Hot Spot Detection from Geo Spatial Temporal Data using Apache Spark

Aditya Prasad Mishra
(1211165121)
amishr28@asu.edu

Pratik Mishra
(1213076694)
pmishr14@asu.edu

Nikita Shanker
(1211212766)
nshanker@asu.edu

Sarvarth Bhatnagar
(1213114771)
sbhatn12@asu.edu

ABSTRACT

The geolocation and time information offered by the Presently Global Positioning System (GPS) is used by a plethora of data processing engines for solving corresponding business problems. The objective of this project is to use this geolocation data and demonstrate a hotspot detection module which will examine the density or hotness in a particular set of area coordinates. The data source is New York City Yellow Cab.

Keywords

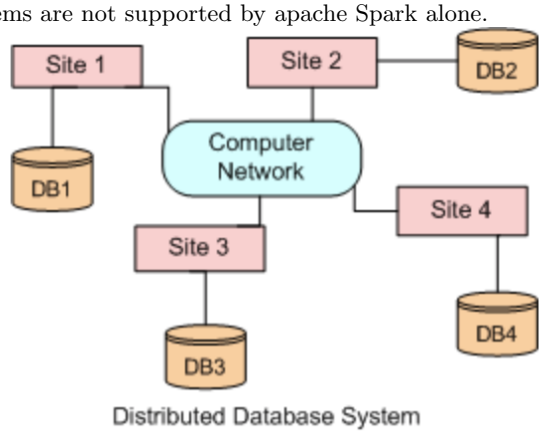
GeoSpark, MapReduce Algorithms, Geo Spatial Data, Hadoop, Spark, Parallel and Distributed DBMS

1. INTRODUCTION

Distributed database systems evolved from the era of centralized database system and provided a significant advantage over the former. In Centralized system, data was stored only on a single machine which provided some advantages like high data integrity, and high availability on one machine yet the disadvantages like network bottleneck, fault tolerance led to the adoption of distributed approach. In distributed systems, the data is sliced and stored in multiple data nodes and the sliced data can have many copies in a network. This maintains a high data availability and also helps in achieving scalability of the system which was chiefly missing in the centralized approach.

Distribute database systems has led us Big-Data. The power to store data which was considered unnecessary earlier due to low caapcity of database systems. Big-Data is everywhere. From it's application in AI , Machine learning to doing Spatial Analysis to analyzing Scientific data, it's application is varied. In a few years from now it will become a necessity like electricity and internet. As we approach towards singularity, we will see data burst happening each year. As per [http://www.businessinsider.com/mind-blowing-growth-and-power-of-big-data-2015-6] "The amount of digital data in the universe is growing at an exponential rate, doubling every two years, and changing how we live in the world".Also, from the same link, only 0.5% of big data is currently in use. Motivation for this project came from our earlier experience with big data. Most of us have seen how big data traversal is difficult and sometimes impossible with SQL and RDBMS. While working professionally,many of us in this team faced similar issue. Also recent application of big data to predict presence of Higgs-Boson [https://home.cern/topics/ higgs-boson] or predict cyclone Phailin [https://en.wikipedia.org/ wiki/Cyclone/Phailin] accurately has made it a tool for human development and survival. Proper analysis of big-data is not only a privilege it's a necessity. Nowadays we are getting more information from our space probes than we can analyze. Just think about the speed of research, if we can analyze such data. There is no doubt big data will help us to find our neighbouring Alien Civilization in near future. It will also answer the "Theory of Everything" conundrum.

For processing of multidimensional Geospatial Big-Data set on a distributed database system we need technologies like Hadoop, Apache Spark. In this paper, we will demonstrate and analyze the use of GeoSpark for resolving geospatial problems. Geospark on integrating with Apache Spark and Hadoop can be used to answer such problems as these problems are not supported by apache Spark alone.



Distributed Database System

Chronologically, Phase 1 of the project was dedicated to setting up Apache and Hadoop on a distributed system by

configuring the master and data nodes in a cluster. Phase 2 was to demonstrate the operations and architecture of GeoSpark. The Phase 3 describes hotspot and hot zone detection algorithm implemented using Scala programming language.

Data Abstractions is achieved by spark using itâĂŹs RDD (Resilient Distributed Datasets), which minimizes the overall network cost and achieves consistent results. The support for iterative algorithms helps querying for data in a loop with these immutable structures. There are two kinds of functions which can be done on an RDD -

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

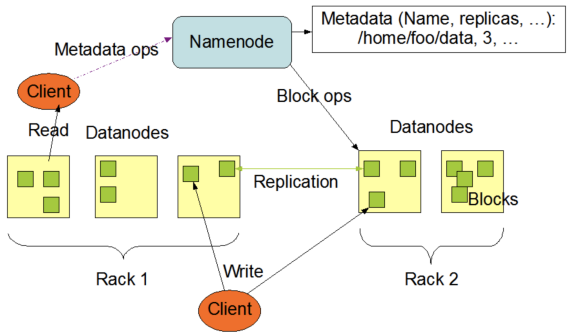
2. SYSTEM ARCHITECTURE

In this section we would outline and discuss the set of technologies used for computing of a geospatial problem in a distributed environment.

