GeoSpatial Analysis using Apache Spark

Sagar Patni
School of Computing, Informatics and Decision Systems Engineering, Arizona State University
shpatni@asu.edu
1213217718

Pratik Suryawanshi
School of Computing, Informatics and Decision Systems Engineering, Arizona State University
psuryawa@asu.edu
1213231238

Manasa Pola
School of Computing, Informatics and Decision Systems Engineering, Arizona State University
mpola@asu.edu
1214416203

ABSTRACT

This project report provides detailed explanation of GeoSpatial data analysis using GeoSpark framework. It includes details about Hadoop, Apache Spark, hadoop and Spark environment setup and implementation of user defined and geospatial operations.

Keywords

Hadoop, Apache Spark, Scala, MapReduce, HDFS, GetisOrd, Hotspots, RDD, large-scale data, Spatial data, DAG.

1. INTRODUCTION

Distributed database means sharing a database across multiple nodes so that it can obtain a storage space extension and also can benefit from multiple processing resources. Hadoop Mapreduce is a used for large dataset computations. Apache Spark helped us to perform the computations in the cluster and perform complex computations easily. Using GeoSpark we calculated hotspots using the spatial temporal data.

2. SYSTEM ARCHITECTURE

Apache Hadoop

Apache Hadoop is a framework that provides a simple programming interface to perform large-scale cluster computing. It is designed such that it can scale vertically from a single server to thousands of machines each having it's on computing power and storage. Every computation of Apache Hadoop is written on disk which makes it fault tolerant. The library can detect and handle any system failures at application layer which makes it highly available.

Apache Hadoop provides distributed storage and processing of big data. The storage part is called as own as Hadoop Distributed File System (HDFS) and the processing part is called as MapReduce programming model. The Hadoop split the data into data blocks and splits them into different clusters.
It sends each cluster a part of the code to perform computations on data. So, each node has its own copy of data and operations to perform using such model Hadoop takes advantage of faster processing at local sites.
Apache Hadoop is composed of different modules [7],

- Hadoop Common: Hadoop Common has libraries for Hadoop.
- Hadoop Distributed file system: HDFS provides distributed storage of Apache Hadoop framework.
- Hadoop Yarn: Hadoop YARN (Yet Another Resource Negotiator) was introduced in Hadoop 2, It consists of two daemons called resource manager and application master. The resource manager allocates resources to various applications and application manager monitors the execution of the processes.
- Hadoop MapReduce: MapReduce is programming model for large scale data processing.

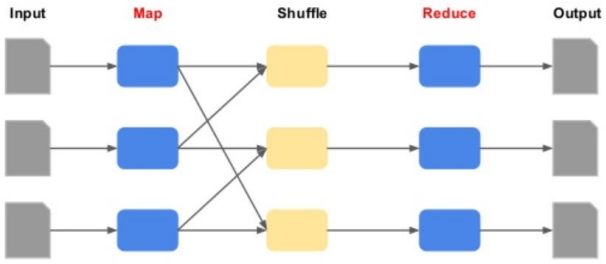


Figure 1: Hadoop MapReduce working.

A Hadoop cluster has a single master node and multiple worker nodes. The master node has Jobtracker, NameNode, DataNode. A worker node can act as both DataNode and TaskTracker. NameNode is central to HDFS, its single point of failure. NameNode stores metadata about HDFS, It does not have actual data. Using metadata NameNode can construct the original file from different blocks divided into clusters. The data node is responsible for storing actual data on HDFS.

Apache Spark

Apache spark is a cluster computing framework which is based on Apache MapReduce. MapReduce is widely adopted for processing and generating large datasets

with a distributed cluster processing. In MapReduce to reuse data between computations is to write on disk HDFS which a costly operation which makes the data sharing slow. Batch processing and Stream processing applications nowadays requires much faster processing of the data. Apache Spark is a good fit for both batch processing and stream processing, meaning it's a hybrid processing framework. Spark speeds up batch processing via in-memory computation and processing optimization. Its resilient distributed dataset (RDD) allows Spark to transparently store data in-memory and send to disk only what's important or needed. As a result, a lot of time that's spent on the disk read and write is saved.

Spark Architecture

- Apache Spark Core: It is core engine of spark that provides in memory computations.
- Spark SQL: Spark SQL is a distributed structured data processing engine. It provides data abstraction called DataFrames.
- Spark Streaming: It is distributed real time stream processing engine. It uses Spark Core fast scheduling capability to perform streaming analytics.
- Spark MLlib: Spark MLlib is a distributed machine learning framework.
- Spark Graphx: It is a distributed graph processing engine. It provides an API for expressing graph computation that can model the user-defined graphs by using Pregel abstraction API.

