CSE 512 Project: Geo Spatial operations using Apache Spark and Hadoop

Shailee Desai
Arizona State University
1210936321
smdesai2@asu.edu

Abhishek Pharande
Arizona State University
1211213793
abhishek.pharande@asu.edu

Nishtha Punjabi
Arizona State University
1211226117
npunjab1@asu.edu

Amit Kumar
Arizona State University
1211257070
akuma152@asu.edu

ABSTRACT

Data is going on increasing day by day and thus the data being processed has also increased in huge folds. All this is due to the widespread use of computers in every field. Since such large amounts of data must be processed we require new technologies to handle big data and do its processing. One type of such data is geospatial data. This data contains data points with geographical positioning which is used in this project. So, we make use of GeoSpark which is an extended version of the Apache Spark to process such geospatial data. To do this, we need to upload the data into the Hadoop Distributed File System which is read using Scala. We then run the query on PointRDD data. As we proceed spatial queries are performed using user defined functions in Scala SparkSQL to check if the point lies in the given rectangle. In the last phase we completed we completed a modified version of the GISCUP 2016 challenge. The project culminates on analyzing location based data to find hot spots for taxi pickups in space and time. Spatial hot spots are located by using Apache Spark on HDFS to process the given dataset. Hot zone analysis performs a range join operation on a rectangle dataset and a point dataset. For each rectangle, the number of points located within the rectangle will be obtained. The hotter rectangle means that it includes more points. So, this task is to calculate the hotness of all the rectangles. And finally, we analyze all the results obtained from all the tree phases in phase 4.

Keywords

Distributed Database System (DDS), Geospatial operations,

Resilient Distributed Datasets(RDD), Hadoop

distributed file system (HDFS) , GeoSpark,

Apache Spark, Join Query, Cartesian

Product, z score.

Terminology

1.1 GeoSpark: GeoSpark is a cluster computing framework. It extends Apache Spark with a set of out-of-the-box Spatial Resilient Distributed Datasets (SRDDs) that efficiently load, process, and analyze large-scale spatial data across machines. GeoSpark provides APIs for Apache Spark programmer to easily develop their spatial analysis programs with SRDDs which have in house support for geometrical and distance operations [1]

1.2 Spatial Range Query: Given a query object q, a distance threshold and a dataset D, a spatial range query returns a set of objects in D that is at most distant from the query object q [2]

1.3 Spatial KNN Query: Given a query object q and a dataset D, a k-nearest neighbor query returns the k closest spatial objects to query q [2]

1.4 Spatial Join Query: Given a distance threshold, a similarity function that returns how similar two objects are, and two datasets D 1 and D 2, a join query returns pair of objects from D 1 and D 2 that are at least like each other [2]

1.5 Indexing Structure: A data structure that speed the search operations on a database. Creating and Maintaining an index structure has an associated cost of extra writes and storage space. This cost is amortized at the search phase, where querying using an index is faster than searching every row in a database. [2]

1.6 Spatial Index: A spatial index is a specialized indexing structure where the indexing key is the spatial location of objects indexed. The type of searches on a spatial index is the set of spatial queries, for example, range and overlapping queries [2]

1.7 R-Tree: R-trees are tree data structures used for spatial access methods, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons. [13] When data is organized in an R-tree, the k nearest neighbors (for any L_p-Norm) of all points can efficiently be computed using a spatial join [4]

1.8 Spatial Partitioning: GeoSpark supports R-Tree **(**GridType.RTREE**)** and Voronoi diagram **(**GridType.VORONOI**)** spatial partitioning methods. Spatial partitioning is to repartition RDD according to objects' spatial locations. Spatial join on spatial partitioned RDD will be very fast. [5]

1.9 Grid type: Grid type is a spatial partitioning method. We used three types of grid types namely Equalgrid, R-tree, Voronoi

1.10 Getis Ord statistic (G_i*): Getis-Ord statistic is a computation that provides locally significant points i.e points where there is higher or lower frequency values are clustered spatially. Getis-Ord statistic provides z-score and p-value based on which the points of interest are found. This statsistic not only considers the points but also its neighbors in calculating the hotness or z-score of the point. [8]

The value of Getis-Ord (G_i^*) statistic is calculated using the following formulae

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

where x_j is the attribute value of the cell j, $w_{i,j}$ is the spatial weight between cell i and j, n is equal to the total number of cells and:

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

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

1. INTRODUCTION

Big data has gained a lot of prominence mainly because it can handle huge amount of data. Nowadays a lot of data is being generated both structured and unstructured. This large amount of data will be of no use if we cannot extract the useful information and determine the various trends and patterns. Big data is very useful because it does not just process large amount of data but also makes sense of that data in determining the business trends which will help any company stay ahead in the market and so a lot of research is being done in this field and it has gained a lot of popularity. The data companies need to leverage vast amount of historical as well as real time data and big data can help in processing data real time as well.

There are two types of systems available centralized and distributed. In centralized systems the computing is done at a central location using terminals attached to a central computer. This system offers greater security than decentralized system as all the data and processing is done at a central location. Also, the availability increases because if one system is down then the user can log onto another system and access all his data. Depending on the machine the user can also resume his session from where it was last left. However, it does have some disadvantages as the centralized server controls all the remote terminals and if the central server fails then the entire system will go down. Distributed computing is where the components are shared among multiple computers to improve the efficiency and performance. The computers in distributed system can be close together connected by a local network or they can be geographically distant and connected by a wide area network. Some of the advantages of distributed system is availability, reliability and scalability. Distributed systems are transparent and though the tasks are distributed for the end user it is a single interface. Another important advantage is redundancy several machines can provide the same service, so if one is unavailable the work does not stop. These machines are small and thus inexpensive. However, in case of distributed system also there are disadvantages like security problems due to sharing, some messages might be lost in the network especially for big data, Bandwidth is another problem especially when the data is huge and thus all wires need to be replaced which is very expensive. The database in distributed system is difficult to manage compared to the single system. Overloading is another problem. If database is connected to a local system and many users are accessing remotely or in a distributed way, then the system can become slow.