2.1 Apache Hadoop

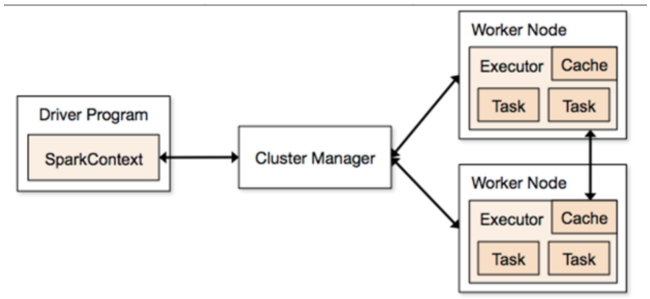
The concept of Hadoop sprung from the paper of Google on google file system and Map- Reduce. Apache came up with an open source software to leverage these concepts and provide a scalable option for computing data in a distributed environment. It provided a reliable and fault tolerant architecture and helps in achieving a high performance. For storing data Hadoop makes use of its own file system internally as default file system - HDFS. Hadoop works on a write once and read many concept, providing a transparency to the users that they are reading and writing on a single source. HDFS is internally fragmented into blocks and these blocks are stored in the participating data nodes. Hadoop achieves its high availability and data backup using its replication factor which for our cluster is set to 3. Version used for Project : Hadoop 2.7.3

HDFS Architecture



2.2 Apache Spark

Apache Spark is a cluster based computing framework which is open source and helps in achieving data parallelism. The underlying technology of Spark is RDD that is, Resilient Distributed Database which gives a higher performance over the traditional linear MapReduce structure through its shared memory approach. Spark supports Hadoop Yarn and has its own standalone cluster as well. For storage in a distributed environment it supports a wide range of systems like Hadoop HDFS, Cassandra and others.



2.3 Spark SQL Overview

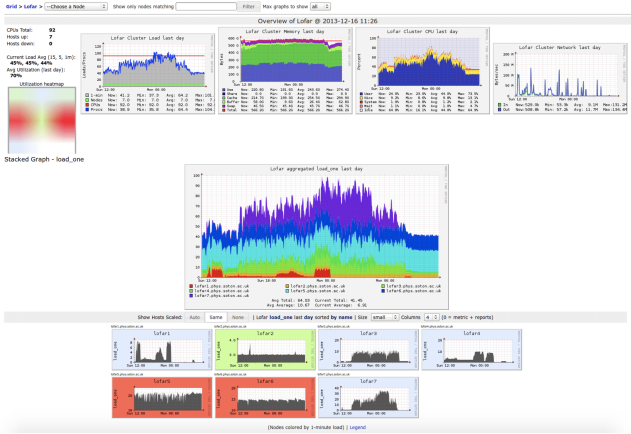
Spark SQL provides the capability of querying structured and semistructured spark data through SQL or through dataframe apiâĂŹs. In this project we are querying the structured data through dataframe api in scala. We get the capability to perform extra optimizations as more computational information is provided through this API. DataFrames are structurally similar to a relational database and are in actual RDD of row objects. These are stored more efficiently as compared to native RDDs.

2.4 ThoTh Lab

ThoTh Lab is used for achieving a computing capacity which is re-scalable and available on cloud. This is used in our project phase 2 and 3 to help achieve a continuous up time.

2.5 Ganglia

Ganglia is used in Project Phase 3 for monitoring the computing performance in distributed environment. It is used to achieve graph-based reports of the statistics which include average CPU load and node-level network utilization.



2.6 Scala

Scala that is Scalable language works on an objected oriented approach and is the one being used in Project Phase 3 for GeoSpark operations. Since Spark is built over scala hence it is the most ideal language for coding in GeoSparkâĂŹs parallel processing and data processing operations.

3. GEO SPATIAL ANALYSIS

For GeoSpatial Analysis the Phase3 of the project makes use of Apache GeoSpark. It is built over Apache Spark and the key advantage is that it uses its in-memory cluster computing capability for handling geospatial data.

3.1 GeoSpark Architecture

GeoSpark comprises of three primary layers.

- Apache Spark layer - Provides essential functionalities including data loading which is used by RDD operations.
- Spatial RDD layer - Provides key Spatial RDD objects - PointRDD, PolygonRDD and RectangleRDD
- Spatial Query Processing layer - For executing spark query with the facility of spatial indexing.

Different types of spatial queries executed are:

3.1.1 Spatial Range Query

The source geospatial data is loaded and then partitioned using the signature RDD partitioning. If necessary, the spatial indexes are created on this partitioned data by Spark. Now whenever a spatial range query is run it will figure out from these partitions if any data block contains a coordinate which belongs to the specified range.