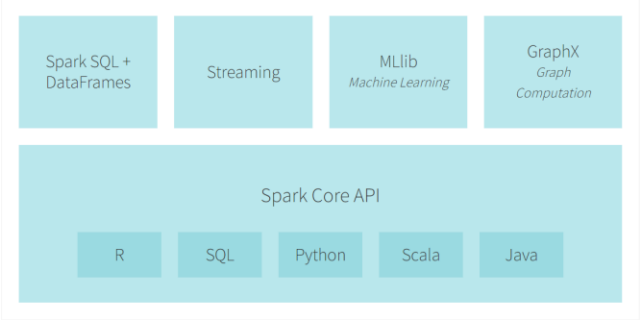


Figure 2. source: https://databricks.com/spark/about

Spark performs in memory computations using in-memory data structure called RDD (Resilient Distributed Datasets). RDD is a read only collection of records that can be divided into logical partitions which can be computed on different nodes. RDD are used to achieve faster and efficient MapReduce operations.

GeoSpark

GeoSpark enables large scale cluster computing with Apache Spark. GeoSpark provides Spatial Resilient Distributed Datasets (SRDDs) to perform spatial operation similar to Spark SQL.

GeoSpark supports the following [15]:

- Geometrical and Spatial Queries like Range, K Nearest Neighbors, Join queries
- Geometrical objects such as Point, Polygon, Rectangle and Line String
- Geometrical operations such as Minimum Bounding Rectangle, Polygon Union, and Overlap/Inside(Self-Join)
- Spatial indexing: R-Tree and Quad-Tree
- Spatial query operations such as Spatial range query, spatial join query and spatial K-Nearest-Neighbours query.

Apache Spark over Hadoop

Apache Spark was designed to enhance the Hadoop Stack. Apache Spark is designed to read and write data to Hadoop HDFS to There are three ways to deploy Spark in a Hadoop cluster: standalone, YARN, and SIMR. Below diagram depicts how spark can be used in these three modes,

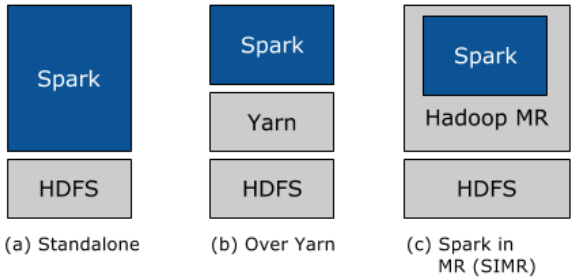


Figure 3. Source: https://databricks.com/blog/2014/01/21/spark-and-hadoop.html

Spark UDF

Spark lets us define user defined function to work with Spark SQL. UDF are used when built in SQL functions are not sufficient but they must be used sparingly because they are not performant. In phase II and III of this project, we will use UDF to perform spatial query operations.

Ganglia

Ganglia is a distributed monitoring tool which can measure metrics of the cluster such as CPU utilization, network use.

System Architecture

Below is the system architecture of our cluster,

- Number of nodes: 3
- Number of cores per node: 2
- Operating System: Linux Ubuntu 16.04 LTS 64-bit.
- Java OpenJDK 1.8.0
- Hadoop 2.7.5
- Spark 2.2.1
- Ganglia.

3. ENVIRONMENT SETUP

In this section configurations steps to set up Hadoop and Spark cluster are given, (referred from [14])

3.1 Password Less SSH login

The password less ssh login is required for cluster to automate the the login process in order to perform the operations without user intervention. To make the environment secure we create a ssh key on local machines and copy it to the master server.

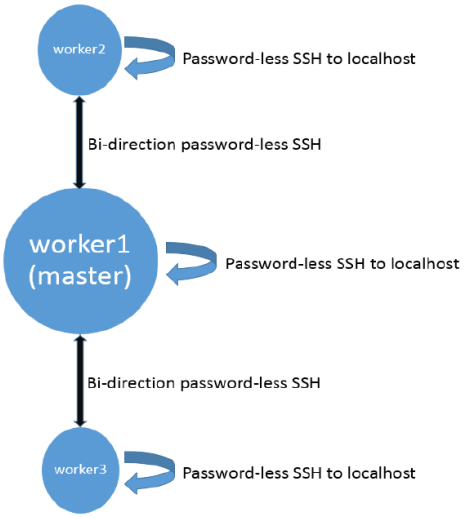


Figure 1 Password-less SSH Topology

Figure 4: password less ssh. [9]

The steps we followed are as follows [9]:

- Generate the public key and private key (On worker1 and worker2) using ssh-keygen.

```
ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa_worker1
ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa_worker2
```

- Copy the public key of worker2 to master (On worker2)

```
scp~/.ssh/id_rsa_worker2.pub
USERNAME@192.168.0.41:~/.ssh/id_rsa_worker2.pub
```

- Shell into master (worker1) with passwords (On worker2)

```
ssh USERNAME@192.168.0.41
```

- Authorize the public key by adding it to the list of authorized keys (On worker2)

```
cat ~/.ssh/id_rsa_worker2.pub >> ~/.ssh/authorized_keys
```

- Log out of the current shell (On worker2) exit
- Test that we can log in master with no password (On worker2)

```
ssh 192.168.0.41
```

- After the steps above, we have a one-direction passwordless SSH login from worker2 to master (worker1) We can create similar for others.

