uses ST_Within to find the points which are withing the given distance.

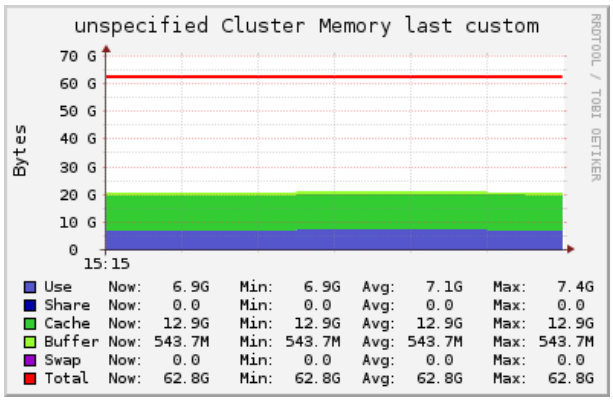

Fig. 9. Range join query CPU Usage

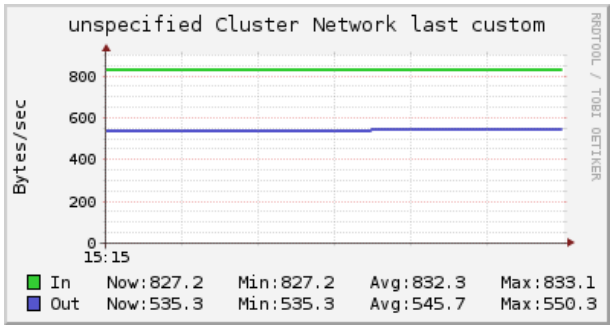

Fig. 10. Range join query Network Usage

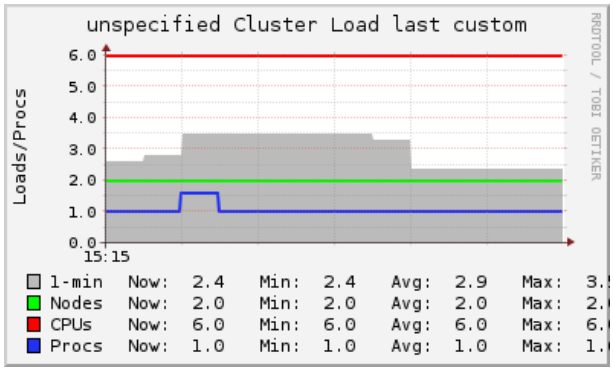

Fig. 11. Range join query Server Load

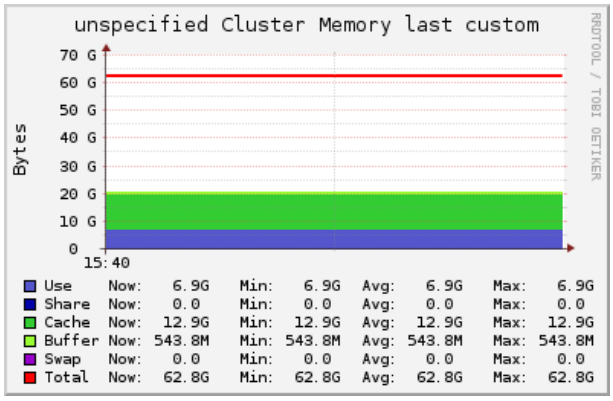

Fig. 12. Distance query CPU Usage

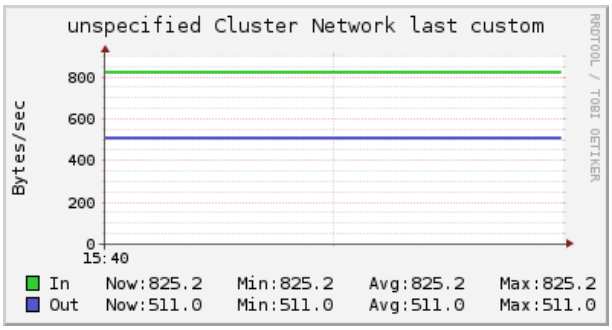

Fig. 13. Distance query Network Usage

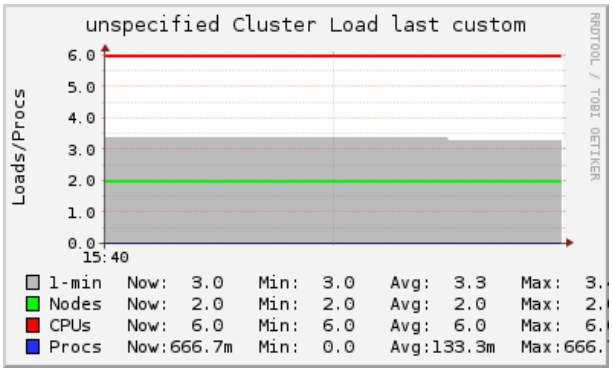

Fig. 14. Distance query Server Usage

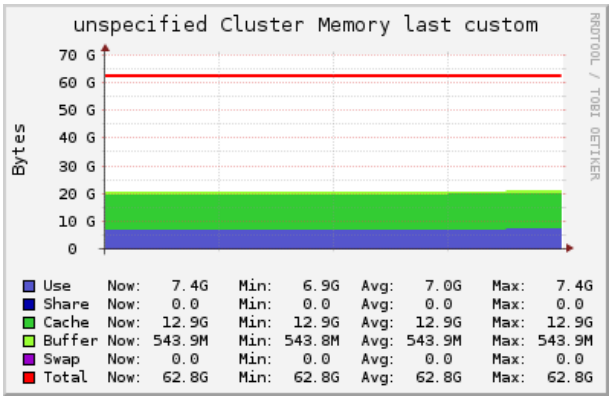

Fig. 15. Distance join query CPU Usage

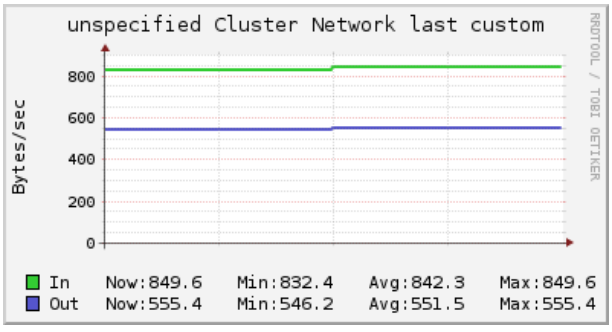

Fig. 16. Distance join query Network Usage

6 Project Phase 3

In this phase, we have taken a real world problem from GISCUP 2016, where we have taken the Geo-Spatial data

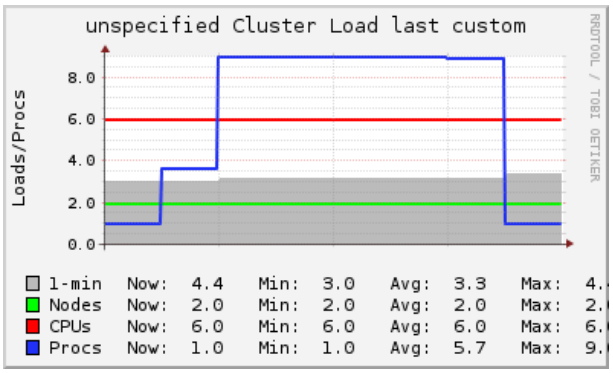

Fig. 17. Distance join query Server Usage

from New York City taxi data and have done Hot spot analysis. In this phase, we have done 2 Hot spot analysis tasks,

- Hot Zone Analysis
- Hot Cell Analysis

6.1 Hot Zone Analysis

In Hot Zone analysis, we are basically looking for rectangles with maximum number of points. The more the points ,the hotter the rectangle. We have implemented a spark program to analyse the spatial data and give the hot zones. The algorithm used for our program is as follows,

Algorithm 1: Hot Zone Analysis

Input: Spatial data
Output: Hot Zones

```
1. Process all the points and run
the ST_CONTAINS function to check
whether it is valid point or not

2. For sorting all the points we
will run the group by and order by
sql query on processed cell points
which will give us the sorted cell
points.
e.g: joinDf.orderBy("rectangle")
       .groupBy("rectangle").count()
```