To deal with these challenges, new software frameworks to multithread computing tasks have been developed. These frameworks are designed to get their parallelism from computing clusters not from any supercomputer. These clusters are large collection of commodity hardware which include conventional processors connected with ethernet cable and inexpensive switches. These have a new type of file system called distributed file system. Thus, MapReduce on Hadoop platform is used for this purpose. They provide features like load balancing, fault tolerance, and locality aware scheduling. Since all the predecessors clustering components were based on acyclic data flow models, they cannot be used to express a wide variety of problems. So, [3] proposed Resilient Distributed Dataset (RDD) to overcome this issue. Apache Spark uses Resilient Distributed Dataset (RDD) which -is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes. RDD is read only and fault tolerant collection of elements that can be operated in parallel. Data sharing is very slow in MapReduce because of disk IO, serialization and replication. So, apache spark is used which supports in-memory processing computation. Data sharing in memory is much faster than between memory and disk and thus this is much faster as it stores the state of memory as an object.

So, considering all the features of Apache Spark that we can use, it still has some limitations on the type of data which it can run on. For example, there is a lot of Geographical data in the form of maps, locations, Socioeconomic data, Weather, location specific data, etc. Since due to the lack of support for spatial data operations, [3] proposed a new System GeoSpark which is in-memory clustering system for large scale spatial data.

2. HADOOP AND HDFS

Hadoop is a framework that can process and handle large amount of data using powerful computational models. The most important aspect of Hadoop is scalability it can scale up easily from a single cluster to multiple machines without affecting the performance. Each machine makes use of its own computational power and storage instead of using any supercomputer this makes the concept even more interesting and inexpensive. Fault tolerance in case of Hadoop is also done differently as it does not rely on hardware like other systems but handles it at application level. The file system used by Hadoop is HDFS which is Hadoop distributed file system

HDFS is a distributed file system that provides high performance access to data across Hadoop clusters. This file system is highly fault tolerant deployed on low cost hardware. It has many components across which all the data is distributed so every component holds a piece of information and that is why some part

is always non-functional, so quick recovery is a major goal. It supports a traditional hierarchical file architecture. Data replication is an important feature of the HDFS which stores large files in large clusters. The files are replicated for fault tolerance. The block size and replication factor are configurable per file. This file system is robust due to properties like data disk failure management, re-replication, cluster rebalancing, data integrity and snapshot.

HDFS has the following goals: Efficiency as data is distributed and processed on different machines in parallel, fault tolerant as it can quickly fix the issues by itself. The logic is data centered and so gives importance to data rather than any other aspect which is very important and is reliable since the data is replicated and if one node goes down we can access data which is replicated on the other node.

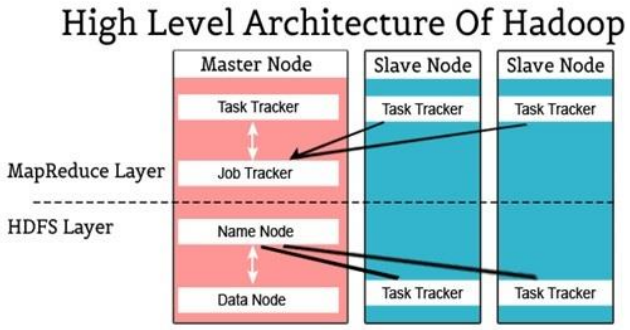

Figure 1: Hadoop Architecture

3. GEOSPARK

Apache Spark cannot support spatial data which is one of its drawbacks thus we make use of GeoSpark which uses Apache Spark concepts and processes large spatial datasets. GeoSpark consists of three layers: Apache Spark Layer, Spatial RDD Layer and Spatial Query Processing Layer. This makes use of the basic functionality of apache spark along with Resilient Distributed Dataset to support spatial data. The various forms of SRDD'S include Point RDD, Rectangle RDD and Polygon RDD and we ca choose to use any of these depending upon the type and form of the data points provided.

Another important feature of GeoSpark is that it can perform indexing on the data which is essential and in an adaptive manner decides whether a spatial index needs to be created on the partition thus strikes a balance between the performance and memory or CPU utilization. Moreover [3] also have provided various experimental results that GeoSpark achieves better run time performance when it is compared to SpatialHadoop which is Hadoop based.

3.1 Architecture

The architecture of the GeoSpark is as shown in the figure, divided into 3 layers mainly which are explained in detail in the following sub-sections. First is the basic apache spark layer which is the base underlying layer. On top of it is the Spatial RDD layer and on top of it is the spatial Query processing layer.

3.2 Spatial RDD layer

As seen earlier Spatial RDD layer makes sue of the basic concepts of the Apache Spark and perform functionalities like loading and saving data from and to the persistent storage. It also helps index the partitions which is decided adaptively.

According to the figure, there are three types of RDD`s, that are used in SRDD layer, which include Point RDD, Rectangle RDD, Polygon RDD. Depending upon the use and need we can use the RDD`s.

- **PointRDD**: We use this RDD when we must deal with Point data objects.
- **RectangleRDD:** We use this RDD when we must use Rectangle objects.
- **PolygonRDD**: This class of RDD is used when we must deal with Polygon set of points.

GeoSpark accepts data in all formats some of which are Comma Separated, Tab Separated data and well-known text. We do not require to convert the input format, but we just need to specify the format and the GeoSpark will transform and implement accordingly. It has built in memory and makes use of the pre-built functions calling the API's. So once Spatial RDD is initialized it makes available the pre-built functions to the user and then he can perform complex operations on the large spatial datasets [3].