Algorithm: SpatialRangeQuery

Input: Query Window Envelope
Output: Count of points in the Envelope

```
1. Define an envelope corresponding to the input.
2. ObjectRDD read from HDFS should be converted to
PointRDD
3. R-Tree if required should be made on  ObjectRDD.
4. For calculating spatial range, run GeoSpark API
RangeQuery.SpatialRangeQuery()
5. For using same via index, run RangeQuery.
SpatialRangeQueryUsingIndex().
6. Run and return back the result count of step 4/5.
obtained in the step4.
```

3.1.2 Spatial Join Query

This is used to join the results of of ObjecRDD and PointRDD as per the predicate used. Post SRDD partitioning, local SRDD indexing may be executed. The predicate is grid IDs and in case it is same then both results are kept. The result type is Point.Point which is data stored in form of a rectangle data. In the end, the algorithm will take care of the duplicates and the action is then performed on the required coordinates to store them.

Algorithm: SpatialJoinQuery

Input: HDFS path of both the inputs at the start.
Output: Result size

```
1. Define an envelope corresponding to the input.
2. Read from HDFS path1 the target CSV and populate
the coordinate information in PointRDD.
3. R-Tree if required should be made on  ObjectRDD.
4. Read from HDFS path1 the target CSV and populate
the coordinate information in RectangleRDD.
5. Run Join query on Point Rdd and Rectabgle RDD that
is, JoinQuery(scalaContext, PointRDD, RectangleRDD).
6. Run and return back the result size
```

3.1.3 Spatial KNN Query

This query makes use of a top-K algo, which is heap based and calculates the top-k nearest elements in the heap. Spatial KNN Query comprises of two phases:

In case SRDD Indexing existed on local partitions, it would increase performance over the R-Tree index approach.

Algorithm: Spatial KNN Query

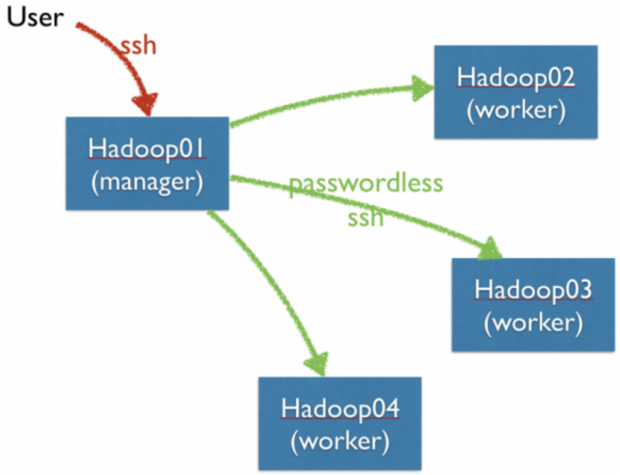
Input: Geological location via HDFS.
Output: k-neighboring coordinates

```
1. Create Point RDD from input and convert it into
Object RDD.
2. If specified, create R-Tree over ObjectRDD
3. If specified, create R-Tree over PointRDD
4. Run KNNQuery.SpatialKnnQuery() or KNNQuery.
SpatialKnnQueryUsingIndex()
API where the latter queries using Index.
6. Return K-nearest coordinates.
```

4. PROJECT PHASE I

Project Phase1 was based on understanding the key distributed database system concepts. Hadoop was configured in a distributed environment and master-worker was established on the participating machines. The composition was 1 Master and 2 Workers. Spark was installed on top of Hadoop on all the nodes. We were introduced to GeoSpark which is used for operating on geospatial data and uses SRDDs which is an extension of RDD.

After doing the initial multi-node cluster setups on our clusters, we executed the Geospark spatial queries and captured the performance using Ganglia tool.



: Structure of nodes configuration

4.1 Experimental Setups

Datasets: Following are the datasets that were used for our experiments."arealm.csv" was used to create the PointRDD distributed dataset and "zcta510.csv" was used to create the RectangleRDD dataset which were used to run our GeoSpark Spatial queries.

Cluster Settings: Following were the configurations being used for our multi-node hadoop cluster settings. All of the clusters were running the same Ubuntu version. Master node cluster had a slightly higher disk space compared

Table 1: Datasets

Dataset	Size
arealm.csv	2.7 MB
zcta510.csv	1.4 MB
zcta510-small.csv	4 KB

to the rest.But the computation power across all the nodes were equal.

Table 2: Cluster Configuration

Machine	Processor	Disk Space	OS
Master	2.8 GHz	40 GB	Ubuntu 16.04
Worker 1	2.8 GHz	30 GB	Ubuntu 16.04
Worker 2	2.8 GHz	30 GB	Ubuntu 16.04
Worker 3	2.8 GHz	30 GB	Ubuntu 16.04

Evaluation Metrics

We are evaluating our experiments based on the performance by varying Spark cores to measure execution time, throughput, memory usage and effect of varying iterations on the average execution time.