3.2 Configuring Hadoop

We downloaded Hadoop package from http://www.apache.org/dyn/closer.cgi/hadoop/common/ After extracting the zip file to Hadoop Folder, we performed the following configuration changes [6]:

- Append the following lines in /etc/hosts. (Master only)

```
192.168.0.41 master
192.168.0.42 worker2
192.168.0.45 worker3
```

- Append the following line in /etc/hosts. (Workers only)

```
192.168.0.41 master
```

- Add the following line in Hadoop Folder/etc/hadoop/hadoop-env.sh to point out Java folder. (All machines)

```
export JAVA_HOME=/usr/lib/jvm/YOURJAVAFOLDER
```

- Clear content and add the following lines in Hadoop Folder/etc/Hadoop/slaves. This step gives the list of all the machines who can be workers to Hadoop master. (Master only)

```
master
worker2
worker3
```

- Append the following lines in Hadoop Folder/etc/Hadoop/core-site.xml. (All machines)

```
<property>
        <name>fs.default.name</name>
        <value>hdfs://master:54310</value>
</property>
```

- Append the following lines in Hadoop Folder/etc/Hadoop/mapred-site.xml. (All machines)

```
<property>
        <name>mapred.job.tracker</name>
        <value>master:54311</value>
</property>
```

- Append the following lines in Hadoop Folder/etc/Hadoop/hdfs-site.xml. The value should be consistent with the number of machines in the step 3(4). (All machines)

```
<property>
        <name>dfs.replication</name>
        <value>3</value>
</property>
```

- Delete "hadoop-USERNAME" folder in /tmp folder in all machines (if it exists) before you format HDFS in Step 3(9). (All machines)
- Run the following line in Hadoop Folder/bin/ folder to format the HDFS system if this is the first time to run this Hadoop or after you change any machine attributes such as IP address. (Master only). ./hadoop namenode -format
- Run the following line in Hadoop Folder/sbin/ folder to start your cluster. (Master only). ./start-all.sh
- Check the status of Hadoop cluster by entering "localhost:50070" in linux browser (master only).
- Now, Apache Hadoop is setup on our cluster.

3.3 Configuring Spark

After downloading Apache Spark package from https://spark.apache.org/downloads.html and extracting to SparkFolder, the following configuration changes are required [12]:

- Add the following line in SparkFolder/conf/spark-env.sh. In our case, master (worker 1) has IP address 192.168.0.1 (Master only)

```
SPARK_MASTER_IP=192.168.0.41
```

- Run the following line in Spark Folder/sbin/ folder to start Spark master. (Master only)

```
./start-master.sh
```

- Check the status of Spark master by entering localhost:8080 in your linux browser.
- Run the following line in Spark Folder/bin/ folder to register and run the worker in Spark master. 192.168.0.41:7077 is on the master.(Workers only)

```
./spark-class org.apache.spark.deploy.worker.Worker spark://192.168.0.41:7077
```

- If you see prompts like the following, that means this worker has been registered to the master. (Workers only).
- Do not close the terminal in workers after step.
- Check the status of Spark master again.

After setting up the environment we worked on the project in three phases enlisted below sections and to get more understanding about the working of spark in clustered environment we monitored the server metrics using Ganglia.

3.4 Ganglia Setup

To measure the performance we installed distributed monitoring tool Ganglia using below steps[14],

- sudo apt-get install -y ganglia-monitor rrdtool gmetad ganglia-webfrontend (master only)
- sudo apt-get install -y ganglia-monitor (client only)
- modify sudo vi /etc/ganglia/gmond.conf to add cluster information (all machines)
- restart the ganglia monitoring service on each node
  sudo service ganglia-monitor restart
- Restart the ganglia monitor gmetad on master node.
  sudo service ganglia-monitor restart && sudo service gmetad restart && sudo service apache2 restart

4. EXPERIMENTAL SETUP

4.1 Phase 1

After setting up password-less SSH, configuring Hadoop and Spark, we first load datasets to HDFS. The zcta510.csv file contained data representing rectangles and arealm.csv had data for points. We used GeoSpark RectangleRDD to process rectangle dataset and GeoSpark PointRDD to process point dataset. Phase-1 was run on 3-node cluster setup on local cluster.