TABLE 2
First 10 Hot Zones

cell_x1	cell_y1	cell_x2	cell_y2	No. of points
-73.789411	40.666459	-73.756364	40.680494	1
-73.793638	40.710719	-73.752336	40.730202	1
-73.795658	40.743334	-73.753772	40.779114	1
-73.796512	40.722355	-73.756699	40.745784	1
-73.797297	40.738291	-73.775740	40.770411	1
-73.802033	40.652546	-73.738566	40.668036	8
-73.805770	40.666526	-73.772204	40.690003	3
-73.815233	40.715862	-73.790295	40.738951	2
-73.816380	40.690882	-73.768447	40.715693	1
-73.819131	40.582343	-73.761289	40.609861	1

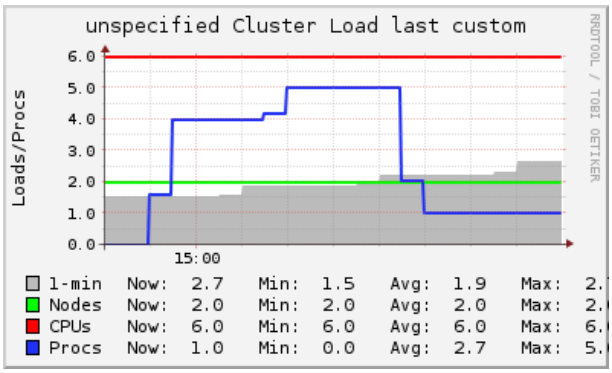

Fig. 18. Hot Zone Analysis - Server Usage

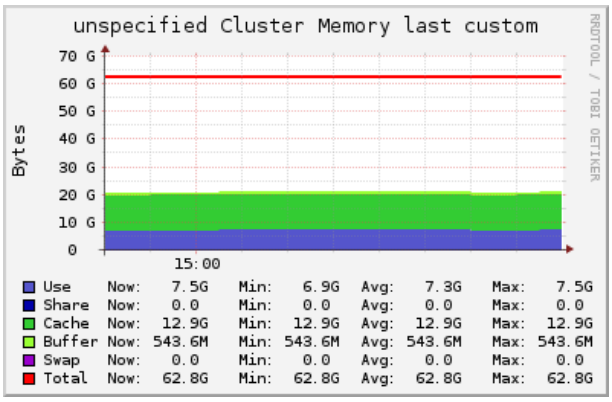

Fig. 19. Hot Zone Analysis - CPU Usage

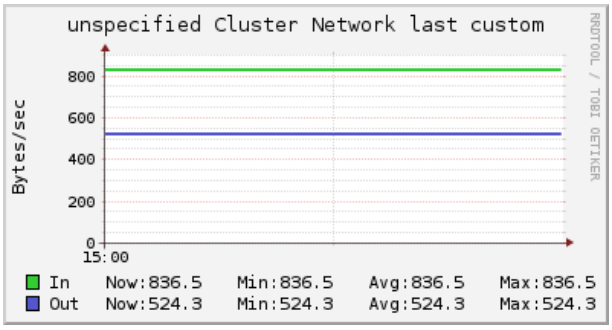

Fig. 20. Hot Zone Analysis - Network Usage

6.2 Hot Cell Analysis

In Hot Cell analysis, we take the New York city taxi trips data and find the top 50 hot spot cells in time and space. For finding such cells, we have implemented a spark program to calculate the Getis-Ord statistic of NYC taxi trip data.