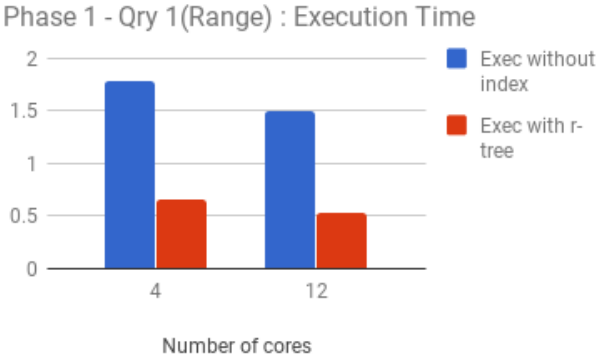
4.2 Experimental Results

For each of the three queries, we did the below experiments individually, by observing the readings for each query first without indexing and then with r-tree index.
To calculate the execution time, we ran each query by executing scala code in spark shell using Geospark api and calculated the time elapsed based on start time and end time for a program. And we again observed our readings by running the query after changing the cores from 4 to 12.
Similarly, for measuring the Peak Memory, we observed the ratings on the Ganglia graph for mem_report for cores 4 and 12. For Throughput, we observed the Ganglia readings for the proc_run graph and for Iterations, we ran the code three times by running the code iteratively for 10, 100 and 1000 times.

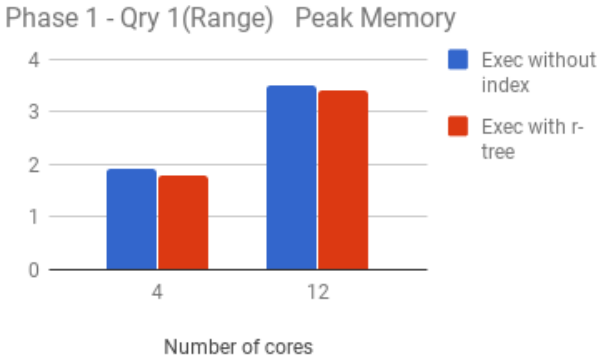
Finally, we combined the ratings of both indexed and non-indexed queries in corresponding individual query graphs to compare their perfomance against the change in cores and with the indexing.

4.2.1 Spatial Range Query

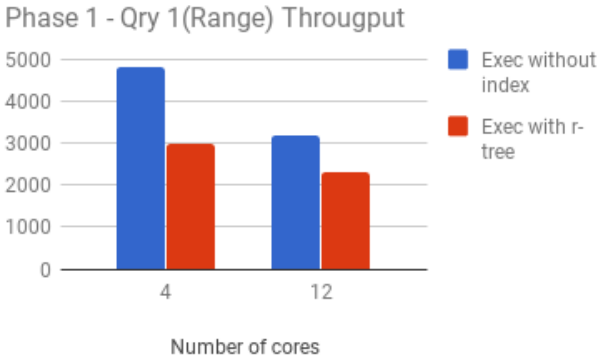
: Execution Time vs No. of Cores



: Peak Memory vs No. of Cores



: Throughput vs No. of Cores

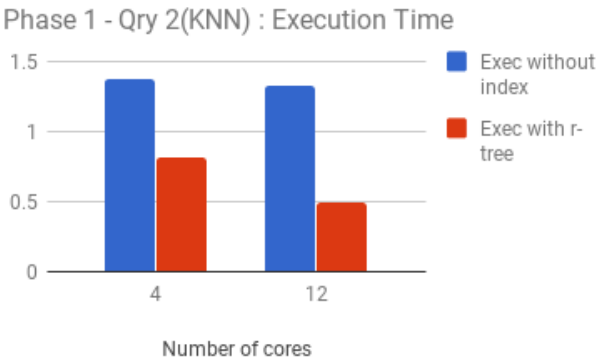


: Avg Execution Time vs Iterations

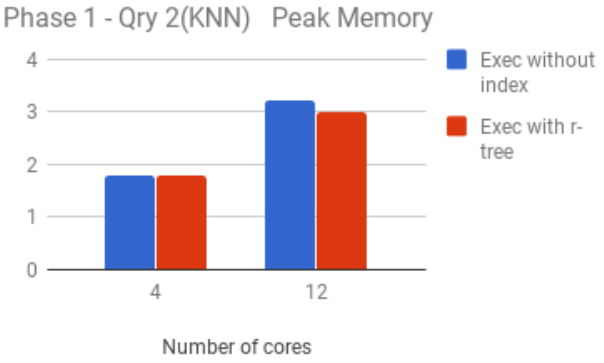
We observed that the execution time were much lower for indexed range query compared to non-indexed one. This was also observed for Throughput, while the Peak memory did not have a significant effect due to indexing. The average execution time for each iterations were also lower for indexed query. The increase in cores reduced the execution time in general. The reason could be due to higher throughput at per-process level with increase in number of cores.

