Geospatial Data Analysis Using Apache Spark
CSE 512: DISTRIBUTED DATABASE SYSTEMS

Nishant Washisth
Arizona State University
nwashist@asu.edu
1217130460

Aakash Rastogi
Arizona State University
arastog9@asu.edu
1215964854

Vivek Agarwal
Arizona State University
vagarw14@asu.edu
1213164067

ABSTRACT

There has been a tremendous increase in the data volume over the past few years and it has been increasing with the rate, it has never been before. It has become very difficult to handle such a vast amount of data which leads to the need for distributed systems. Geo-spatial data is the data which has some geographical information linked to it. It means that the records in the dataset consists of some data which can be in the form of coordinates, positions, addresses, etc. Geospatial data can originate from various sources like satellites, GPS and mobile applications. In this project, we used the GEOSPARK framework which is an in-memory cluster computing framework built on the Apache Spark for processing very large scale geographical data. Apache Spark is a distributed data processing system designed for query processing very large datasets. Two different hot spot analysis tasks have been performed in the project: Hot cell analysis and Hot zone analysis.

KEYWORDS AND PHRASES

Hadoop, Apache Spark, HDFS, Map Reduce, Spatial data, Hot cell analysis, and Hot zone analysis and big data

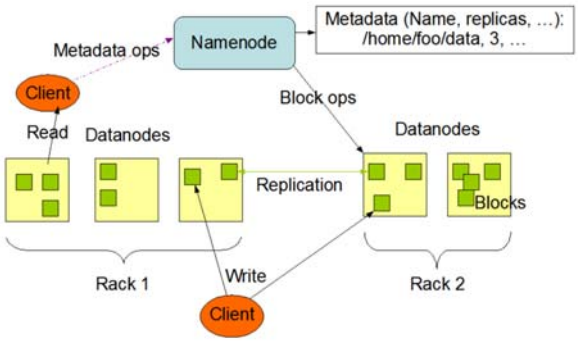
1.INTRODUCTION

The project was divided into divided into 2 phases. The first phase involved setting up a Hadoop and Spark cluster on Google Compute Engine with three n1-standard-1 (1 vCPU, 4 GB memory) Virtual Machines. Our task is to implement geospatial data analysis using a distributed database system using the above mentioned GCE virtual machines. The data contains coordinates of a specific location in the form of latitude and longitude. The distributed database system is a collection of multiple, logically interrelated databases distributed over a computer network[1] The Distributed management system is the software that manages these distributed databases. The distributed database system has advantages like increased Reliability and availability, easier expansion and improved performance. But traditionally, the time, cost and effort for implementing system was a huge challenge which the modern distributed system solutions has made storage and processing of huge amount of data simpler. Some of these distributed database solutions are Apache Hadoop, Spark, Apache Cassandra, Hbase, Couchbase, Amazon SimpleDB.

2. SYSTEM ARCHITECTURE

2.1 Apache Hadoop

Apache Hadoop is an open-source software platform, which provides a way to deal with large datasets. We can store a large amount of data across clusters. It is owned by Apache foundations and is written mostly in Java programming language with some basic utilities in C. Hadoop allows us to store the humongous amount of data, huge processing

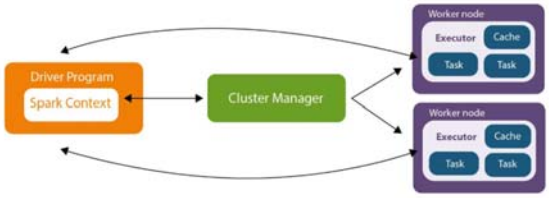


HDFS Architecture

Power and we can perform an immense number of operations simultaneously. It mainly composed of the four essential modules: Hadoop Distributed File System, Hadoop YARN, Hadoop Common and Hadoop MapReduce.

2.2 Apache Spark

Apache Spark is a distributed data processing engine used for multiple purposes. Spark core data processing engine is supported by various libraries for querying, machine learning, graphs and for stream processing. Spark supports many programming languages like Python, Java, Scala, and R. Spark is used for applying query over large datasets which are stored in the distributed database systems like Hadoop.



Apache Spark Architecture

2.3 Scala

Scala is a high-level language that combines functional programming and object-oriented programming. It runs on JVM and can use the existing Java library and code. Apache Spark is written in Scala and has been designed by keeping parallelism and concurrency in mind for big data applications.

