Geo-Spatial Hotspot Analysis of Large scale datasets using Apache Spark and Framework

Laveena Bachani
School of Computing, Informatics and Decision Systems Engineering,
Arizona State University,
lbachani@asu.edu
1213246721

Shikhar Sharma
School of Computing, Informatics and Decision Systems Engineering,
Arizona State University,
sshar133@asu.edu
1213091904

Krishna Chaitanya Jinka
School of Computing, Informatics and Decision Systems Engineering,
Arizona State University,
kjinka@asu.edu
1213246279

Shashank Kapoor
School of Computing, Informatics and Decision Systems Engineering,
Arizona State University,
skapoo14@asu.edu
1213181604

ABSTRACT

Global Positioning Systems are ubiquitous and with the advent of IOTs, the amount of Geospatial Data that we get today is massive. The problem has motivated us in this project, to study about Geospatial Data Analysis using Spark SQL. The area that we have targeted is Hot Spot Analysis. For that, the dataset that we have is NYC Yellow Cab taxi trip [1] and this project is divided into 4 phases. Every phase will be discussed in detail in this paper.

KEYWORDS

MapReduce, Apache Hadoop, HDFS, Apache Spark, Spark SQL, GIS, Hotspots, Getis Ord

1 INTRODUCTION

MySQL, PostgreSQL, IBM DB2 etc are amazing relational databases which provide sophisticated Structured Query Language. They are centralized having very low latency and you get data from a single node. But, the problem with them is that when it comes to scaling for Big Data and Fault tolerance, they are not the best tools to manage those things. The problem gave rise to Distributed Database Systems where various nodes in cluster contribute together to solve a problem. A breakthrough in Distributed Databases Systems was made by Google: they invented a paradigm called Map & Reduce [2]. After that many tools like Apache Hadoop, Apache Spark, Apache Flink, Apache Zookeeper, Apache Kafka, etc. have been developed which work on Distributed Databases Systems concepts and have improvised a lot on high availability, high reliability, high fault tolerance, and scalability.

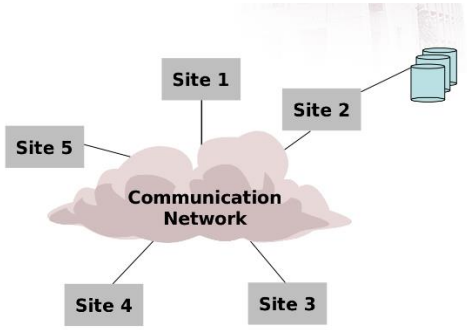


Figure 1: Centralized Database Architecture [5]

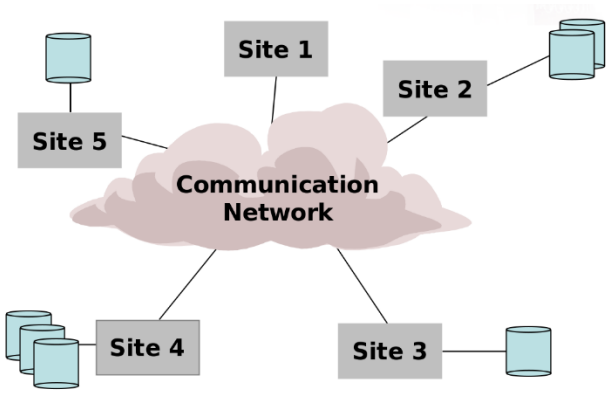


Figure 2: Distributed Database Architecture [5]

Geospatial data contains Global Positioning Coordinates -- Latitude and Longitude -- and temporal data contains time-

related information. To analyze hotspots we are using Spatio-Temporal Data using Spark SQL[3]. The motivation to use Spark SQL is that it is declarative, and a lot of optimization is done by spark itself for the optimized query execution. With that, we can care more about our application output and code optimization is done by Spark. Some other advantages Spark SQL has is that it can integrate with Apache Hive, JSON, Apache Avro etc.

As mentioned our project is divided into 4 phases. Phase 1 of our project was cluster setup of Apache Hadoop/ HDFS and Apache Spark. Phase 2 helped in understanding the nitty-gritty of Spark. In Phase 3 we analyzed hot spots in a month for NYC Yellow Cabs. Phase 4 is a Poster Presentation where we evaluate our project performance.

2 SYSTEM ARCHITECTURE

2.1 Apache Hadoop

The Apache Hadoop[4] software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single server to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

The project includes these modules:

Hadoop Common: The common utilities that support the other Hadoop modules.

Hadoop Distributed File System (HDFS™): A distributed file system that provides high-throughput access to application data.

Hadoop YARN: A framework for job scheduling and cluster resource management.

Hadoop MapReduce: A YARN-based system for parallel processing of large datasets.