4.2.2 Spatial KNN Query

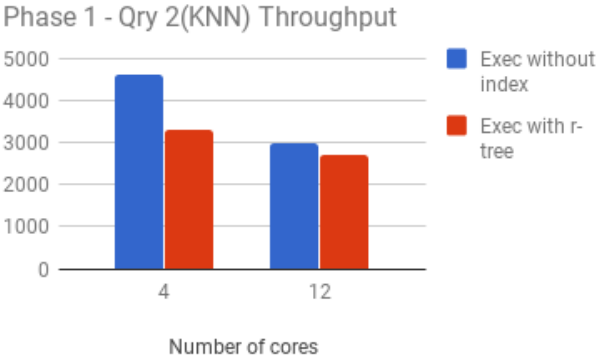
: Execution Time vs No. of Cores



: **Peak Memory vs No. of Cores**



: **Throughput vs No. of Cores**



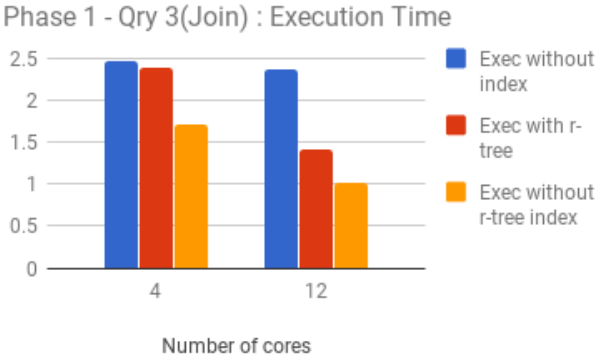
: **Avg Execution Time vs Iterations**

We observed that the execution time were much lower for indexed KNN query compared to non-indexed one. This was also observed for Throughput in case of 4 cores, while the Peak memory did not have a significant effect due to indexing. The average execution time for each iterations were also lower for indexed query. The increase in cores reduced the execution time in general, but increased the peak memory consumption.

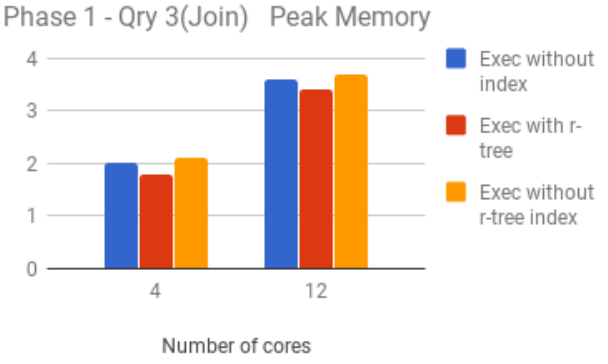
4.2.3 Spatial Join Query

Here, we perfomed our experiments for three types of join queries:Join Query with non-indexed equal grid without R-Tree index,Join Query with non-indexed equal grid with R-Tree index and Join Query with R-Tree grid without R-Tree index.

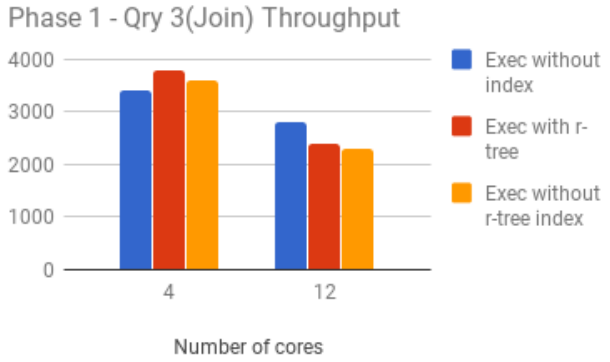
: **Execution Time vs No. of Cores**



: **Peak Memory vs No. of Cores**



: Throughput vs No. of Cores



: Avg Execution Time vs Iterations

We observed that the execution time were much lower for Join query with indexed grid without R-Tree index query. Throughput was comparable for all the join queries, while the Peak memory increased significantly after increasing the cores. The average execution time for each iterations were also comparable for all the join queries.

5. PROJECT PHASE 2

In Project Phase 2, our aim was to run four Spatial queries using two user-defined functions, that were **ST Contains** and **ST Within**.For this phase we used SparkSQL to define our functions and run our queries. We used Spark SQL DataFrames to run queries on the provided csv data files. Following are the algorithms for the two functions defined:

Algorithm: ST Contains

Input: query rectangle string,query point string
Output: True or False whether point lies in rectangle

```
1. Determine minX, maxX,minY and maxY
   from the rectange points
2. Compare query point 1st coordinate
   with minX,maxX and 2nd coordinate with maxY,MinY
3. If the point lies within the above
4. limits,return true else return false
```

Algorithm: ST Within

Input: query point1 string, query point2 string, query distance
Output: True or False whether point1 lies within given distance from point2

```
1. Calculate euclidean distance between the two
   points
2. If the obtained Euclidean distance is within
   the input query distance, then return true
   else return false
```

Using the above two functions we performed the following four queries:

- **Range Query:**Determines set of points that lie within a given rectange
- **Range Join Query:**Outputs (point,rectangle) pairs from given datasets such that the point lies in the paired rectangle
- **Distance Query:**Determines all points that lie within a given distance from a given point location.
- **Distance Join Query:**Outputs pairs of points belonging to two datasets which lie within a given distance from each other.

5.1 Experimental Setup

Datasets: Following are the datasets that were used for our experiments."arealm.csv" was used to create the Point DataFrame and "zcta510.csv" was used to create the Rectangle DataFrame. The smaller datasets were used to measure performance in case of data load change.