Assumptions:

To calculate the Getis-Ord statistic, we have made the following assumptions,

- Every point in the 3-D space corresponds to a record in the input dataset.
- Latitude and longitude of pick-up locations is used for the grid of cells along x and y axis
- Size of each cell is 0.01 x 0.01 in terms of latitude and longitude

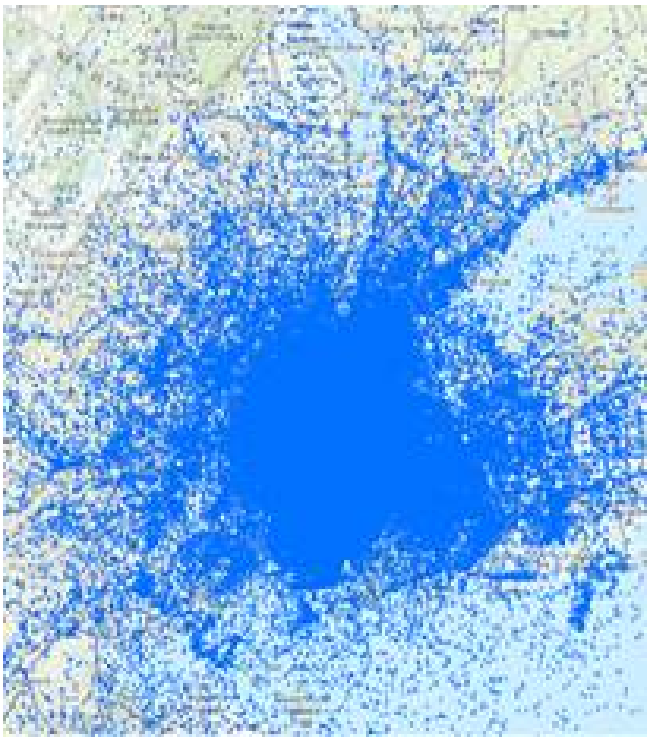

Fig. 21. The dataset of New York Yellow Taxi January 2015 and the boundary around the dataset

Fig. 22. The boundary around the dataset

- A time step of 1 day is taken and time forms our z-axis
- Each month will have 31 days and the time is aggregated accordingly with a step value of 1 day into a total 31 values on the time-axis.

- In order to avoid some noisy error data, the source data is clipped to cover only the five New York City within the range of latitude 40.5N- 40.9N, longitude 73.7W-74.25W.
- Each cell in the 3-D space has 26 neighbors and the cell itself is also considered its own neighbor as shown in the following figure.

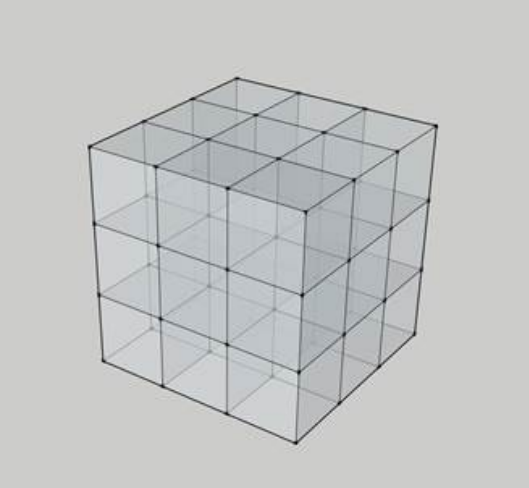

Fig. 23. The 26 neighbors of a cell in the 3-D space with the cell itself also its neighbor

The Getis-Ord statistics provide us values like z-score and p-values. These values help us determine if the object is statistically important or not. Pickup date, time, drop off date, time, location (latitude, longitude), trip distance, passenger count and fare count are some of the key information present in each row of the data set.

Getis-Ord, Gi* statistic is computed using the following formula:

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

where
x_j = attribute value of cell j,
$w_{i,j}$ = spatial weight between cell i and j,
n = total number of cells,
Mean:

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

Standard deviation:

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

6.3 Results

The top 10 hot spots obtained by running the experiments on the data are shown in Table 3