It is built in Java and uses HDFS as a data storage. Important components of HDFS are Datanode and Namenode where in our system we have 1 Namenode and 3 Datanode. Namenode allows in the creation of files in the HDFS file system and Datanode provide write-once access many data model. Also, replication factor is an important parameter where we have the replication factor of 3. The version we use for Hadoop is 2.7.4.

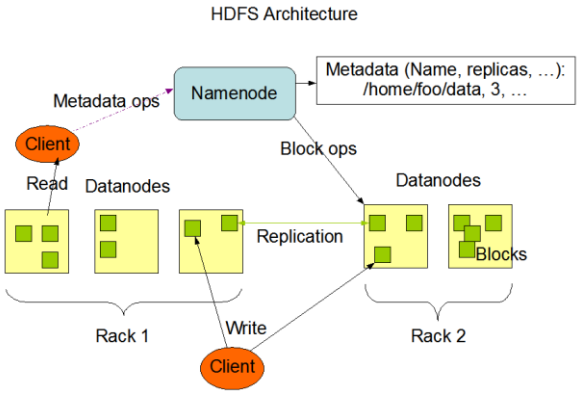


Figure 3: HDFS Architecture

2.2 Apache Spark

Apache Spark[3] is an in-memory data processing engine with elegant development API to allow users to efficiently run SQL jobs. The Hadoop YARN based architecture given Spark to share common Platform and dataset while ensuring consistent levels of service and response.

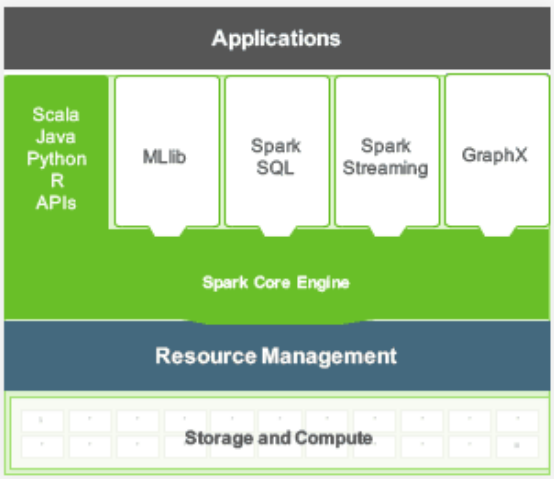


Figure 4 Apache Spark Engine [6]

The main components of Apache spark are:

- Spark core engine
- Set of libraries in Java and Python

Customers are using Spark to improve their business by detecting the patterns. Geospatial Analysis is one of the many uses of Spark. Spark works on RDD (Resilient distributed dataset). A RDD is read-only partitioned collection of atomic records. RDD can only be created through data on stable storage or other RDD. Direct Acyclic graph that contain the flow of operations from one transformation to another.

2.3 Scala

Scala (Scalable Language), a hybrid functional programming language, a mixture of object oriented and functional languages. It got best of both worlds. In this, functions are equivalent to objects in Scala[7] so it is one of the ideal language to use when it comes to scalable software, concurrent or distributed data processing and parallel processing. Scala runs on Java Virtual Machine. This makes Scala general purpose, suitable for data science.

2.4 SBT (Simple Build Tool)

SBT [8] (Simple Build Tool) is an open source build tool for Scala. It provides features such as Native support for compiling Scala code and integrating with many Scala Test Frames and Continuous compilation, testing, and deployment.

3 EXPERIMENTAL SETUP

3.1 Phase 1

Phase 1 was divided into two halves. In the first half, we set up a local cluster on our machines and in the second half we set up a good configuration cluster on powerful mac systems.

To set up the cluster every machine in the cluster should have java installed so as a prerequisite all machines were provided with Oracle Java 8[9]. Java 9 is the current version but at the time of working on this project, there were some issues reported with Java 9 and Spark so we decided to continue with Java 8. In next step, every node was given a name either Hadoop-master or Hadoop-worker1 or hadoop-worker2 and this name convention was written in hosts file; in Ubuntu, it is present at /etc/hosts. After providing this naming convention, a user for Hadoop was created in every system. We call Hadoop user as "hduser" but this is just our naming convention. After this, a password less SSH was set up according to the Figure 5.

The reason for password less ssh setup is to start the master and slaves from one common point. Else, we would have to go to every slave and start the slave manually which is a cumbersome job and hindrance in good software development. Once that was done the setup of Hadoop was required. We used Hadoop 2.7.4 and the location of Hadoop was set to common in all the three machines. To setup Hadoop, three files were manipulated core-site.xml, hdfs-site.xml and mapred-site.xml. These files help in setting up HDFS and Hadoop configurations; Namenode, Datanode, where MapReduce will run, HDFS GUI and other configurations. After this, yarn setup was done with help of yarn-manager.xml.