Table 3: Datasets

Dataset	Size	Description
arealm.csv	2.7 MB	point dataset
arealm10000.csv	212 KB	smaller point dataset
zcta510.csv	1.4 MB	rectangle dataset
zcta10000.csv	420 KB	smaller rectangle dataset

Cluster Settings: For Phase 2, we used the ThothLab cluster machines to setup our multi-node configurations. All the nodes shared the same configurations.

Table 4: Cluster Configuration

Machine	RAM	Disk Space	OS
Master	32 GB	250 GB	Ubuntu 14.04 LTS
Worker 1	32 GB	250 GB	Ubuntu 14.04 LTS
Worker 2	32 GB	250 GB	Ubuntu 14.04 LTS

Evaluation Metrics

We are evaluating our experiments based on the performance by varying Spark cores and varying datasets to measure execution time, throughput, memory usage and effect of varying iterations on the average execution time.

5.2 Experimental Results

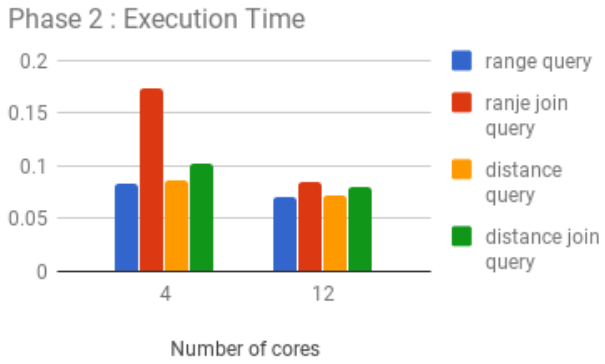
To calculate the execution time, we ran the four queries individually through Spark submit. We observed the total time taken for spark workers to run each query by observing the reading on the Spark master page. And we again

observed our readings by running the query after changing the cores from 4 to 12.
Similarly, for measuring the Peak Memory, we observed the ratings on the Ganglia graph for mem_report for cores 4 and 12. For Throughput, we observed the Ganglia readings for the proc_run graph and for Iterations, we ran the code three times by running the code iteratively for 10, 100 and 1000 times.

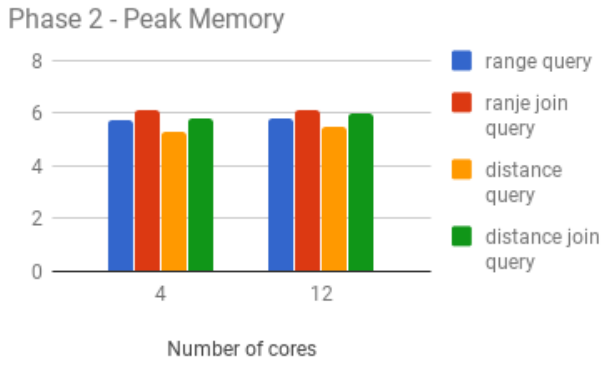
Finally, we combined the ratings of all four queries in individual praphs to compare their perfomance with each other against the change in cores.

Following are our observations as per our varied experiments run:

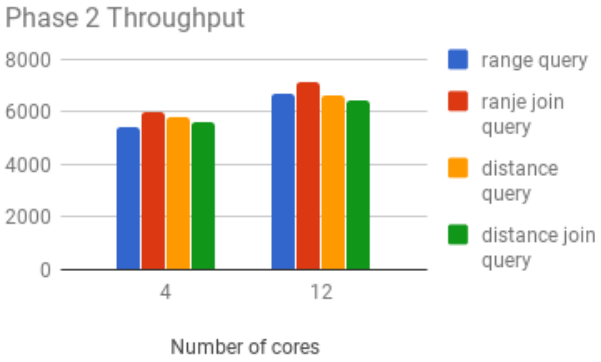
: Execution Time vs No. of Cores



: Peak Memory vs No. of Cores



: Throughput vs No. of Cores

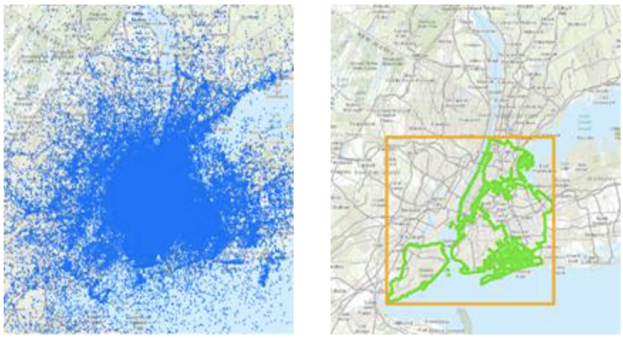


: Avg Execution Time vs Iterations

We observed that the Range Join Query takes maximum execution time compared to the other queries. It is expected as we are joining over range without indexing. We also observed that peak memory was almost same for all the queries and did not have any significant effect on core increase, while the throughput increased slightly on core increase. Use of Dataframes could explain this phenomena as Dataframes are immutable and give us better performance then Spark RDD for large datasets. Unlike a RDD, information is sorted out into named sections, similar to a table in a relational database. Intended to make expansive informational indexes preparing significantly less demanding, Range Join Query also had the maximum throughput, i.e. more jobs were executed for this query compared to others. Also, the average execution time for each iteration was almost static for range query, which it varies comparably for the other three queries.