Algorithm 2: Hot Cell Analysis

Input: New York City Taxi Dataset (approx. 2.5 GB) with every record containing pick-up latitude and longitude and other data etc.
Output: Top 50 most significant pick-up locations in both time and space using Getis-Ord Gi* Statistic

```
1)  Process the input x,y,z
2)  Creating isValid function to
    check the point resides in
    given longitude and latitude.
3)  Run the spark sql query :
    "select x,y,z,count(*) as count
    from allpoints where isValidPoint
    (" + minX + "," + maxX + "," +
    minY + "," + maxY + "," + minZ
    + "," + maxZ + ", x,y,z) group by
    x,y,z")
4)  We will have the count for each
    x,y,z cell
5)  Calculate the mean and standard
    deviation using this count.
6)  Create the countMap with key as
    string x,y,z and value as count :
    We aggregate these map values
    using reduce operation ignoring
    Outliers(not a valid points). This
    gives a total count of each unique
    cell id. This will help to find
    the hot cells. The output is
    created in the format:
    [X, Y, time step] : [totalCount].
7)  We will iterate over all the cell
    value and call the
    getGetisOrdStatisticValue
    function.
8)  For calculating the Getis score
    we will see all its neighbor and
    their count from our populated
    map. If neighbor is the valid one
    isValid(point) then we will add
    the count of it to neighbor list.
9)  We will use this neighbor list to
    calculate the Getis-Ord Score
10) Populate the zscoremap
11) Sort the map according to the
    value of zscore
```

Fig. 24. Hot Cell - Algorithm

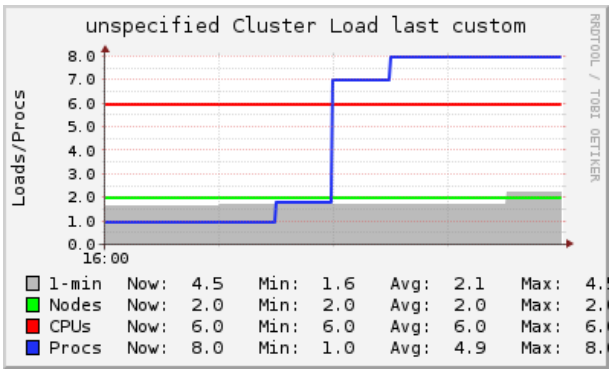

Fig. 25. Hot Cell Analysis - Server Usage

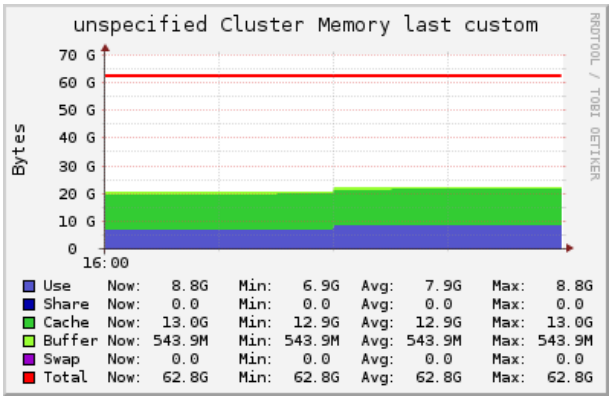

Fig. 26. Hot Cell Analysis - CPU Usage

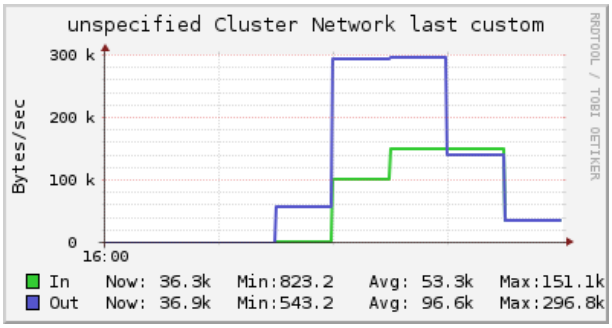

Fig. 27. Hot Cell Analysis - Network Usage

TABLE 3
Top 10 Hot Cells

cell_x	cell_y	time_step	zscore
40.75	-73.99	15	78.47816293025222
40.75	-73.99	29	77.73259524512093
40.75	-73.99	22	76.87066891201765
40.75	-73.99	28	76.82029842111804
40.75	-73.99	14	75.59086448402098
40.75	-73.99	30	75.26669238616431
40.75	-73.98	15	75.01413643319005
40.75	-73.99	23	74.95757515569942
40.75	-73.99	16	74.88258221813102
40.75	-73.98	29	74.06624258628882