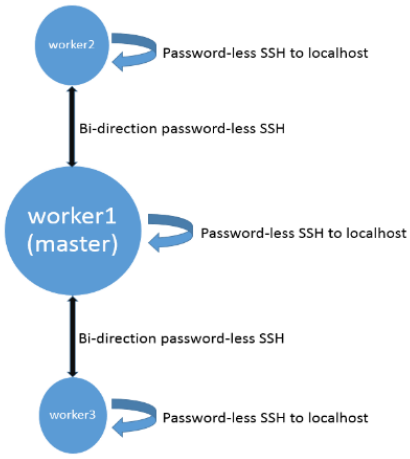


Figure 5: Password less SSH [10]

To set up Spark, hdfs-site.xml and core-site.xml were copied from Hadoop conf to Apache Spark conf. The beauty of Apache Spark is that it runs on top of the Hadoop also and that is what we are leveraging here. The configurations are as follows for two environments we had.

	Node 1	Node 2	Node 3
OS	Ubuntu 16.04	Ubuntu 16.04	Ubuntu 16.04
Hard Disk	100 GB	100 GB	50 GB
Ram	4 GB	8 GB	8 GB
JAVA	Oracle 8	Oracle 8	Oracle 8
Spark Version	2.2.0	2.2.0	2.2.0
Hadoop Version	2.7.4	2.7.4	2.7.4
Cores	4	8	8

Table 1: Configurations of Local Cluster (Used Only for Phase 1)

3.2 Phase 2

In Phase 2[11] we learned about, in-build libraries of Spark and Scala. This task was the foundation for the Phase 3. Our task was to implement two functions in Scala ST_Contains and ST_Within and using them implement the four queries.

We used the above two functions for implementing the following queries.

- **Range_Query**: Query to return all the points that lie within in the given rectangle.
  *select * from point where*
  ST_Contains(givenRectangle, point)
- **Range for Join**: Query to return all pairs of rectangle and points such that the point lie in the rectangle for given set of rectangle and points.
  *Select * from rectangle,point where*
  ST_Contains(rectangle,point)
- **Distance query**: Given a point P and Distance D, find all the points that are in distance d from point P.
  *Select * from point where ST_Within(point1, givenPoint, givenDistance)*
- **Distance Join Query**: Given a set of Points S1 and set of Points S2 and Distance D in km, find all points pairs(s1,s2) where S1 is distance D from S2.
  *select * from point where ST_Within(point1, point2, givenDistance)*

We implemented Range_Query and Range for Join with ST_Contains and Distance query and Distance Join Query with ST_Within.

Algorithm for ST_Contains

Input: QueryRectangle(x1,y1,x2,y2), PointString(x,y)
Output: True if PointString lies in QueryRectangle

Algorithm
minX = min(x1, x2)
maxX = max(x1, x2)
minY = min(y1, y2)
maxY = max(y1, y2)
if point(x) lies inside the range minX <= x <= maxX and if point(y) lies inside range minY <= y <= maxY then return true else false

	Node 1	Node 2	Node 3
OS	Mac OS	Mac OS	Mac OS
Hard Disk	250 GB (SSD)	250 GB (SSD)	250 GB (SSD)
Ram	8GB	8GB	8GB
JAVA	Oracle 8	Oracle 8	Oracle 8
Spark Version	2.2.0	2.2.0	2.2.0
Hadoop Version	2.7.4	2.7.4	2.7.4
Cores	8	8	8

Table 2: Configurations of Testing Server (Used in Phases 2 and 3)

Algorithm for ST_Within

Input: PointString1(x1,y1) , PointString2(x2,y2), distance
Output: True if distance between PointString1 and PointString2 is less than or equal to distance else false

Algorithm
distanceBetweenPointStrings = SQRT(POW(x1-x2, 2) + POW(y1-y2, 2))
If distanceBetweenPointStrings<=distance return true else false

3.3 Phase 3

The motivation behind the task is to analyze the data for the New York taxi cabs to return the area where maximum cabs have taken using Apache Spark: frequent places where cab is taken. We were asked to solve two problems, hot-zone Analysis and hot-cell analysis. While performing the task, GIS helps to provide the geographical information of the area where we need to find out the best zone. To identify the best cluster, density plays a crucial factor because it tells the area of the cluster. Only issue with the density is that it won't explain if the cluster is significant. To find out the solution, we need to opt a method which will tell the correct statistics of the cluster. Approach is to find a threshold and then calculate value among all identified clusters. The cluster with the maximum value will be the best choice to select.

Cluster analysis is performed with a very common and efficient approach, called Getis-Ord measurement. This calculates a z-score and p-score value. If the value of z-score is high and p-value is small, then the cluster or cell is important hotspot. It is based on the z-score, if it tends to zero then that means there is no clustering possible.

