Hot Spots Identification using Geo Spatial Data

Deepak Soundararajan
Dsounda2@asu.edu

Narendra Kumar Sampath Kumar
nsampat1@asu.edu

Swati Kanchan
skanchan@asu.edu

ABSTRACT

The aim of the project is to identify statistically significant hotspots in the collection of New York City Yellow Cab taxi datasets of January 2015. The Hotspots were identified based on the Getis-Ord statistics. To obtain this, we use the Hadoop File system and the spark cluster consisting of a master and two worker nodes. We setup the Hadoop and spark with multiple live nodes and execute the tasks parallelly to have a computational advantage and CPU utilization being shared among the connected nodes. We also give a detailed analysis on how indexing affects the Memory usage, CPU Utilization, and Execution time and how a naïve Cartesian Join algorithm behaves under such scenario. The results obtained are compared against different indexing methodologies. Also, the specific technologies used in projects are beneficial in the scenarios where large amount of data needs to be processed and analyzed.

Keywords

Hadoop, Spark, Hotspots, Getis-Ord, Distributed Database Systems

1. INTRODUCTION

The amount of temporal-geospatial data being generated is increasing every day. The rapid growth of geospatial data and its pervasive uses in Distributed data processing and parallel computing highlights the requirement of modern storage and processing mechanism. A spatial referencing system consists of location co-ordinates of geographic space. It is hard to analyze high volumes of geospatial data which is beyond the storage capacity with traditional data mining techniques. Thus, Big Data handling tools are required to apply advanced Data processing and storage mechanism with fault tolerance. Big Data here refers to the High volume of data that are continuously generated over a time that cannot be stored and managed by traditional approaches. Hadoop deals with big data and it is an open source framework. There are two main components in Hadoop namely, HDFS (Hadoop Distributed File System). It is the ability of a system to continue normal operation against hardware or software failures and the other one is the Map reduce technique which has Map and Reduce part for processing and generating big data sets with a parallel distributed algorithm on a cluster.

We are given with the dataset containing the records of the Yellow Cab taxi trips in New York City of January 2015. The key elements in the dataset is alone taken into consideration. The columns such as pickup date, Latitude, Longitude are the three important Attributes forming the space-time cube in 3 dimension [1]. The latitude and longitude for each cell are of length 0.01 and the day is taken as the third dimension in the cell. To perform this high computation, we use Spark to have a cluster of nodes and Hadoop for Distributed Storage and processing.

The contents henceforth will contain the setup of Hadoop & Spark, Analysis of queries with and without indexing techniques, Map-Reduce technique to identify the statistically significant hotspots.

2. CONFIGURATION

2.1. SSH

We have used Hadoop 2.6.0 for this project. It requires SSH access to manage the nodes which includes the remote machines connected and the local nodes. We have created a new user 'hduser' for this setup. The prerequisite would be to install SSH in all the nodes and configured to allow SSH public key authentication.

Following are the steps to generate an SSH key for hduser.

1. su – hduser
2. ssh-keygen -t rsa -P ""

It will generate the public/private rsa key pair. The key will be saved to the default location /home/hduser/.ssh/id_rsa

3. Now push the public key to the authorized keys in order to connect the local machine with the hduser using this command. cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
4. ssh to the localhost.

The above steps should be carried on all the nodes forming the cluster.

2.2. SSH Acsess

Once the SSH to the localhost been established, the next step will be connecting all the machines over the network. We use IP address and update the hosts file

Following are the steps [3]:

1. Go to /etc/hosts/
2. Add the IP address of the master and worker machine in the following format,

    192.168.0.143 master
    192.168.0.123 worker1
    192.168.0.133 worker2

3. The hduser on the master must be able to connect to its own account using ssh master command
4. Push the key to the worker nodes using

    ssh-copy-id -i $HOME/.ssh/id_rsa.pub hduser@worker1

5. This command will prompt for password for the first time, once it is entered, the machine will be able to access worker1. Similarly, for worker2 also.