3. IMPLEMENTATION OF PHASE-1

The implementation of all parts of the project has been done in Ubuntu 16.04. Passwordless SSH has been implemented for master as well as a worker node. The setup included three virtual machines that are running on Google Compute and all the machines have Apache Spark and Hadoop. Bi-directional passwordless SSH was enabled between these machines and the localhost so that the nodes can communicate with each other.

3.1 Dataset

Two datasets are used

1) arealm10000.csv - Contains a list of coordinates of a point.
2) zcta10000.csv - Contains a list of coordinates of a rectangle.

3.2 Setup

1) Three Google Compute instances are used to set up for the execution of the queries.
2) Use ssh-keygen and a private key to configure passwordless ssh on each Google Compute instance as the first step of the setup as this is required by both Hadoop and Spark for the execution of the queries.
3) Download Java and Hadoop on each instance as each instance leverages functionalities of HDFS and set the JAVA_HOME and HADOOP_HOME.
4) Modify /etc/hadoop/core-site.xml to specify tasks the name node. Do this on all the instances.
5) Modify the /etc/Hadoop/slaves file to specify the IP of all the data nodes on the Master node.
6) Perform a name-node-format when starting the Hadoop system for the first time.
7) Go inside the /sbin/ folder and perform a ./start-dfs.sh to start the distributed file system, on the Master node.
8) Download Spark on the Master node.
9) Modify Spark /conf/slaves.template file to specify the IP of all the slaves, just on the Master node.
10) Execute the command ./start-all.sh i..

3.3 Implementation

Two methods were implemented by us which we described below:

ST_Contains: For this method, we are provided with two inputs: pointString, which are the coordinates of the points and queryRectangle, which are the coordinates of the points lying on the rectangle and both are of String type. So we need to check whether the points are contained within a rectangle and need to return a boolean value(true or false) as the output.

We performed the following steps for this

1. Parse the given Rectangle and Point strings to split based on a ",".
2. Carve out x and y coordinates of the point and rectangle using a for loop for each.
3. Check if the x coordinate of point lies within the rectangle and if y coordinate of point lies within the rectangle, return true if so, false otherwise.

ST_Within: For this method, we provided with three inputs: pointString1, which are the coordinates of the points, pointString2, which are the coordinates of the other points and distance which is the distance constraint for the two above points of the points pointString1 and pointString2 are of String types while distance is of Double type. So we need to check whether the two given points satisfy this distance constraint and need to return a boolean value(true or false) as the output. Following steps are performed for this

1. Parse the two points to split on ",".
2. Store x and y coordinates of each point in an array as the array's 0th and 1st index respectively.
3. Calculate the Euclidean distance between the two points and compare it with the given distance.
4. Return true if the calculated distance is less than or equal to the given distance else return false.

The two implemented methods serve as helper methods to the following queries.

Range Query: Uses ST_Contains.We have as an input the set of points and set of rectangles. We need to find whether the pits are contained within the rectangle.

Range Join Query: Uses ST-Contains. We have as an input the set of points and set of rectangles. We need to find all the pairs(point, Rectangle) in which points are there within the rectangles.

Distance Query: Uses ST_Within. We have as an input a point P and a distance D. We need to find all the points which lie within the distance D from the point P.

Distance Join Query: Uses ST_Within. We have as an input two sets of points s1 and s2 and a distance. We need to find all the pairs(points from set 1, points from set2) such that they lie within the distance given.

4. IMPLEMENTATION OF PHASE 2

In this phase, we need to find the hotspots in the given NYC taxi trip dataset. We need to write the code for two different types of hot spot analysis task. These tasks are hot zone analysis and hot cell analysis. We implemented them in different methods.

4.1 Dataset

This project phase concentrates on the real-world scenario to identify the top 50 hotspots in New York city**.**

Point_hotzone.csv: This contains the pickup point of New York Taxi trip datasets.

zone-hotzone.csv : This contains the zone data.

yellow_trip_(2009-2012).csv: This contains a collection of New York City Yellow Cab taxi trip records.

4.2 Setup:

For this phase, we reused the cluster we created in phase 1.