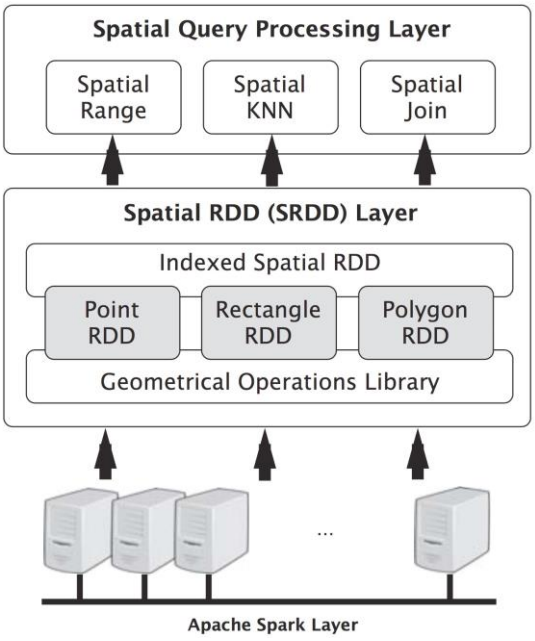

Figure 2: GeoSpark Architecture

For partitioning grid files are used which are creating by doing the space into equal sized grids. To get the complete grid back we can just merge all the cells back. In case the same data is written in two

grids for duplication purpose the ID specified for both these grids will still be different. This is how duplication is performed in this layer.

3.3 Spatial Query Processing layer

This is the top most layer out of the three layers discussed above as specified in the figure. The storage and processing of the geometrical object in the spatial RDD layer after that in the Spatial Query Processing layer the query is executed which is done in the in-memory cluster. The result of the query is then returned to the user in this layer itself. There are mainly 2 types which are explained in detail in the following section amongst others.

1. **Spatial Range Query**:

   In this case the user is expected to input the area. Let's say for exam we provide the area of the circle or in that case any polygon and then all the points, rectangle or polygon that lie in that area is computed and the result is returned to the user.

2. **Spatial Join Query**:

   This Query does the join operation on the Spatial Data. The input is taken from two SRDD and added to results and duplicates are removed. After the duplication is dealt with then the result is stored in the disk for later use.

4. PHASE 1

4.1 Problem Definition

The project for this course was distributed into three phases and all these phases are explained in detail n the following sections. In the first phase we had to install and setup Hadoop and Spark. In case of Hadoop as well as Spark we required one master and two slaves. To facilitate smooth communication among all three we needed to configure password-less SSH login. This would allow bidirectional communication among master and both the slaves. So due to these all three devices were inter-connected and could communicate without the need of password. After all the initial setup was done and the system was ready we needed to load the GeoSpark jar into Apache Spark Shell and execute the operations using Scala. We had to run the Apache Spark on the cluster. We were expected to pre-load data into Hadoop Distributed File System and use Scala to read the file from HDFS. Then once the file was loaded successfully and read using Scala we had to execute the query to create GeoSpark SpatialRDD and then query the PointRDD using Spatial Range Query, Spatial Join Query and Spatial KNN query.

4.2 Approach

In this phase the first task to perform was creating a password-less SSH login. So, to execute this task, we first generated the public and private keys on master and one of the slaves that is worker1 and worker2. Then copied the public key of slave which is worker2 on master. The next step was to shell into the master using the password of the slave using the command sshUSERNAME@IPADDRESS. Then make sure the public key is authorized by entering it into authorized keys. To do this we make use of the cat command on the command prompt. Logout of the current shell which is master2 and then test logging into the master without it asking for a password. If we can successfully log in without using a password from worker2 which is the slave into master, then we have successfully setup one direction password-less ssh. The same steps must be followed to setup bi-directional password-less SSH among the cluster using the following diagram.

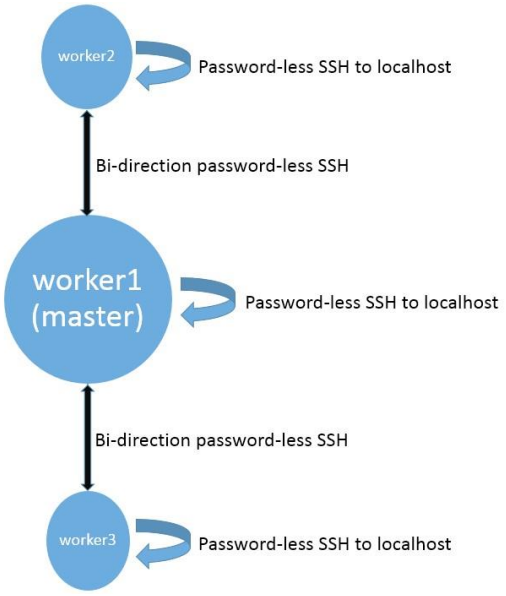

Figure 3: Password-less SSH

The next step to perform was the Hadoop configuration. B adding the correct java path in the appropriate folder to make sure it can access the java folder. Append the property tag correctly in all the 3 Hadoop folders we start Hadoop by running the command: ./start-all.sh. We checked the status of the Hadoop cluster by entering localhost:50070 as seen in the following diagram.