GeoSpark jar was loaded into Apache Spark Scala shell and following operations were performed: [6]

1. Create GeoSpark SpatialRDD (PointRDD).
2. Perform Spatial Range Query: Query the PointRDD using query window
   [x1(-113.79), x2(- 109.73), y1(35.08), y2(32.99)].
   a. Query the PointRDD.
   b. Build R-Tree index on PointRDD and then query this PointRDD.

3. Spatial KNN query: Query the PointRDD using this query point [x1(35.08), y1(-113.79)].
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: Create GeoSpark RectangleRDD and use it to join PointRDD.
a. Join the PointRDD using Equal grid without R-Tree index.
b. Join the PointRDD using Equal grid and R-Tree index.

c. Join the PointRDD using R-Tree grid without R-Tree index.

The result obtained for spatial range query without index for given point is 445 which is number of points closed to the given point. Same operation using R-tree indexing provides result much faster.

Code Snippet of Spatial Range Query

```
/* Spatial Range Query without Index */

val envelope=new Envelope (-113.79,-109.73, 35.08, 32.99);
val rdd = new PointRDD(sc, "hdfs://192.168.0.41:54310/arealm.csv", 0, FileDataSplitter.CSV, false, StorageLevel.MEMORY_ONLY);
val resultSize = RangeQuery.SpatialRangeQuery(rdd, envelope, false, false).count();
println(resultSize);


/* Spatial Range Query with Index */

val envelope=new Envelope (-113.79,-109.73, 35.08, 32.99);
val rdd = new PointRDD(sc, "hdfs://192.168.0.41:54310/arealm.csv", 0, FileDataSplitter.CSV, false, StorageLevel.MEMORY_ONLY);
rdd.buildIndex(IndexType.RTREE,false);
val resultSize = RangeQuery.SpatialRangeQuery(rdd, envelope, false, true).count();
println(resultSize)
```

The Spatial KNN query with k=5 returned 5 nearest neighbors to the given point. Using spatial KNN query along with index improves its performance.

Code Snippet of KNN Query:

```
/* Spatial KNN Query without Index */

val fact=new GeometryFactory();
val point=fact.createPoint(new Coordinate(-113.79, 35.08));
val rdd = new PointRDD(sc, "hdfs://192.168.0.41:54310/arealm.csv", 0, FileDataSplitter.CSV, false, StorageLevel.MEMORY_ONLY);
val resultSize = KNNQuery.SpatialKnnQuery(rdd, point, 5,false).size();
println(resultSize);

/* Spatial KNN Query with Index */

val fact=new GeometryFactory();
val point=fact.createPoint(new Coordinate(-113.79, 35.08));
val rdd = new PointRDD(sc, "hdfs://192.168.0.41:54310/arealm.csv", 0, FileDataSplitter.CSV, false, StorageLevel.MEMORY_ONLY);
rdd.buildIndex(IndexType.RTREE,false);
val resultSize = KNNQuery.SpatialKnnQuery(rdd, point, 5,true).size();
println(resultSize);
```

For next query i.e., Spatial join using Equal grid without index, it can be done by loading point and rectangle data, doing spatial partitioning on them and then performing the join operation. The result of the operation is 25743. Amount of time required to get result for this query is lot more than other queries.

R-Tree Index:
R-Tree is aimed at handling geometrical data, such as point, line segments, surfaces, volumes, and hyper volumes in high dimensional spaces. R-tree is a height balanced tree like B-tree. R-tree is used in spatial database as a spatial access method. Rtree applications cover a wide spectrum, from spatial and temporal to image and video databases. R-tree has root node, intermediate node and leaf node. Every leaf record is a smallest bounding box. Root has at least two children. leaf node does not store the actual spatial objects, but it stores the minimum bounding rectangle of the actual spatial objects, but it stores the minimum bounding rectangle of the actual spatial objects. the tree is constructed hierarchically by grouping the elaf boxes into larger, higher level boxes which may themselves be grouped into even larger boxes at the next higher level.

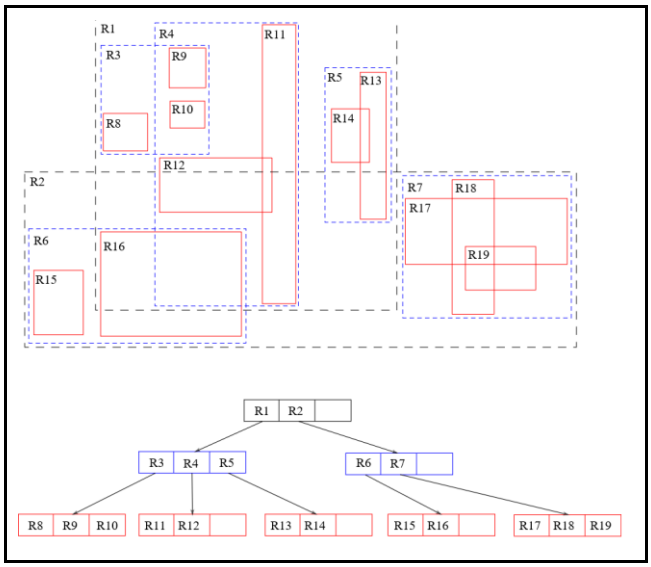


Figure 5. R-Tree [13]

Equal Grid
Equal grid partitions the data in equal sized spatial areas, this would cause performance issues when the data is not equally distributed.