7 Experimental Results

Varying Cores: We tried evaluating the performance of our implementation for Hot cells and Hot zones by varying the size of the number of cores. From our experiment, we observed that the execution time decreased whenever we increased the number of cores. The results above clearly show that by scaling the number of cores, the execution time decreases. This is due to the fact that by accommodating more cores for analysis, the data can be distributed across more cores and hence analysis can be done at a much faster rate

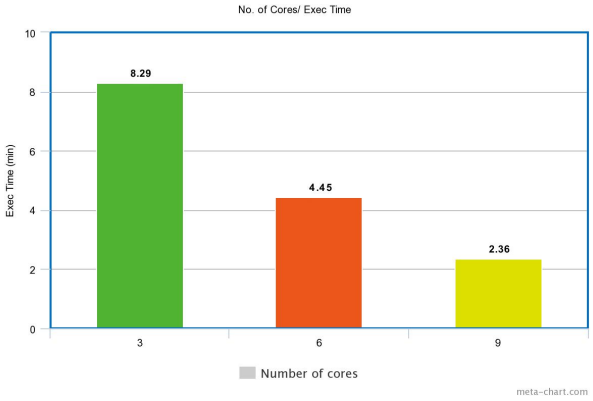

Fig. 28. Performance of Hot cell and Hot zone implementation for varying number of cores

Varying Iterations: We also tried evaluating the performance of our implementation for Hot cells and Hot zones by varying the number of iterations we run our implementation. For our experiment, we have used 3 cores and observed that the execution time decreased exponentially when we increased the number of iterations we run the code for our implementation. The results in the figure 29 clearly show that by increasing the number of iterations we run our implementation, the execution time decreases. This is because of the in-memory computation of Spark. i.e. the intermediate result data is stored on the cluster cache instead of a disk system. This will greatly reduce the time because, when run the same experiment again, the intermediate results on the HDFS cluster can used for analysis.

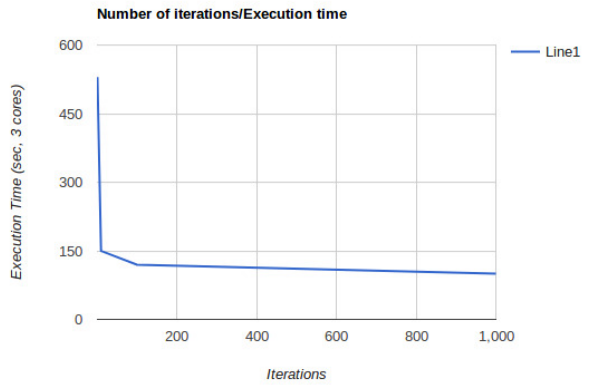

Fig. 29. Performance of Hot cell and Hot zone implementation for varying number of cores

8 Conclusion

By implementing this project, we have gained valuable experience in setting up a Distributed Database system for handling very large volumes of data. We got to know the nuances of how a distributed database system operates and practical knowledge of handling such huge volumes of data. We got to know how Hadoop and Spark work, how we can configure MapReduce to run tasks on it and how we can use GeoSpark in running spatial queries on GeoSpatial data. This course provided us an ideal platform in learning and understanding the basic concepts of Distributed Database systems.

9 Acknowledgement

We would like to thank Prof. Mohamed Sarwat and Yuhan Sun for their valuable inputs throughout the duration of this project. We would also like to thank them for their encouragement and support in helping out with this project. And lastly, we would like to thank and appreciate all the members of our team GLINT for their hard work and insight in contributing to the completion of this project.

References

[1] *"Apache Hadoop"*. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.
[2] *"Apache Hadoop 2.6.5"*. http://mirror.olnevhost.net/pub/apache/hadoop/common/hadoop-2.6.5.
[3] *"Apache Spark"*. https://spark.apache.org/docs/latest/cluster-overview.html.
[4] *"Apache Spark 2.2.0"*. https://spark.apache.org/downloads.html.
[5] *"Ganglia"*. http://ganglia.info/?page_id=66.
[6] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 70. ACM, 2015.