4.3 Implementation

Hot zone Analysis: In this task, we applied a range join on the given rectangle datasets and the point datasets. The total number of points that lie within each rectangle is calculated which is used to calculate the hotness of the rectangles. More the number of points a rectangle contains, hotter is the rectangle.

We perform the following steps for finding hotness of rectangles:

1. Parse point data and remove the header.
2. Now, read the rectangle dataset.
3. Now, register the ST_Contains UDF.
4. Join the two datasets.
5. Run a Spark SQL query that uses ST_Contains to check if a given point is within the rectangle.
6. The result of this spark SQL is a list sorted by the rectangle, which contains the final results which can be used to compute the hotness of a rectangle.

Hot Cell Analysis: In this phase of the project we are required to apply spatial statistics technique to find the hotspots in Spatio-temporal Data. The provided data is related to cab pickups in New york city within time period from 2009-2012. Here hotspots are the pickup locations and number of pickups in the area contributes towards the hotness. The aim of this phase is to calculate Getis-Ord statistics for NYC Taxi Trip Data. Getis-Ord statistics 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 an attribute value for cell j, $w_{i,j}$ is the spatial weight between cell i and j, n is the total number of cells

Where mean $\underline{X}$ and standard deviations S are

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

We perform the following steps for finding the hot cell:

1) First, we define min, max for latitude and longitude, using which we find the mean.
2) Now, we find the valid points which lie in a given range of min and max. Using a sparkSQL query, we select all the coordinates and add it to the dataframe.
3) Now, we find the mean and standard deviation using the equation mentioned above.
4) Now, we select neighbouring count that lie in the map_dataframe.
5) Now, we calculate the z-score.
6) Now, we use a spark sql query to select the top 50 pickup points by ordering them based on their getis_score.

5. IMPLEMENTATION OF PHASE 3

For Phase 3, we have used Ganglia which is a scalable distributed monitoring system for our cluster of three nodes. We tried to measure the various system performance metrics using Ganglia.

To extract these statistics, we ran the functions we had implemented during the phase 1 and 2 by varying system configuration settings such as number of cores, number of machines, number of executors, driver memory, <size of dataset>. We have included the metrics cpu usage, memory usage, network usage, load per minute, load per 5 mins, free memory, bytes transferred, packets transferred in our analysis.

Analysis of Phase 1 functions:

First Run Configuration: We have used default hadoop configuration for this run. Configuration settings are: Number of executors = 1, executor memory = 1GB , Number of executor cores = 1, driver memory = 1GB.

Second Run Configuration: Configuration settings for this run are: Number of executors = 4, executor memory = 4GB, number of executor cores = 4, driver memory = 4GB.

(In the provided figures) First run is around 41-72 secs and around 136-155 secs in the second run.

Runtime Analysis: For Runtime analysis we computed time spent to execute each function separately using our script.

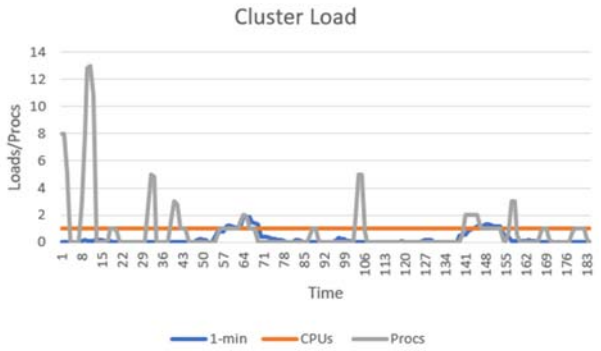
First Run:

Range Query	Range join query	Distance query	Distance join query
19 secs	279 secs	17 secs	143 secs

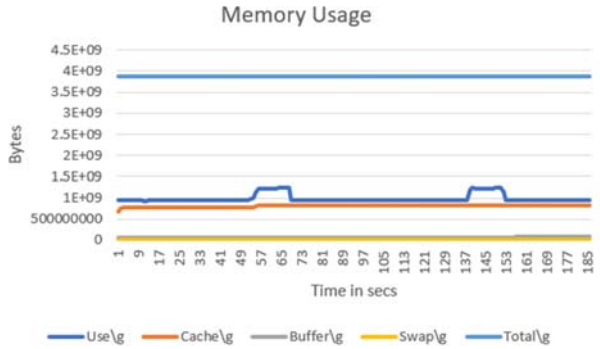
Second Run:

Range Query	Range join query	Distance query	Distance join query
18 secs	300 secs	16 secs	154 secs

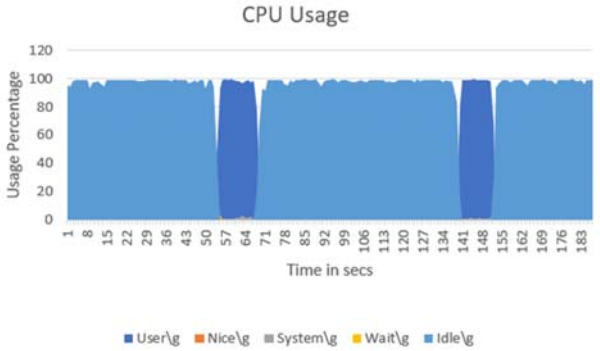
Other Metrics:



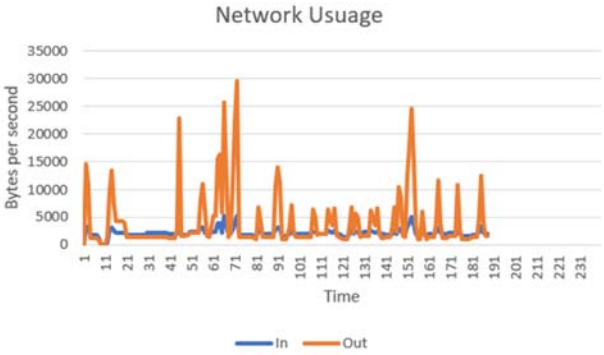
The load on the system was higher during the first run for a longer time and during the second run it was more stable except the beginning and commencement phase of the job. The second run had more processes than the first since the program was running on more than one core.



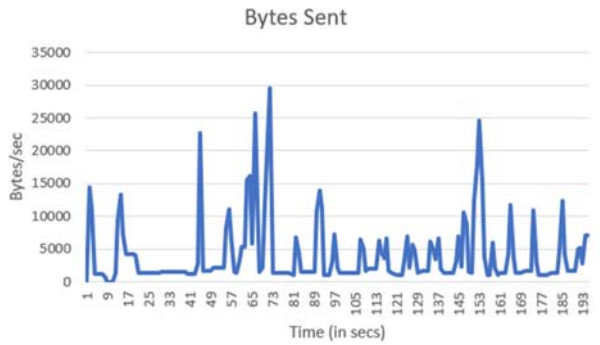
Memory usage is almost similar due same size of dataset.



We found CPU usage was also similar in both the runs.



Network usage peaked during the beginning of the job when spark submit was executed in both runs.

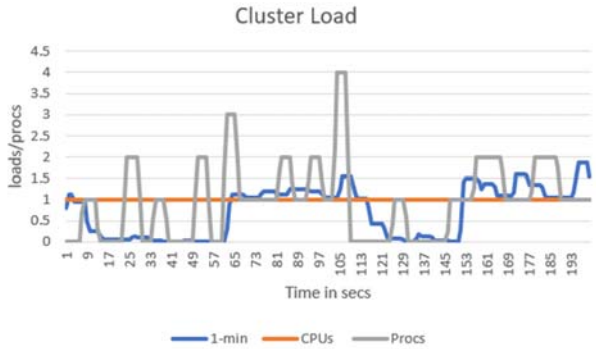


Bytes sent changed during the runs. It was slightly more in the second run.

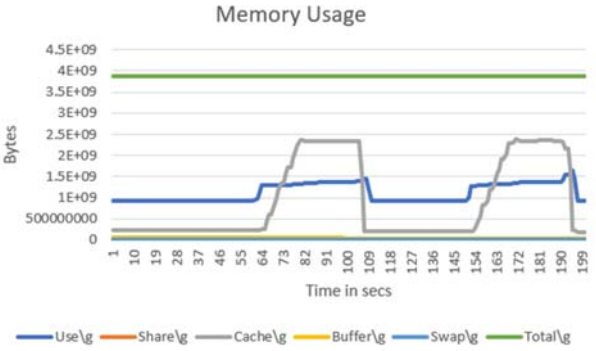
Analysis of Phase II functions:

First Run Configuration: We have used default hadoop configuration for this run. Configuration settings are: Number of executors = 1, executor memory = 1GB , Number of executor cores = 1, driver memory = 1GB.