R-Tree Grid
R-Tree would partition the data more logically leading to better query performance. R-tree groups the data points according to their distances and data points according to their distances and which data point belong to which group. This results in more memory utilization.

But Using an R-Tree index for the same join operation took much less time and returned the same result. Same join operation using R-tree grid without index also too longer to get result.

4.2 Phase 2

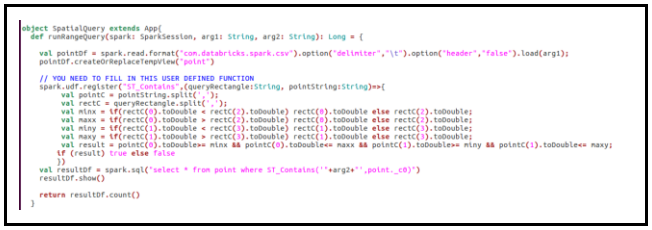
In Spark, DataFrame is distributed collection of data organized into columns which can be related to tables in relational database/ Developers can treat these DataFrames as tables in RDBMS and can perform SQL operations to get required results.

In Project Phase 2 [2], we used Spark SQL UDF i.e. User Defined Functions which are custom functions that can be used in SQL queries. We used ST_Contains and ST_Within predicates in Spark
SQL to determine the results of range, range-join, distance and
distance-join queries.

Rectangle and point DataFrame were created using csv files containing coordinate data and above-mentioned queries were performed.

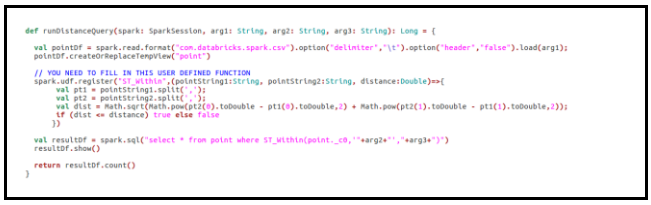
Spatial queries performed on the DataFrames using Spark SQL UDF:

(1) Range query: Given a query rectangle R which is coordinates string of diagonally opposite points of rectangle and a set of points P. To find points within query rectangle R, we loaded data from point DataFrame using select query in SQL and perform custom predicate evaluation written in ST_Contains to determine if current selected point lies in rectangle. If evaluation results into True, then that point will be considered. This set of points that lie in rectangle is obtained.



(2) Range join query: Given a set of Rectangles R and a set of Points S. We are required to find all Point-Rectangle combinations such that point lies in that corresponding rectangle.
Data is loaded from Rectangle and Point DataFrame using select query in SQL, which performs cartesian product of rows in Rectangle and rows in Point. Result is evaluated through UDF ST_Contains which determines if the point lies in Rectangle- Point pair obtained. If evaluation results to True, then that pair is considered in answer set.

(3) Distance query: Given a point location P and distance D in km. We are required to find all points that lie within distance D from P. The select query in SQL will load points from Point DataFrame and evaluate UDF ST_Within for current point coordinates. If it evaluated to be True, then that point lies within D range of input point P and will be considered in answer set.



(4) Distance join query: Given a set of Points S1 and a set of Points S2 and a distance D in km. We were required to find pairs (s1, s2) such that s1 is within a distance D from s2. The select query in Spark SQL loads data from Point DataFrame. To form pairs, we perform Cartesian product on set S1 and S2 and later perform custom predicate evaluation in ST_Within. If this evaluation results to True, then pair of points from cartesian product lies within distance D from each other.

To test out phase 2 application, we created local spark cluster of 3 nodes. Finally, solution jar file was submitted in Vocareum [13], a powerful cluster for final evaluation.

4.3 Phase 3

Problem Statement
In phase 3 [3], we applied spatial statistics techniques to find the hotspots in Spatio-temporal data.Data is related to cab pickups in New York City (NYC). Here hotspots are locations with highest pickup count.