Hot-Zone Analysis

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.

Algorithm Hot-zone Analysis:

Input: Point data, the input point dataset is the pickup point of New York Taxi trip datasets. Zone data (only for hot zone analysis): at "src/resources/zone-hot zone" of the template.
Output: All zones with their count, sorted by "rectangle" string in an ascending order.

Algorithm

Cross Join on Rectangle and Point and select all those tuples in which point lies in the rectangle (We use ST_Contains which we implemented in Phase 2):
select rectangle._c0 as rectangle, point._c5 as point from rectangle, point where ST_Contains (rectangle._c0 , point._c5)

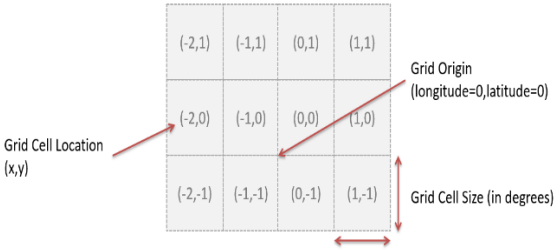
Count the number of points the come in the rectangle:
select rectangle,count(point) as count from joinResult group by rectangle order by rectangle asc.
Return the tuples

Hot-cell Analysis

This task will focus on applying spatial statistics to spatio-temporal big data to identify statistically significant spatial hot spots using Apache Spark.

Some Definitions

Cube:



Space – Each cell size is in degrees and the origin for this cube is at 0,0

Time – time step size is defined in days with the origin being at the first of the month

Getis-Ord

The Getis-Ord G*I statistic is calculated as below

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

Mean(X) and Standard Deviation(S):

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

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

Constraints: The time and space are combined into cubic cell as shown below. They are combined to form a space-time cube.

Algorithm Hot-cell Analysis:

Input: Latitude and Longitudes of the Pickup Location and the date of the Pickup.
Output: Reverse Sorted based on Getis Ord/Zscore/Hotness
Constants:
coordinateStep = 0.01
minX = -74.50 / coordinateStep
maxX = -73.70 /coordinateStep
minY = 40.50 / coordinateStep
maxY = 40.90 / coordinateStep
minZ = 1
maxZ = 31
numCells = (maxX - minX + 1) * (maxY - minY + 1) * (maxZ - minZ + 1)

Algorithm

1) Convert Latitude, Longitude and Day in x,y,z coordinates
2) x: floor(latitude/coordinateStep(0.01 default))
   y:floor(longitude/coordinateStep(0.01 default))
   z: day of the month

3) Count how many times a particular cell is Hit. That becomes the attribute of this cell:
select x, y, z, count(*) as attr from PickUpLocation group by x,y,z
We call this view, AttributeList
4) Find Mean and Standard Deviation for the attribute list
summations = *select sum(attr),sum(attr*attr) from AttributeList*
Mean = summations sum(attr) / numCells
Standard Deviation = sqrt(summations.get(1) / numCells) - (Mean * Mean))
5) cross join on Attribute list and find the hotness of the cell based on its neighbours

```
countValidNeighbourCount(x,y,z) => {
  initialize neighbour as 0
 for (i <- x - 1 to x + 1) {
      for (j <- y - 1 to y + 1) {
        for (k <- z - 1 to z + 1) {
          if (i >= minX && i <= maxX
            && j >= minY && j <= maxY
            && k >= minZ && k <= maxZ) {
            increment  neighbour
          }
        }
      }
  }
 return neightbour
}

validNeighbour =>  (x1, y1, z1, x2, y2, z2) => {
          if (x2 > maxX || x2 < minX || y2 < minY || y2 >
          maxY || z2 > maxZ || z2 < minZ) return false
          else if ((x1 - 1 == x2 || x1 == x2 || x1 + 1 == x2) &&
          (y1 - 1 == y2 || y1 == y2 || y2 == y1 + 1)
          && (z1 == z2 || z2 == z1 - 1 || z2 == z1 + 1)) return
          true
          else return false
}
```

select first.x as x, first.y as y, first.z as z, sum(second.attr) as nrcount, countValidNeighbourCount(first.x,first.y,first.z) as ncount from AttributeList first cross join AttributeList second where validNeighbour(first.x,first.y,first.z,second.x,second.y,second.z) group by first.x,first.y,first.z
6) Finally, sort based on Getis-Ord

```
giveGetisOrd(nrCount, ncount) = {
          numerator = nrCount. - xBar * ncount
          partDenom = math.sqrt(((numCells * ncount) –
          (ncount * ncount)) / (numCells - 1)) * sValue
          return numerator / partDenom
}
```

select x, y, z, getisord(nrcount, ncount) as zscore from NeighboursWith order by zscore desc

return only x, y, z coordinates.