6. PROJECT PHASE 3

In project Phase 3, we were required to perform spatial hotspot analysis on the provided datasets. We were provided with New York City yellow cab trip records dataset to perform spatio-temporal analysis for determining the hot spots in New York City.A hot spot is an area with maximum amount of pickup and dropoff, that is, the most active area. Our main goal was to calculate the Getis-ord statistics for the given dataset. These statistical analysis can help determine the need of more or less cabs in a particular area, thus increasing the revenue for providers as well as increasing convenience for the consumers, i.e. reducing the wait-time for a passenger, reducing the inactive time for a cab, etc.



: Hot Spot Analysis

Our problem statement for this phase consisted of two subtasks:

Hotzone Analysis: Objective of this task was to calculate the hotness of a rectangle. Given a point dataset and rectangle dataset, degree of hotness of a rectangle is measured by the total number of points it contains. So, larger the no. of points, hotter the rectangle is.
For determining whether a point lies in a rectangle or not, we used the ST Contains function as defined in Phase2. We submit a Spark SQL query to retrieve all (point,rectangle) pairs for which the point lies in the rectangle.
Next, we run another Spark SQL query which aggregates points found for each rectangle and returns the count of points found in a rectangle sorted in ascending order. Hence, by comparing the total no. of points present in each rectangle we measure its hotness.

Algorithm: Hotzone Analysis

Input: query rectangle string,query point string
Output: True or False whether point lies in rectangle

```
1. Range query with (Rectangle, Point)
   using ST_Contains as described in Section 5
2. Find hotness by using a count operation
   over the Join aggregating over the rectangles.
3. Return Rectangle tuples found
   in step 2 order by the rectangle string.
```

Hotcell Analysis: Objective of this task was to provide a z-score for a given location data. Firstly, we divide each area based on the NYC taxi dataset into blocks. Then we further divide each block into cell units. Now, based on position of the input location in the area, we select the neighboring cells for our evaluation. For Example, if the point lies in a corner block, we will select 7 neighboring cells, if the point is in a center block, we will select 20 neighboring cells, etc. After selecting the cells, we calculate the z-score for each cell, which is a measure of how busy that cell is. Finally, the highest z-score value is the winner for that query location and we return the value as our result.

Algorithm: Hotcell Analysis

Input: query point string
Output: z-score value

```
1. Select the point in data frame
   belonging to the given Hypercube range
2. Count the observation sequence x^t from the
   selection in point 1
3. Calculate Mean,Standard deviation on these
   observation sequences.
4. Based on the location, select the no. of
   neighboring cells
5. Calculate the weighted sum of neighbors
   of each cell and place it in
7. Calculate the z-score for each cell using
   the values of data frame from
   previous sequences.
8. Order as per  the z-score descending and
   return the cells.
```

6.1 Experimental Setup

Datasets: Following are the datasets that were used for our experiments.We used monthly NYC taxi datasets from 2009-12 with information such as pick-up locations, drop-off locations, etc. from data link:
https://datasyslab.s3.amazonaws.com/index.html?prefix=nyctaxitrip

Table 5: Datasets

Dataset	Size	Description
yellow_tripdata_2009_01_12point.csv	26.6	Monthly data '09
yellow_tripdata_2010_01_12point.csv	28.8G	Monthly data '10
yellow_tripdata_2011_01_12point.csv	30.2G	Monthly data '11
yellow_tripdata_2012_01_12point.csv	30.0G	Monthly data '12

Cluster Settings: Following were the configurations being used for our multi-node hadoop cluster settings. All of the clusters were running the same Ubuntu version. Master node cluster had a slightly higher disk space compared to the rest.But the computation power across all the nodes were equal.

Table 6: Cluster Configuration

Machine	RAM	Disk Space	OS
Master	32 GB	250 GB	Ubuntu 14.04 LTS
Worker 1	32 GB	250 GB	Ubuntu 14.04 LTS
Worker 2	32 GB	250 GB	Ubuntu 14.04 LTS

Evaluation Metrics
We are evaluating our experiments based on the performance by varying Spark cores to measure execution time, throughput, memory usage and effect of varying iterations on the average execution time.

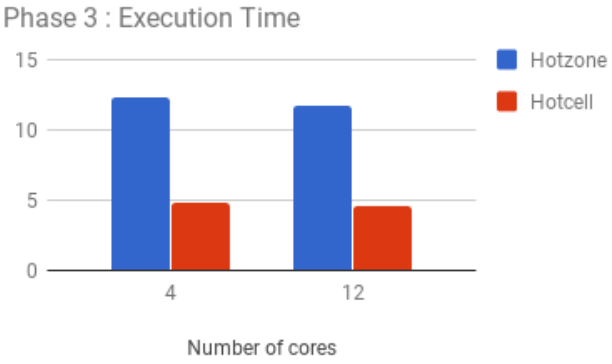
6.2 Experimental Results

To calculate the execution time, we ran the two queries individually through Spark submit. We observed the total time taken for spark workers to run each query by observing the reading on the Spark master page. And we again observed our readingby running the query after changing the cores from 4 to 12.
Similarly, for measuring the Peak Memory, we observed the ratings on the Ganglia graph for mem_report for cores 4 and 12. For Throughput, we observed the Ganglia readings for the proc_run graph and for Iterations, we ran the code three