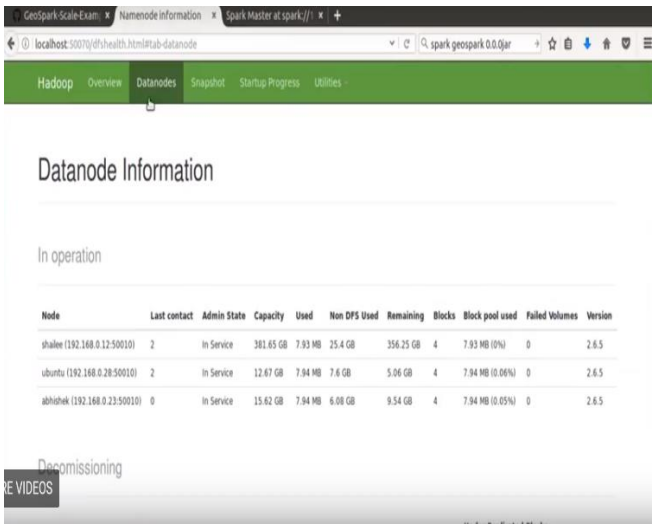

Figure 4: Hadoop Datanode Information

The next task was to do the spark configuration. We first add a line which states the IP address of the master in the spark folder. Run the command in the spark folder to start the master. Then to check the status of Spark enter the following in the web browser localhost:8080 as seen in the following figure.

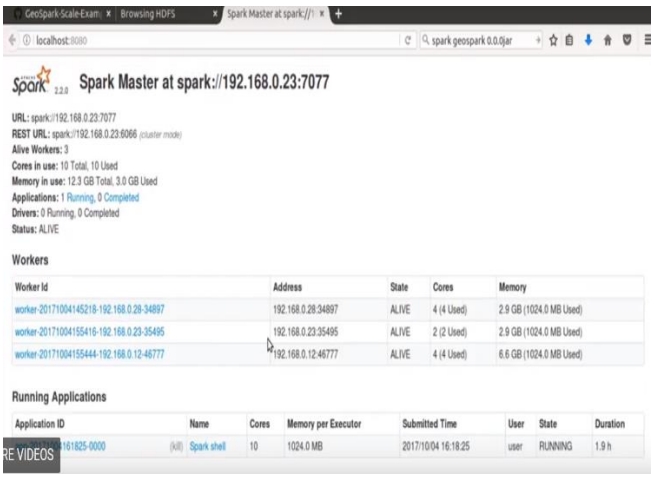

Figure 5: Spark Configuration.

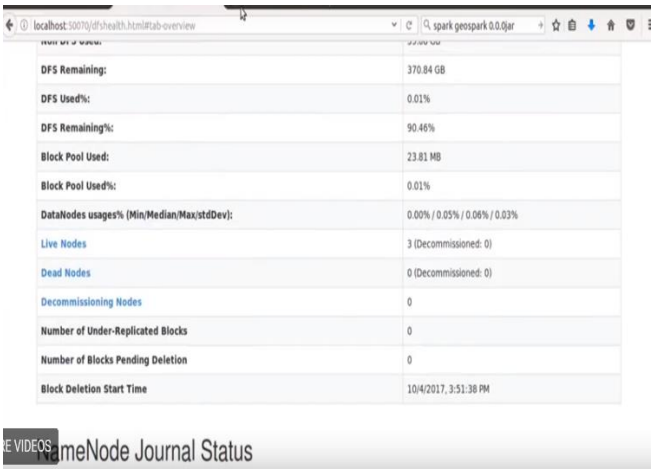

Figure 6: Live DataNodes in Hadoop cluster

After we finish setting up the Hadoop and spark following the instructions mentioned above and set up the bi-directional password-less SSH login among all the master and both the slaves we can check the status of the live nodes I the DataNode section of Hadoop which correctly displays 3 live nodes as seen in the above diagram.

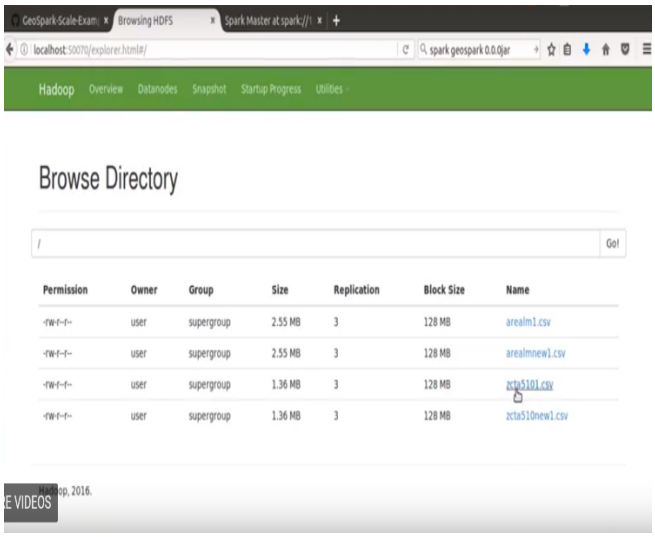

Figure 7: Browse directory in HDFS.

Once the above steps are completed we load the data into the Hadoop Distributed File System and make use of Scala to read the file and execute the query. As shown in figure 7 we go to the browse directory ta of Hadoop and check the file uploaded to HDFS. It displays the file which is uploaded which means the upload is done successfully.

Several example codes were provided in a GitHub repository which were referred during this phase of the project [4].

4.3 Difficulties

We tried using virtual machines to setup both the slaves on the same machine. There was a bridge connector issue which we successfully sorted by running a few commands and providing the necessary permissions. Also, the IP kept changing all the time which was one of the major issues since we had to keep editing the file every time the Ip changed so we found a solution for this problem and manually assigned the IP for all the machines. Also, due to the use of the same name on the two slaves the password-less SSH was not understanding which machine the key is authorized for and thus we had difficulty performing the password-less SSH, so we changed the names and successfully implemented the bi directional password-less ssh.

4.4 RESULTS

The query and results are shown in the following figures.