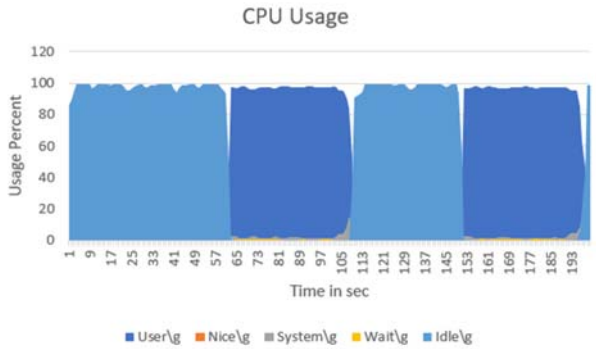
Second Run Configuration: Configuration settings for this run are: Number of executors = 4, executor memory = 4GB, number of executor cores = 4, driver memory = 4GB.
(In the provided figures) First run is around 64-110 secs and around 154-199 secs in the second.



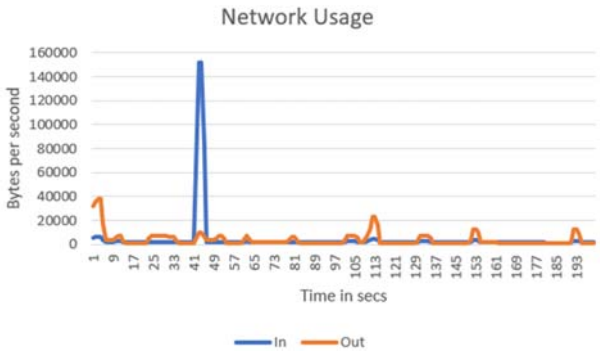
Load peaked multiple times in the first run, while in the second run it peaked only during the end of the run.



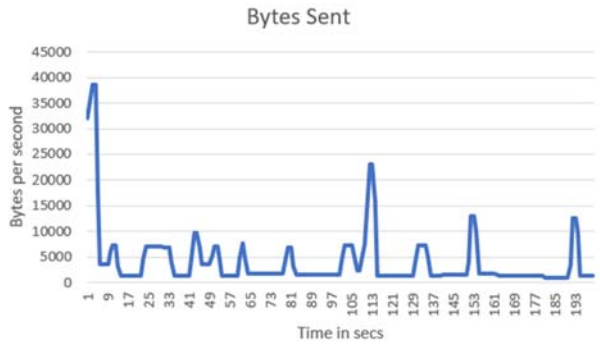
Memory utilization changes when the code runs in different spark configurations. Memory utilization per core was more during the first run due to single core.



CPU utilization: There is little to no change in CPU utilization.



Network performance: Network usage peaked during the beginning of the job when spark submit was executed in both runs.



Bytes sent changed during the runs. It was slightly less in the second run.

Runtime Analysis: For Runtime analysis we computed time spent to execute each function separately using our script.

First Run:

Hot Zone Analysis	Hot Cell Analysis
35 secs	368 secs

Second Run:

Hot Zone Analysis	Hot Cell Analysis
34 secs	358 secs

6 CONCLUSION

After working on this project, we have gained practical experience with the latest distributed database technology Apache Spark. We understood the working of distributed databases and how to distribute computation on various independent machines and handle the big data leveraging the shared nothing architecture. We were introduced to the hadoop file system and gained more insights into configuration settings of Spark system.

7. ACKNOWLEDGEMENTS

We thank Dr Mohamed Sarwat for providing us guidance and assistance throughout the project. We are also thankful to Yuhan Sun for giving the insightful ideas and detailed overview about the project.

8. REFERENCES

[1]http://lig-membres.imag.fr/leroyv/wp-content/uploads/sites/125/2016/09/IntroductionDDBMS.pdf
[2]https://dataconomy.com/2014/02/hadoop-what-how-introduction/
[3]https://en.wikipedia.org/wiki/Apache_Hadoop
[4]https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html
[5]https://mapr.com/blog/spark-101-what-it-what-it-does-and-why-it-matters/
[6]https://www.dezyre.com/article/why-learn-scala-programming-for-apache-spark/198