PRESUMPTIONS:

There are certain assumptions that we will consider in this project

- Each cell size formed in the space time cube is 0.01 * 0.01 in terms of both latitude and longitude.
- Time step size is one day i.e. (0-30 as January has 31 days).
- We consider only pickup timestamp, pickup latitude and pickup longitude attributes of the dataset to calculate the z-score.

4 EXPERIMENTAL EVALUATION

We have used Spark GUI and Spark History Server to calculate these metrics.

4.1 Phase 2

Query: *spark-submit --master spark://hadoop-master:7077 --class cse512.SparkSQLExample --total-executor-cores <Variable> --num-executors <Variable> --executor-memory <Variable> target/scala-2.11/CSE512-Project-Phase2-Template-assembly-0.1.0.jar hdfs://hadoop-master:54311/result/output rangequery hdfs://hadoop-master:54311/src/resources/arealm10000.csv -93.63173,33.0183,-93.359203,33.219456 rangejoinquery hdfs://hadoop-master:54311/src/resources/arealm10000.csv hdfs://hadoop-master:54311/src/resources/zcta10000.csv distancequery hdfs://hadoop-master:54311/src/resources/arealm10000.csv -88.331492,32.324142 1 distancejoinquery hdfs://hadoop-master:54311/src/resources/arealm10000.csv src/resources/arealm10000.csv 0.1*

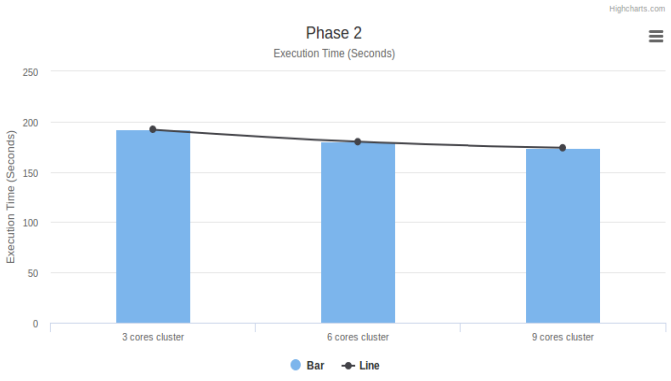


Figure 6**: **Execution Time

Execution time for Phase 2 queries were 3.2, 3 and 2.9 mins on 3, 6 and 9 core cluster. The results were that good because

we are using SSD memory and wherein our local cluster it took more than 6-7 mins in every job no matter how many cores we use. Also, as we increased the number of cores the execution time reduced. With this experiment, we also got hands-on experience on manipulating our spark resources in the cluster. This is important because we have to manage memory wisely in the system.

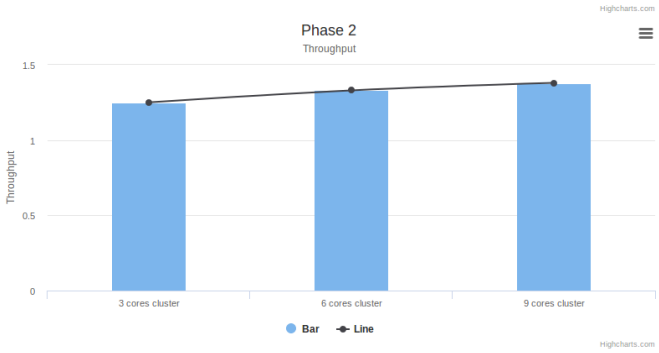


Figure 7: Throughput of Phase 2 queries (4 Jobs)

Throughput came out to be 1.25, 1.33, 1.38 on 3, 6 and 9 cores cluster. Even though this was a small-scale test, but this helped us understand that the increase was not that much. We analyzed the reasons behind it and the result was that shuffles are important parameters which we must keep a check on in a distributed environment. These were not allowing much increase in throughput.

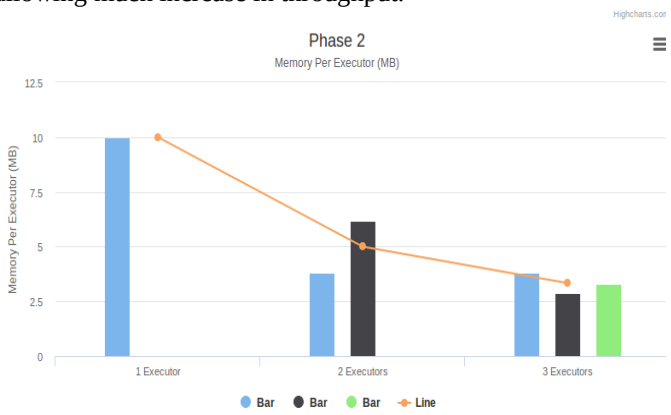


Figure 8: Memory Per Executor (Phase 2)

As we had three workers (we spawned a Worker node on master also), the figure shows how the load was divided among all the executors. We altered from the spark-submit how many executors should work for the job. The more the executors we had the less the load was there on an executor.