2.3. Hadoop Setup

The prerequisite would be to install Hadoop. The framework for the entire master slave setup is as shown in figure 1. The master node will run the "master" daemons as well the "worker" daemons: DataNode for the HDFS layer, TaskTracer for MapReduce processing layer. "master" daemons are responsible for co-ordination and management of "worker" daemons while the workers will help in data storage and data processing.

Following are the configuration steps for master and worker nodes. [4]

1. Go to conf/masters and add the "master" to it

2. Go to conf/slaves and add all the master and worker nodes name to it. It should look like

master

worker1

worker2

3. Go to conf/hdfs-site.xml on all the nodes and add the following content

<property>

<name>fs.default.name</name>

<value>hdfs://master:54310</value>

</property>

4. This specifies the Namenode of HDFS master node (the host) and the port.

5. Go to conf/mapred-site.xml and add the following content

<property>

<name>mapred.job.tracker</name>

<value>master:54311</value>

</property>

This specifies the Job tracker host and port

6. We change the dfs.replication parameter in conf/hdfs-site.xml which specifies the default block replication in HDFS. It defines the number of replications to be made to the data before it is available for the processing

<property>

<name>dfs.replication</name>

<value>2</value>

</property>

7. We must format Hadoop's distributed filesystem (HDFS) via the namenode before starting our multi node cluster. Go to the Hadoop/bin folder and execute the following command in the terminal

Hadoop namenode -format

Start the multi node cluster by running start-dfs.sh in Hadoop/sbin folder

The worker nodes can their log to examine the success or failure of the node.

Check the status of the Hadoop cluster by entering "localhost:50070" in the web browser.

2.4. Spark setup

The prerequisite for this setup would be to download the prebuild spark package for Hadoop 2.6.0 version. Once the library is made executable, the following steps is followed to launch the spark.

1. Add the following to the Spark/conf/spark-env.sh file.
SPARK_MASTER_IP = 192.168.0.143
2. Run the following command in terminal from the master node to start the spark. Go to sbin folder in spark, run
./start-master.sh
3. The status of the spark master can be checked by entering "localhost:8080" in the browser".
4. The worker nodes need to be registered to spark master. The spark runs on 192.168.0.143:7077 on master. Type the following command in every worker node.
.spark-class org.apache.spark.deploy.worker.Worker
spark://192.168.0.143:7077
5. If you are able to see the worker node's IP in the localhost:8080, then it has been successfully registered.

3. IMPLEMENTATION

Project is divided into three phases. The following are the requirements and implementation of different phases.

3.1. Phase 1 – Spatial Queries Using GeoSpark

The first phase involves in creating SpatialRDD objects and performing various spatial queries with the given dataset over the cluster. This task helps to gain knowledge on how GeoSpark library works and how we can leverage the API to solve our problem. Also, this task helped in learning about indexing and how indexing can help in reducing the time complexity and CPU utilization. Following are the queries executed.

3.1.1. Spatial Range Query

```
val pointRDD = new PointRDD(sc,
"hdfs://master:54310/tmp/arealm.csv", 0, FileDataSplitter.CSV,
false);

val queryWindow = new Envelope (35.08,32.99,-113.79,-
109.73);

val result = RangeQuery.SpatialRangeQuery(pointRDD,
queryWindow, 0,false).count();
```

3.1.2. Spatial Range Query With R-Tree Index

```
val pointRDD = new PointRDD(sc,
"hdfs://master:54310/tmp/arealm.csv", 0, FileDataSplitter.CSV,
false);

val queryWindow = new Envelope (35.08,32.99,-113.79,-
109.73);

pointRDD.buildIndex(IndexType.RTREE,false);

val result = RangeQuery.SpatialRangeQuery(pointRDD,
queryWindow, 0,false).count();
```