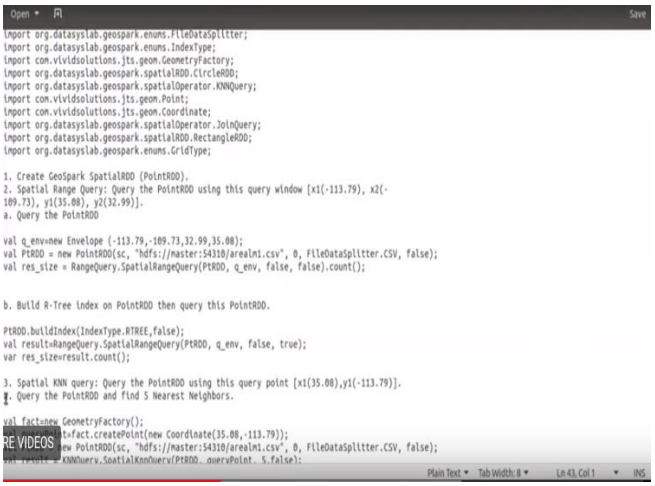

Figure 8: Spatial Range Query

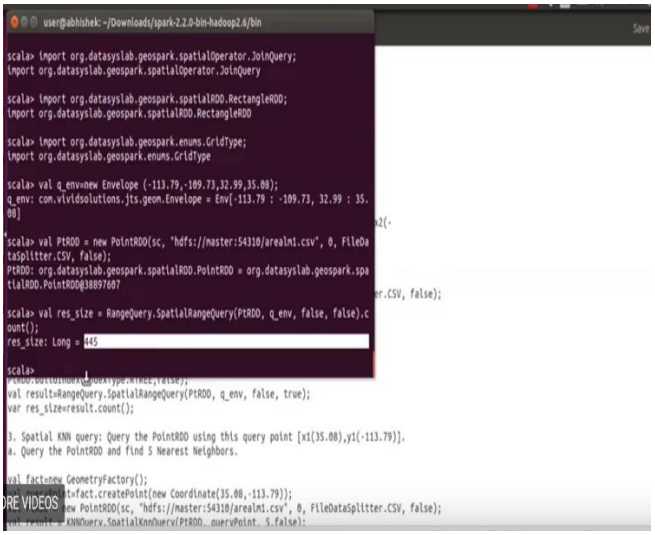

Figure 9: Result of Spatial Range Query

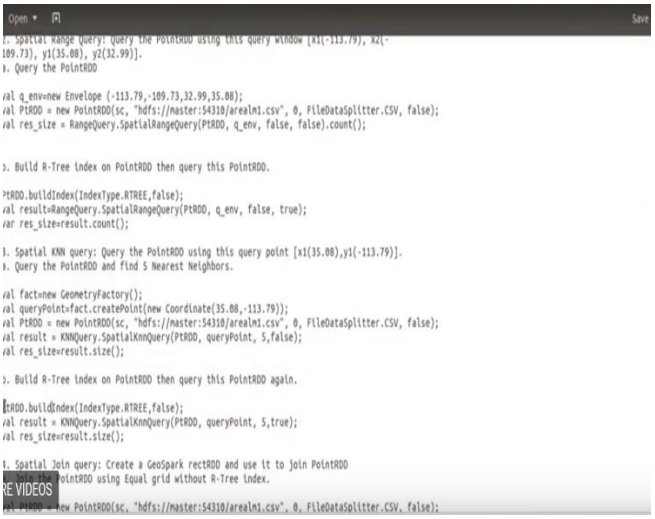

Figure 10: Spatial KNN Query

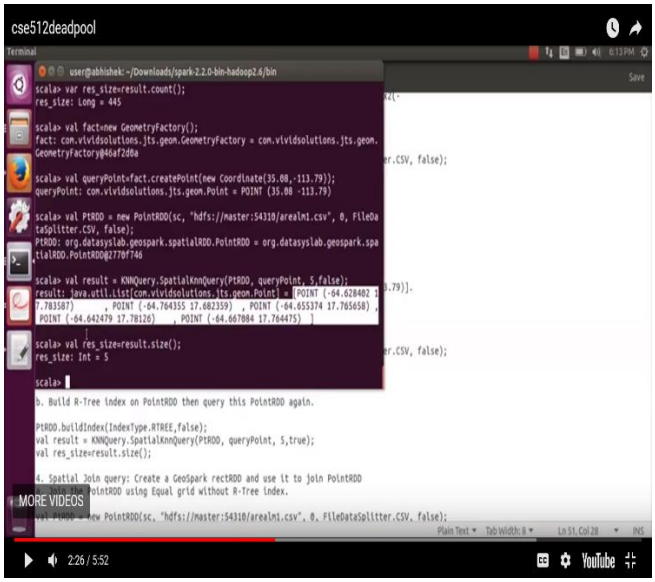

Figure 11: Result of Spatial KNN Query

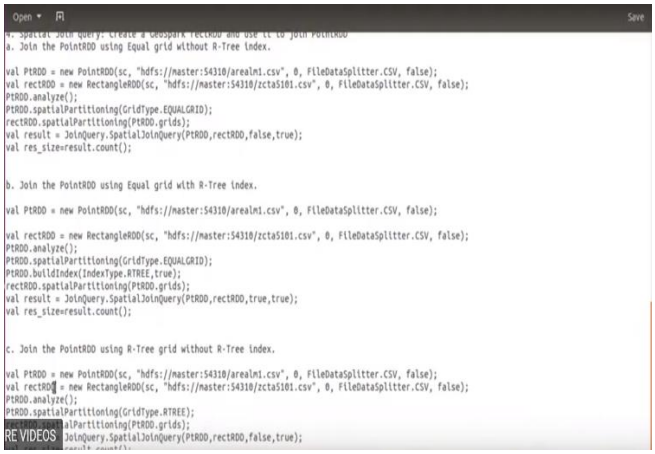

Figure 12: Spatial Join Query

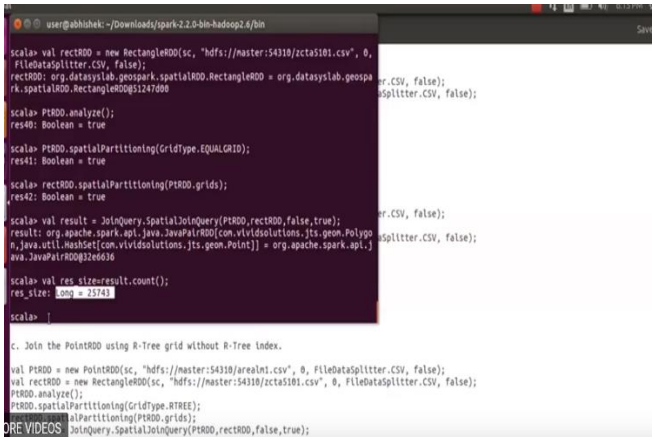

Figure 13: Result of Spatial Join Query

Upon execution of the above queries it was observed that that join on PointRDD using R-tree grid without R-tree Index was the best. Followed by PointRDD using Equal grid with R-tree index in the middle and without R-tree index at last.

5. PHASE 2

5.1 Problem Definition

In this phase of the project we were given tasks to write user defined functions in SparkSQL to perform four spatial queries which are: Range Query, Range Join Query, Distance Query, Distance Join Query. In SpatialQuery.scala we need to parse pointString and queryRectangle to the required format and check whether the queryRectangle fully contains the point. Also, the boundary conditions need to be taken care of. Inside ST_Within function we need to parse pointString1 and pointString2 convert to required format and make sure the two points are within the given distance using Euclidian distance.

5.2 Approach

A total of 3 different machines were used for setting up the cluster and executing the queries for this phase of the project. One machine was set up as the master and the other two were used as slaves named: worker1 and worker 2. This approach involved the password-less SSH login that needs to be set up among all 3 machines which makes it bi-directional.
We need to modify “SpatialQuery.scala” file to implement the user defined functions.
We setup the clusters on Thoth lab with needed configuration of Spark. The following are the details of the functions:
1. ST_Contains:
This function performs operations to find whether a point lies in the given rectangle or not.
Input: pointString:String, queryRectangle:String
Output: Boolean (true or false)
Range Query: We need to find all points containing in a rectangle
Range Join Query: We need to find all pairs (Point, Rectangle) such that the point is in the Rectangle.
Distance Query: We need to find all points that lie within a given distance from a point.
Distance Join Query: Given a distance in km, we need to find all pairs of points within the distance.