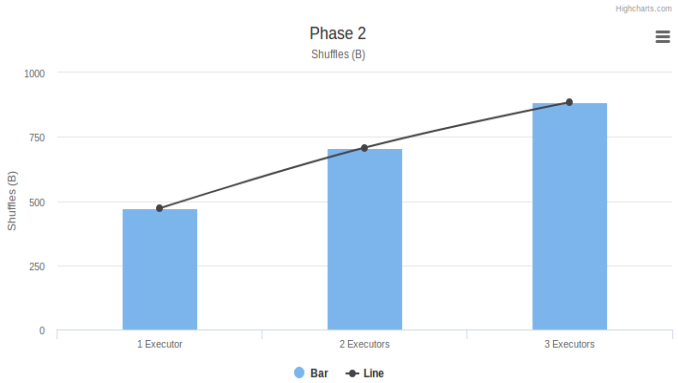


Figure 9: Shuffle read and write (Phase 2)

Figure 9 shows that the more executors we had the more shuffle the system had. Which is bad in Distributed Programming.

4.2 Phase 3

Hot-zone

Query: *spark-submit --master spark://hadoop-master:7077 --class cse512.Entrance --total-executor-cores <Variable> --num-executors <Variable> --executor-memory <Variable> target/scala-2.11/CSE512-Hotspot-Analysis-Template-assembly-0.1.0.jar hdfs://hadoop-master:54311/test/output hotzoneanalysis hdfs://hadoop-master:54311/src/resources/yellow_tripdata_2009-01_point.csv hdfs://hadoop-master:54311/src/resources/zone-hotzone.csv*

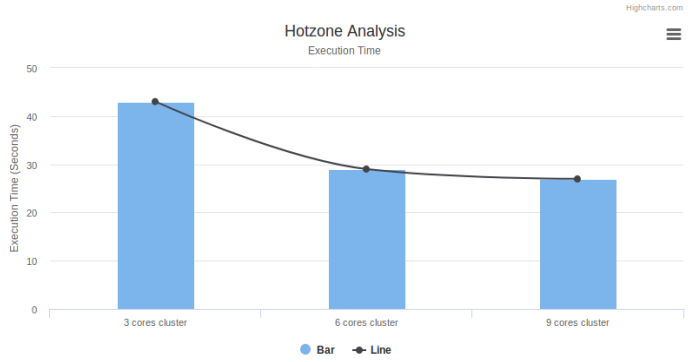


Figure 10: Execution Time for a single query

The execution time for the hot zone was 40, 29 and 26 seconds on 3, 6 and 9 cores cluster. This result aligns with what we had in Phase 2 for execution time.

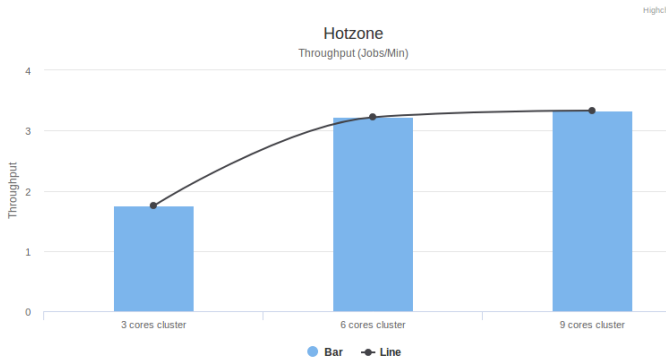


Figure 11: Throughput using different number of core (Hotzone) for 10 Jobs

The throughput for 10 Jobs in Hotzone analysis came out to be 1.75, 3.22, 3.33 in 3,6 and 9 cores cluster. The result again aligns with what we had in Phase 2

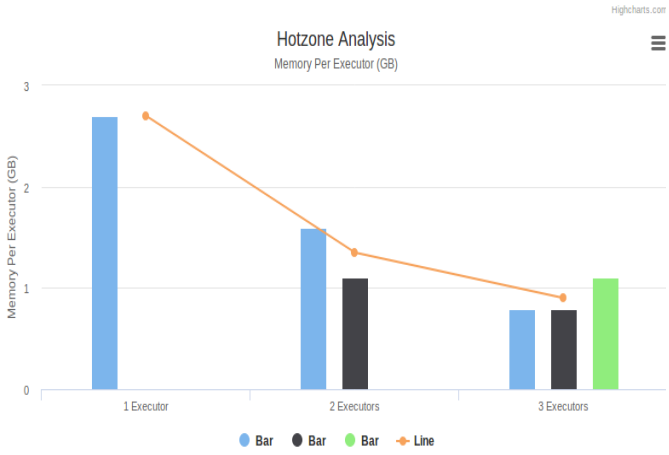


Figure 12: Memory per executor for a job

Figure 12 shows how the memory usage was divided when we spawned a query on 1, 2, or 3 executors. The result aligns with what we had in Phase 2