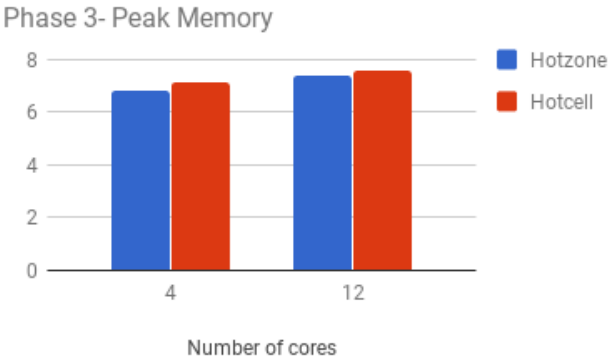
times by running the code iteratively for 10, 100 and 1000 times.

Following are our observations as per our varied experiments run:

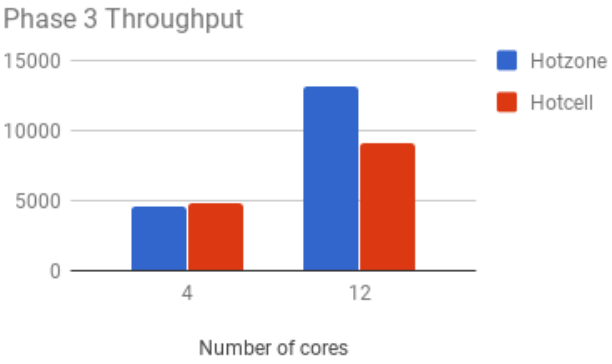
: Execution Time vs No. of Cores



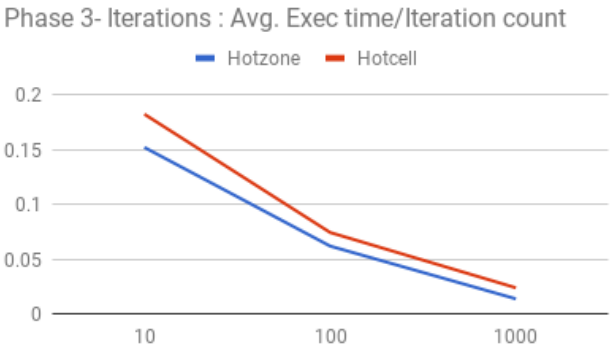
: Peak Memory vs No. of Cores



: Throughput vs No. of Cores



: Avg Execution Time vs Iterations



We observed that the Hotzone analysis takes more execution time than Hotcell Analysis. We also observed that peak memory was almost same for both the queries and did not have any significant effect on core increase, while the throughput increased on core increase. Also, the average execution time for each iterations were comparable for both Hotcell and Hotzone analysis. The reason for such a result is the use of persist function of Spark in Hotcell which was not used in hot-zones.Persist saves the Dataset using the default storage level and returns it. Internally,it hangs on to requests on CacheManager to store the request, which is accessible through SharedState.That is why we see a peak in memory for hotcell greater and lesser execution time.

7. CONCLUSIONS

Through implementing this project we now have a greater understanding of how the Distributed Database Systems can operate on huge volume of data. We learned about how GeoSpark API works and how Spark programmer can eaily leverage the API for a variety of use cases. We also got a hands-on experience on both Apache Hadoop and Apache Spark and how we can operate Map-Reduce tasks on it. In this project we used the GeoSpark API for Hot Spot detection. The task mainly focused on applying spatial statistics to spatial temporal data in order to identify significant hot spots.

8. ACKNOWLEDGMENTS

We would like to express our special thanks of gratitude to Prof. Mahamed Sarwat for giving us the golden opportunity to do this project that is highly relevant in the industry today. His support and motivation were essential for carrying out this project. We are also really thankful to Jia Yu for his lucid explanation on GeoSpark library and his assistance on our doubts when needed. Finally, we thank our teams members, Aditya Mishra, Nikita Shankar, Pratik Mishra and Sarvarth Bhatnagar without whom the project would never have sucessfully completed.

9. REFERENCES

[1] *"Apache Hadoop"*. http://pingax.com/install-apache-hadoop-ubuntu-cluster-setup/.

[2] *'Apache Hadoop FileSystem'*. https://hadoop.apache.org/docs/current/api/org/apache/hadoop/fs/FileSystem.html.

[3] *"Apache Spark"*. http://freecontent.manning.com/wp-content/uploads/

how-to-start-developing-spark-applications-in-eclipse.pdf.

[4] *"GIS CUP 2016"*. http://sigspatial2016.sigspatial.org/giscup2016/problem.

[5] *'Setting Up SSH'*. https://confluence.atlassian.com/bitbucket/set-up-an-ssh-key-728138079.html.

[6] 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] [6] [1] [2] [5] [3]