We have been given a template in which we need to write these user defined functions to complete the task. Then main function in this template takes dynamic length of parameters. The number of queries and the order of queries in the input do not matter. The code template will detect the corresponding query and call it and run on the corresponding dataset.

The following user defined functions were implemented.

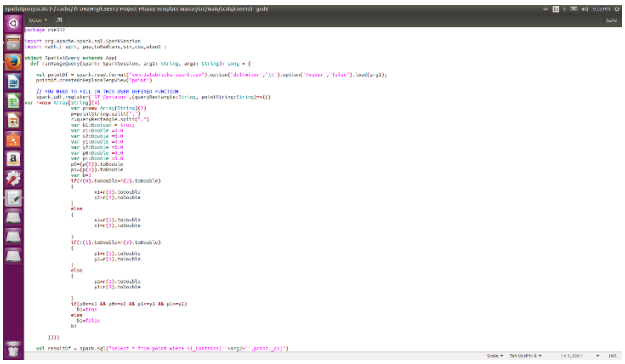

Figure 14: ST_Contains function

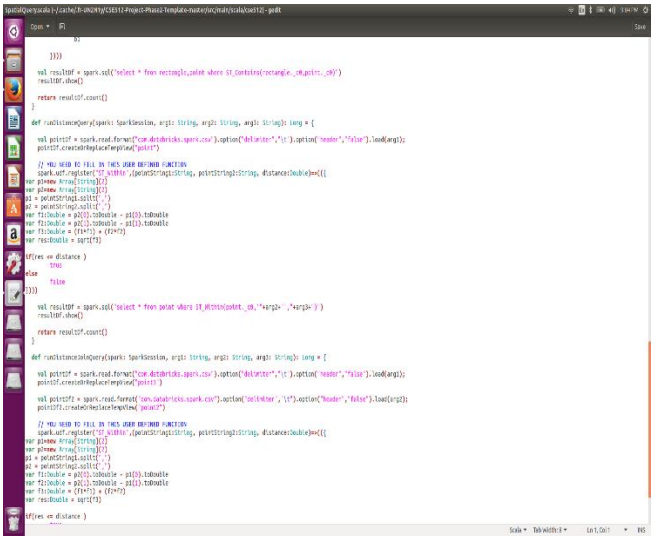

Figure 15: ST_Within function

5.3 Difficulties

We had a lot of problem running the code because of the parsing conditions. Also getting the boundary conditions right was very challenging. Also, we performed the first phase again on 3 separate machines instead of using a virtual box and managed the IP by manually setting it.

5.4 Result

We correctly got the result which was an exact match with the sample input and output. The result was submitted on vocaruem where there are 3 clusters for each login and thus the result was successful as seen on that website. Thus, the four spatial queries were successfully performed using user defined Scala functions.

input: result/output rangequery src/resources/arealm10000.csv -93.63173,33.0183, -93.359203,33.219456 rangejoinquery src/resources/arealm10000.csv src/resources/zcta10000.csv distancequery src/resources/arealm10000.csv -88.331492,32.324142 1 distancejoinquery src/resources/arealm10000.csv src/resources/arealm10000.csv 0.1

Output: 4 7612 302 123362

6. PHASE 3

6.1 Problem Definition

Given a collection of New York City Yellow Cab taxi trip records spanning January 2009 to June 2015, we need to perform two different hot spot analysis tasks. The first one is Hot Zone Analysis. This task needs to perform a range join operation on a rectangle dataset and a point dataset. For each rectangle, the number of points located within the rectangle will be obtained. The hotter rectangle means that it includes more points. So, this task is to calculate the hotness of all the rectangles. The second one is Hot Cell analysis in this task we need to focus on applying spatial statistics to spatio-temporal big data in order to identify statistically significant spatial hot spots using Apache Spark. So, we need to calculate the Getis-Ord statistic of NYC Taxi Trip datasets. We call it "**Hot cell analysis**".