For complete problem definition please refer to [4]: http://sigspatial2016.sigspatial.org/giscup2016/problem

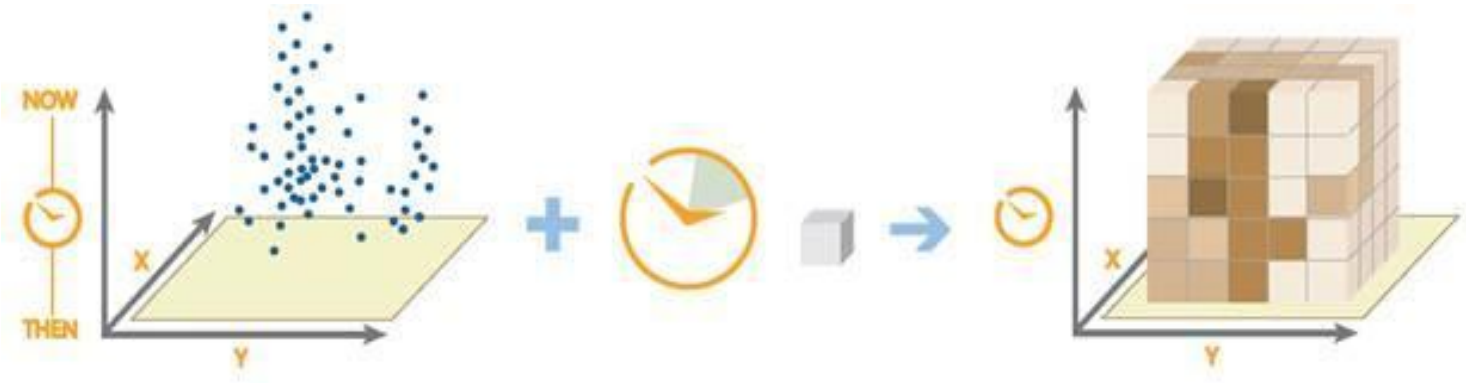
Our aim is to calculate Getis-Ord statistic for the NYC Taxi Trip dataset. Getis-Ord can be calculated by using below formula[5] :

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

where mean is

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

and standard deviation is



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

where xj is the attribute value for cell j, wij is the spatial weight between cell i and j, n is equal to the total number of cells

Problem Analysis
In Phase 3,we were required to develop solutions that performed spatial hot spot analysis, namely Hot zone analysis and Hot cell analysis

Hot Zone Analysis:

This task will needs to perform a range join operation on a rectangle and a point dataset provided in csv.For every rectangle we have to find points within it . Finally we have to get all hot rectangles which is rectangles with highest point covered.

The solution of this problem involves using cartesian product on rectangle and point dataset to get all possible pairs and then using spark sql udf ST_Within to determine if that point lies in rectangle.

The resulting data in dataframe is later grouped based on rectangle coordinates and count of each rectangle is obtained. Finally all data is sorted in ascending order. The count represents the hotness of each rectangle.

Hot Cell Analysis:

For this task we need to find the coordinates of top 50 hottest cells sorted by their Z score in a descending order. Getis-Ord statistic is defined as list of top fifty hot spot cells in time and space. This space-time cube can be visually represented as in Figure:

Following steps were involved to do Hot cell Analysis:

1. Filter the data from given dataset which satisfies range of latitude, longitude and day values.

```
var filteredData = pickupInfo.filter(col("x") >= minX && col("x") <= maxX).toDF()
filteredData = filteredData.filter(col("y") >= minY && col("y") <= maxY).toDF()
filteredData = filteredData.filter(col("z") >= minZ && col("z") <= maxZ).toDF()
```

2. Calculate pickup count for each of the cells to determine mean and variance for given data which later will be used to calculate Z score.

```
SELECT origin,
   Sum(pickupcount) AS total
FROM   neighbors
GROUP  BY origin
```

3. Neighborhood for each cell is calculated since Z score also depends on it.

```
SELECT origin,
       Calculateneighbors(origin)
       AS neighborcount,
       First(total)
FROM   pickupcount
GROUP  BY origin
```

4. Calculate Z score for each cell

```
SELECT neighborcount.origin,
Calculatezscore(
    First(neighborcount.neighborcount),
    First( neighborcount.total),
    First(constants.cellnum),
    First(constants.mean),
    First(constants.sd))
AS zscore
FROM   neighborcount
       CROSS JOIN constants
GROUP  BY neighborcount.origin
ORDER  BY neighborcount.origin
```

5. Sort the dataframe containing cell coordinates based on Z score in descending order to pick top 50 cells.

```
SELECT First(origin)
FROM   zscores
GROUP  BY zscore
ORDER  BY zscore DESC
LIMIT  50
```

Operation like calculating neighbors, calculating Z score are done using Spark SQL UDF.

5. EXPERIMENTAL EVALUATION USING GANGLIA

In this section we will evaluate the performance for phase 2 and phase 3 task by changing the system parameters.

As discussed in previous sections we configured ganglia monitoring system to get performance metrics of the cluster.

We mainly focused on metrics, CPU time, Network time and amount of memory used by system.