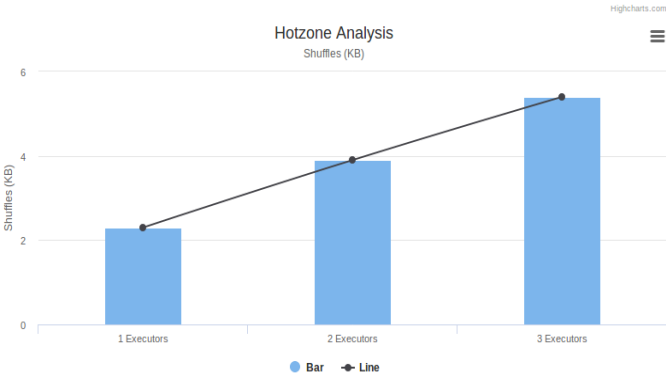


Figure 13: Shuffle in different cores for a job

Figure 13 shows that as we increased the number of cores, the shuffle read and write increased in the Hotzone analysis. The result aligns with what we had in Phase 2

Hot Cell Analysis

Query: *spark-submit --master spark://hadoop-master:7077 --class cse512.Entrance --total-executor-cores <Variable> --executor-memory <Variable> target/scala-2.11/CSE512-Hotspot-Analysis-Template-assembly-0.1.0.jar hdfs://hadoop-master:54311/test/output hotcellanalysis hdfs://hadoop-master:54311/src/resources/yellow_tripdata_2009-01_point.csv hotcellanalysis hdfs://hadoop-master:54311/src/resources/yellow_tripdata_2009-02_point.csv*

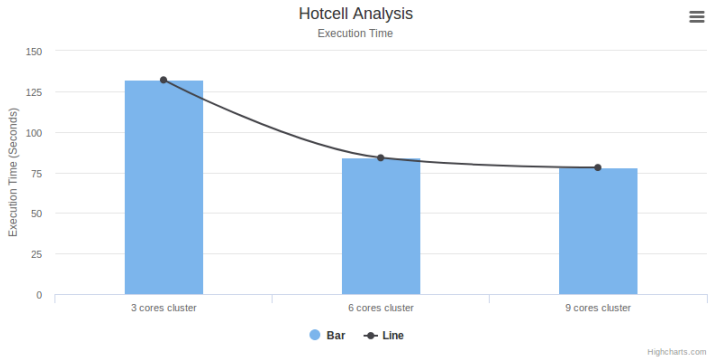


Figure 14: Execution Time for hot cell analysis for two jobs

Figure 14 Shows that as we increased the number of cores to do the spark job, the time to execute the queries decreased. The result aligns with what we had in Phase 2

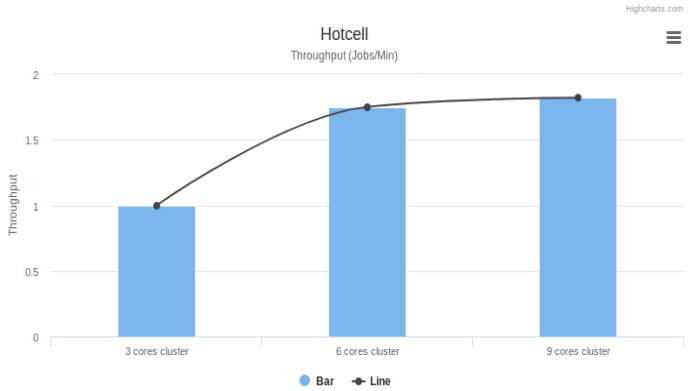


Figure 15: Throughput for hot cell analysis for 10 jobs

Figure 15 shows that as we increased the number of cores to run 10 Jobs the throughput of the system increased. But not much change was visible in 9 cores cluster and 6 cores

cluster. But, if we increase the number of jobs to run, there can be a significant difference in throughput. The result aligns with what we had in Phase 2