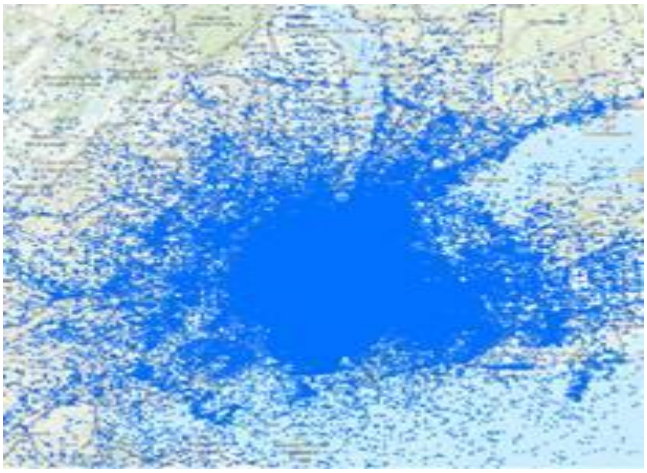

Figure 16: Hot cell overview.

Figure 17: Yellow Cab taxi trip hot spot in NYC

6.2 Approach

The input data format is as follows. The main function is "cse512.Entrance" Scala file. Point data is the input point dataset is the pickup point of New York Taxi trip datasets. The data format of this phase is the original format of NYC taxi trip which is different from Phase 2 which is already provided. Zone data for hot zone analysis is also provided. For hot zone analysis the input point data can be any small subset of NYC taxi dataset. For hot cell analysis the input point data is a monthly NYC taxi trip dataset (2009-2012) like "yellow_tripdata_2009-01_point.csv". However, there are some changes to be made from the requirement of GIS CUP in ot cell analysis as follows. The input will be a monthly taxi trip dataset from 2009-2012. For example, "yellow_tripdata_2009-01_point.csv", "yellow_tripdata_2010-02_point.csv". Each cell unit size is 0.01 * 0.01 in terms of latitude and longitude degrees. should use 1 day as the Time Step size. The first day of a month is step 1. Every month has 31 days. We only need to consider the pick-up location. The Geti-ord statistic is calculated using the following formula.

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

where x_j is the attribute value of the cell j, $w_{i,j}$ is the spatial weight between cell i and j, n is equal to the total number of cells and

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

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

The neighborhood for each cell in the space-time cube is established by the neighbors in a grid based on subdividing latitude and longitude uniformly. This spatial neighborhood is created for the preceding, current, and following time periods (i.e., each cell has 26 neighbors). For simplicity of computation, the weight of each neighbor cell is presumed to be equal.

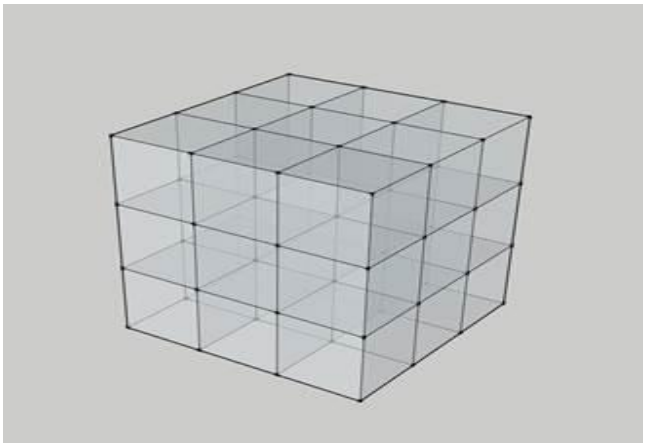

Figure 18: Visualization of a cell (in the center) and its Neighbor cells

A space-time cube is formed by latitude as the X-coordinate, longitude as Y-coordinate and Date in January as the Z-coordinate

b) Each cube is formed by 0.01 difference in interval on X, Y coordinates corresponding to degrees of latitude and longitude respectively and a time unit interval of 1 day on Z coordinate

c) We are considering the latitude and longitude locations belonging to the five boroughs of New York City with latitude values ranging between 40.5N – 40.9N, and longitude values ranging between 73.7W – 74.25W, date takes values from 1 to 31

d) The source data should take care of the outliers to remove errors in data

e) Only pickup location latitude and longitude values are considered for location input

6.3 Result

For hot zone analysis the output is a s follows:

All zones with their count, sorted by "rectangle" string in an ascending order.

"-73.795658,40.743334, -73.753772,40.779114",1

"-73.797297,40.738291, -73.775740,40.770411",1

"-73.832707,40.620010, -73.746541,40.665414",20

For hot cell analysis the coordinates of top 50 hottest cells sorted by their G score in a descending order. We do not have to output the z score.

-7399,4075,15

-7399,4075,29

-7399,4075,22

7. Performance Comparison

Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. It is based on a hierarchical design targeted at federations of clusters. It leverages widely used technologies such as XML for data representation, XDR for compact, portable data transport, and RRDtool for data storage and visualization. It uses carefully engineered data structures and algorithms to achieve very low per-node overheads and high concurrency. The implementation is robust, has been ported to an extensive set of operating systems and processor architectures. So we ran the queries of phase 2 on this tool to generate the following performance graphs for comparison and analysis.