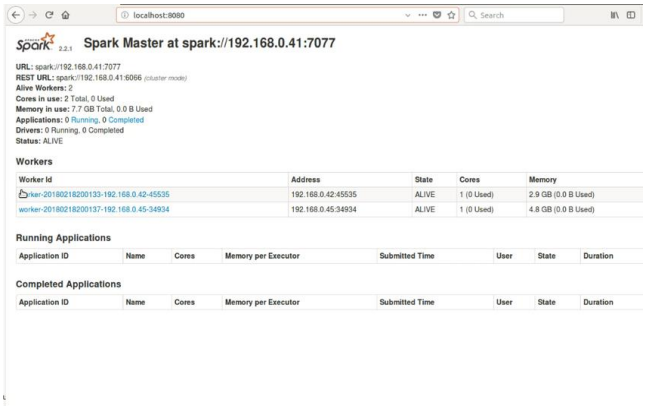


Figure 6. Spark cluster

Figure 6 shows our Spark/Hadoop cluster configuration.

Phase II

We changed number of Nodes and measured performance for each spatial range query. Below are the results and the snapshots from the tool Ganglia. Below are the results obtained using Ganglia monitoring system for using different number of cores. We used the datasets provided arealm10000.csv and ztca100000.csv to calculate results.

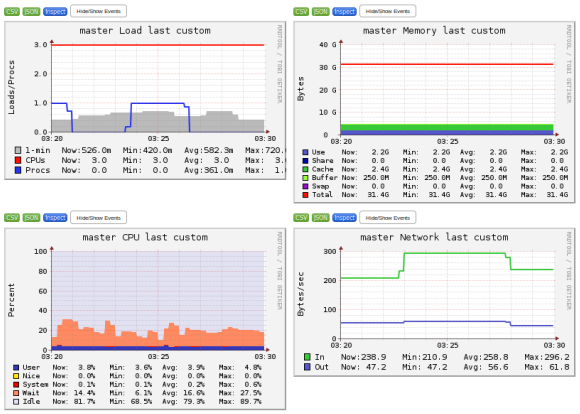


Figure 7. Activity on master, 9 cores

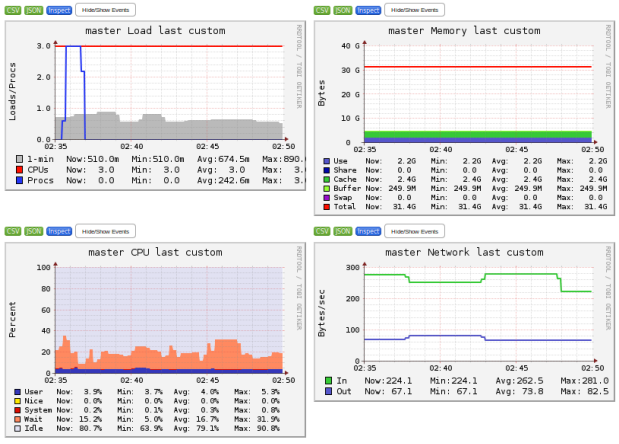


Figure 8. Activity on master node, 3 core phase II

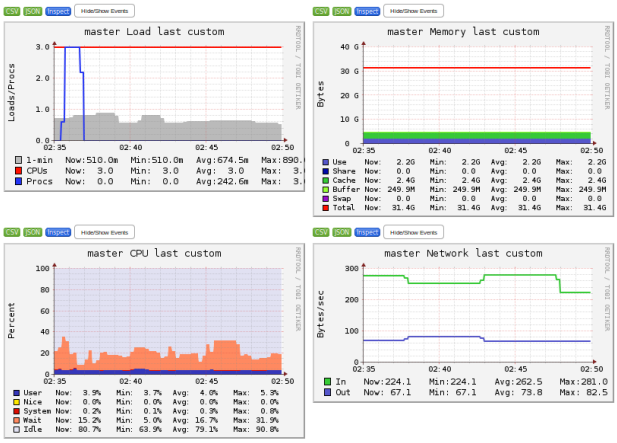


Figure 9. Activity on master node, 6 core phase II

Phase III:

We changed number of Nodes and measured performance for each spatial range query. Below are the results and the snapshots from the tool Ganglia. Below are the results obtained using Ganglia monitoring system. We used New York taxi dataset provided to us which is 2GB in size for each node.