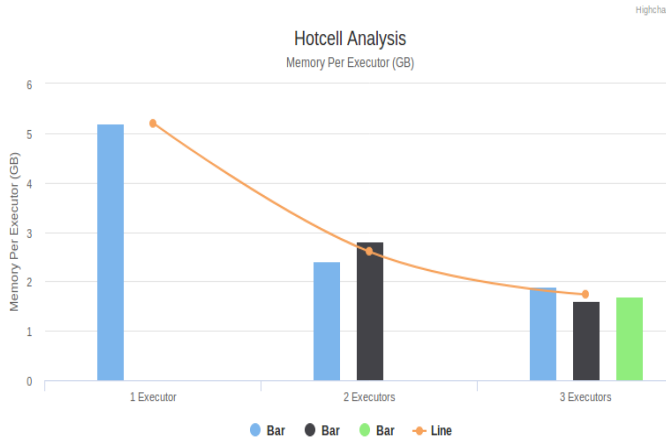


Figure 16 Memory per executor for 2 jobs

Figure 16 shows that the more the executors we had, the less the burden was there on an executor to process the jobs.

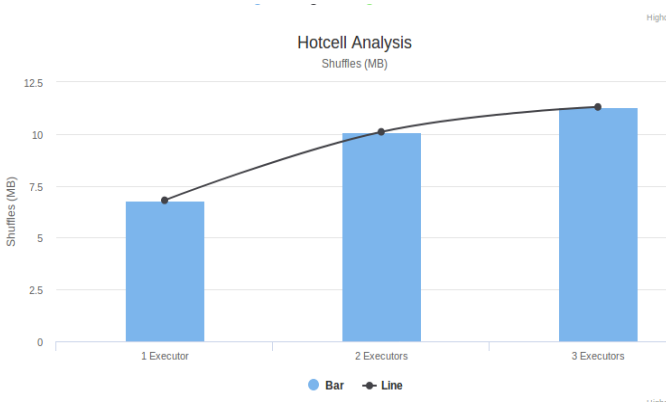


Figure 17: Shuffles in Hotcell Analysis(2 Jobs)

The more the executors where there, the more shuffle happened. The result aligns with what we had in Phase 2

5 CONCLUSION

In the project, we used distributed system to solve the problem of finding hot zones in New York City. To accomplish the task, we have used Hadoop and Apache Spark. In the first phase, the Hadoop distributed system was set up and Geo-Spark queries were executed. In the second phase, we have used the range query to return true if the point is present in the rectangular area. In phase three, we have identified the top fifty hot zones in New York City for taxi pickups. To identify the best spots, Getis-Ord score was calculated for each cell. A Higher value of Z-score concluded that the cell has more density and returned the coordinates of top fifty cells.

Spark SQL is an amazing tool to do Geospatial Data Analysis as because a lot of optimization is done by the Spark SQL executor.

Spark has a module called Tungsten which does the optimization for the DAG execution and recently there has been some improvement in Spark 2 which have optimized Spark SQL also.

6 FUTURE SCOPE

We have not used indexing and partitioning in this project which could have further improvised on the execution time and throughput but would have asked for more memory from the system. But this can definitely be tried out as a further experiment of this project. Also, we haven't leveraged the full power of Spark for Geospatial data analysis. Tools like Geospark can help us in doing that.

7 REFERENCES

[1] NYC Taxi dataset was taken from the repository of Data Sys Lab: https://datasyslab.s3.amazonaws.com/index.html?prefix=nyctaxitrips/

[2] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters, https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

[3] Apache Spark SQL: https://spark.apache.org/

[4] Apache Hadoop: http://hadoop.apache.org/

[5] Lecture 1 Slides by M Sarwat https://myasucourses.asu.edu/bbcswebdav/pid-16723323-dt-content-rid-107552374_1/courses/2017Fall-T-CSE512-82013/CSE512_Lecture1.1.ppt?globalNavigation=false

[6] Apache Spark Architecture: https://hortonworks.com/apache/spark/#section_1

[7] Scala, Object-Oriented Meets Functional: https://www.scala-lang.org

[8] SBT, The interactive build tool, http://www.scala-sbt.org/

[9] Oracle Java 8: https://www.oracle.com/java/index.html

[10] Jia Yu, Hadoop Distributed File System (HDFS) + Spark Integration Guidance https://myasucourses.asu.edu/bbcswebdav/pid-16512097-dt-content-rid-106924535_1/courses/2017Fall-T-CSE512-82013/HadoopDFS%2BSparkIntegrationGuidance%284%29.pdf

[11] Jia Yu, CSE512-Project-Phase2-Template https://github.com/jiayuasu/CSE512-Project-Phase2-Template/blob/master/README.md

[12] ACM SIGSPATIAL CUP 2016 http://sigspatial2016.sigspatial.org/giscup2016/submit

[13] Jia Yu, Jinxuan Wu, Mohamed Sarwat, GeoSpark: A Cluster Computing Framework for Processing Large-Scale Spatial Data
https://pdfs.semanticscholar.org/6d34/20dfdf0a7b60d61ef2eac01332611c870663.pdf