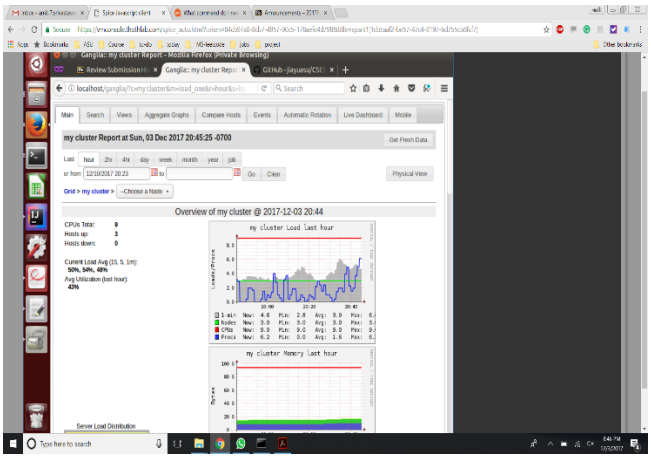

Figure 19: Cluster load and memory

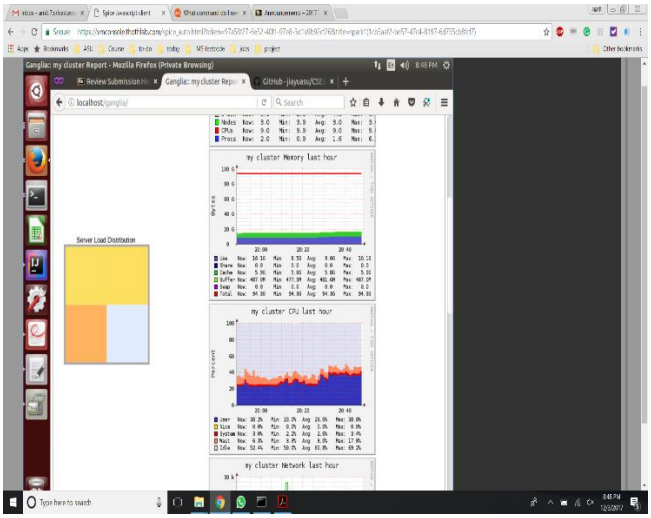

Figure 20: Cluster CPU and Server Load Distribution

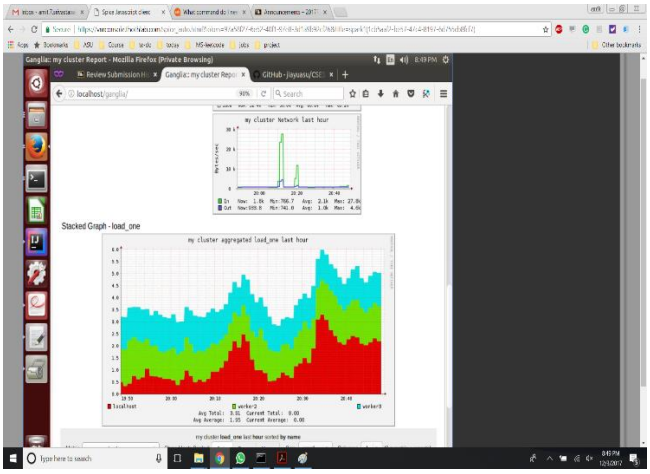

Figure 21: Cluster Network and aggregated load of master and two slaves

As the code was executed the following results are obtained and written as follows, the Ganglia screenshots demonstrate both Execution Time, load average and Average CPU usage for the queries.

8. CONCLUSION

We successfully implemented all three phases of the project and obtained the required result. In the start we learnt the basic concept of distributed systems and then had to research a lot about this topic to implement different frameworks on distributed systems.

In phase 1 we learnt how to setup a multimode cluster which had one master and two slaves. Thus, learnt about Hadoop and Apache Spark installed on these clusters. Also, further as we had to upload the file on Hadoop Distributed File System we learnt in depth about it and how Scala can be used along with its in-built memory to perform operations.

In phase 2 we learnt to implement user defined functions to run spatial queries on geo-spatial database and check if the data point lies within the given area of any polygon which can be useful in many geospatial data in real world for numerous applications.

In Phase 3, we implemented an algorithm to compute hot cell analysis and hot zone analysis in New York City for Taxi data. It taught us how to use MapReduce and how to make the best use of cluster computing for large datasets.

In conclusion, we experimented with Apache Spark, Scala Shell, and GeoSpark API learnt how to set up clusters in Hadoop and Spark, created fully bidirectional password-less accesses to each other computer, processed a large dataset on multiple nodes, and performed hot cell analysis and hot zone analysis.

9. REFERENCES

[1] http://geospark.datasyslab.org/

[2] Marcos R. Vieira (IBM Research, Brazil), Handbook of Research on Innovative Database Query Processing Techniques, 10.4018/978-1-4666-8767-7.ch009

[3] J. Yu, J. Wu, and M. 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.

[4] Brinkhoff, T.; Kriegel, H. P.; Seeger, B. (1993). "Efficient processing of spatial joins using R-trees". ACM SIGMOD Record. 22 (2): 237. doi:10.1145/170036.170075.

[5] https://github.com/DataSystemsLab/GeoSpark/wiki/GeoSpark-Important-Features

[6] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *ACM* sigmod record, volume 24, pages 71-79. ACM, 1995.

[7] https://gist.github.com/jiayuasu/84029bc9877804b876d209936e5b757c

[8] Ord, J.K. and A. Getis. 1995. "Local Spatial Autocorrelation Statistics: Distributional Issues and an Application" in Geographical Analysis 27(4).

[9] https://gist.github.com/jiayuasu/28cb8edb3287f705ed4d30c30de07639

[10] http://www.infoq.com/articles/apache-spark-introduction

[11] https://spark.apache.org/docs/1.2.0/programming-guide.html

[12] http://www.openheatmap.com

[13] https://en.wikipedia.org/wiki/R-tree.

[14] https://github.com/jiayuasu/CSE512-Project-Hotspot-Analysis-Template

[15] http://ganglia.info/