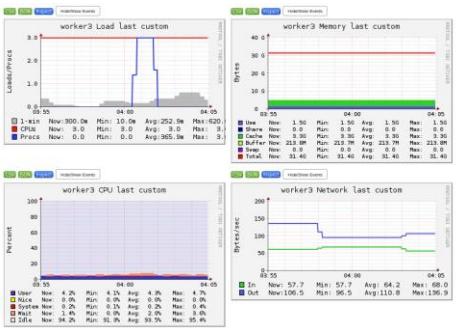


Figure 10. Activity on master 9 cores

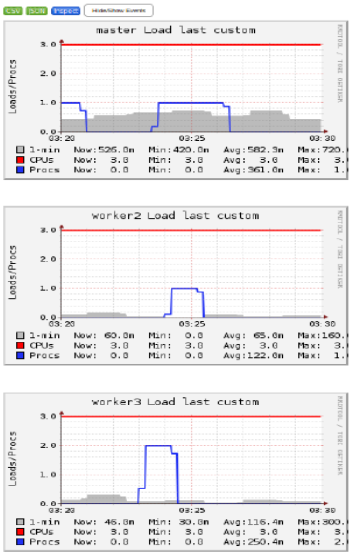


Figure 11. Load on each node, 9 cores

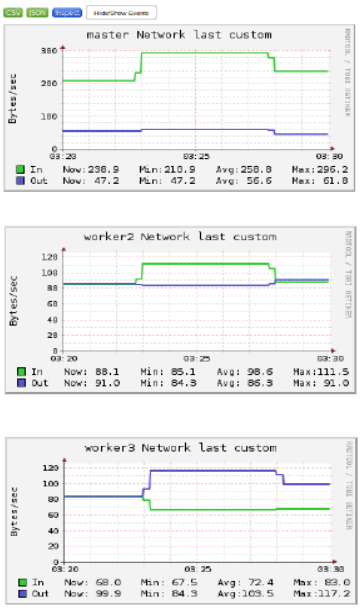


Figure Network Activity on nodes, 9 core setup.

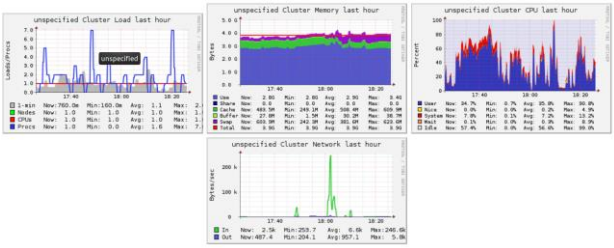


Figure 12. Activity on master 3 cores

6. RESULT INTERPRETATION

We can observe from the images for phase II and phase III, on increasing number of cores from 3 to 9. The load on cpu decreases. This can be attributed to in memory computation with spark with less memory overhead on cpu increases to move block in and out memory. We can see in the image for phase III with 3 cores the CPU utilization is very high and memory usages is also full. Spark store the the data objects in memory and which can be reused.

7. CONCLUSION

In this project we created cluster of nodes using apache spark. Using this cluster we performed operations like range, range join, distance and distance join queries. Unlike Apache Hadoop, in Apache Spark there is more flexibility for performing operations like filter, collect, map, reduce etc. Using Spark SQL we were able to get results more efficiently. DAG in Apache Spark has multiple levels that form a tree structure. Hence, execution is fast because intermediate results are not written into the disk. We were able to understand this performance efficiency through

results interpretation.Through this project we were able to understand concepts in distributed databases systems. Implementing this application to perform analysis on geospatial data gave us insight about how to use spark to solve big data problem.

8. ACKNOWLEDGEMENT

We would like to thank Professor Dr. Mohamed Sarwat for his valuable guidance throughout the course. We would like to thank out Teaching Assistant Yuhan Sun for his continuous support for the project.

9. REFERENCES

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

[2] Project Phase 2 Template - https://github.com/jiayuasu/CSE512-Project-Phase2-Template

[3] Project Phase 3 Template - https://github.com/jiayuasu/CSE512-Project-Hotspot-Analysis-Template

[4] ACM SIGSPATIAL Cup 2016 - http://sigspatial2016.sigspatial.org/giscup2016/problem

[5] ACM SIGSPATIAL Cup 2016 Submit and Evaluation- http://sigspatial2016.sigspatial.org/giscup2016/submit

[6] Spark - https://spark.apache.org/downloads.html

[7] Hadoop - http://www.apache.org/dyn/closer.cgi/hadoop/common/

[8] Geospark Template Project - https://github.com/jiayuasu/GeoSparkTemplateProject

[9] Password less SSH Login - https://www.tecmint.com/ssh-passwordless-login-using-ssh-keygen-in-5-easy-steps/

[10] Hadoop Info - https://hadoop.apache.org/docs/current/api/org/apache/hadoop/fs/FileSystem.html

[11] Installing Scala and Spark on Ubuntu - https://medium.com/@josemarcialportilla/installing-scala-and-spark-on-ubuntu-5665ee4b62b1

[12] Setup Apache Spark Standalone cluster on Multiple Machine - http://paxcel.net/blog/how-to-setup-apache-spark-standalone-cluster-on-multiple-machine

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

[14] https://www.linode.com/docs/databases/hadoop/how-to-install-and-set-up-hadoop-cluster/

[15] https://github.com/DataSystemsLab/GeoSpark