3.1.3. Spatial K-Nearest Neighbor Query

```
val fact=new GeometryFactory();

val queryPoint=fact.createPoint(new Coordinate(35.08, -113.79));

val objectRDD = new PointRDD(sc,
"hdfs://master:54310/tmp/arealm.csv", 0, FileDataSplitter.CSV,
false);

val resultSize = KNNQuery.SpatialKnnQuery(objectRDD,
queryPoint, 5,false);
```

3.1.4. Spatial KNN Query With R-Tree Index

```
val fact=new GeometryFactory();

val queryPoint=fact.createPoint(new Coordinate(35.08, -113.79));

val objectRDD = new PointRDD(sc,
"hdfs://master:54310/tmp/arealm.csv", 0, FileDataSplitter.CSV,
false);

objectRDD.buildIndex(IndexType.RTREE,false);

val resultSize = KNNQuery.SpatialKnnQuery(objectRDD,
queryPoint, 5,true);
```

3.1.5. Spatial Join Query Equal Grid

```
val fact=new GeometryFactory();

val objectRDD = new PointRDD(sc,
"hdfs://master:54310/tmp/arealm.csv", 0, FileDataSplitter.CSV,
false);

val rectangleRDD = new RectangleRDD(sc,
"hdfs://master:54310/tmp/zcta510.csv", 0, FileDataSplitter.CSV,
false);

objectRDD.spatialPartitioning(GridType.EQUALGRID);

rectangleRDD.spatialPartitioning(objectRDD.grids);

val resultSize =
JoinQuery.SpatialJoinQuery(objectRDD,rectangleRDD,false).cou
nt();
```

3.1.6. Spatial KNN Query with Equal Grid and R-Tree Index

```
val objectRDD = new PointRDD(sc,
"hdfs://master:54310/tmp/arealm.csv", 0, FileDataSplitter.CSV,
false);

val rectangleRDD = new RectangleRDD(sc,
"hdfs://master:54310/tmp/zcta510.csv", 0, FileDataSplitter.CSV,
false);

objectRDD.spatialPartitioning(GridType.EQUALGRID);

objectRDD.buildIndex(IndexType.RTREE,true);

rectangleRDD.spatialPartitioning(objectRDD.grids);

val resultSize =
JoinQuery.SpatialJoinQuery(objectRDD,rectangleRDD,true).coun
t();
```

3.1.7. Spatial KNN Query with R-Tree Grid and R-Tree Index

```
val objectRDD = new PointRDD(sc,
"hdfs://master:54310/tmp/arealm.csv", 0, FileDataSplitter.CSV,
false);

val rectangleRDD = new RectangleRDD(sc,
"hdfs://master:54310/tmp/zcta510.csv", 0, FileDataSplitter.CSV,
false);

objectRDD.spatialPartitioning(GridType.RTREE);

rectangleRDD.spatialPartitioning(objectRDD.grids);

val resultSize =
JoinQuery.SpatialJoinQuery(objectRDD,rectangleRDD,false).cou
nt();
```

3.2. Phase 2 – Spatial Join Query using the Simple Cartesian product Algorithm

In phase 2 of the project, we created a Java program along with the Geospark framework to perform Naïve spatial join query by using cartesian product algorithm. The performance of Naïve spatial join query with cartesian product was then compared with the inbuilt Geospark spatial join query functions with and without indexing. The performance were compared with metrics such as Average runtime, Average Memory consumption and average CPU utilization by using a monitoring tool, Ganglia.

The algorithm below was used to code the Naïve spatial join query using cartesian product.
(Source: https://gist.github.com/aniquetahir/acb2a781a55cf76a5d2a32d4f0a4d5d6)

1.Create a PointRDD objectRDD;
2.Create a RectangleRDD queryWindowRDD;
3.Collect rectangles from queryWindowRDD to one Java List L;
4. For each rectangle R in L
do RangeQuery.SpatialRangeQuery(objectRDD, queryEnvelope, 0, true);
End;
5.Collect all results; //"Collect" is a standard function under SparkContext.
6.Parallelize the results to generate a RDD in this format:

JavaPairRDD<Envelope, HashSet<Point>>. ;//"Parallelize" is a standard function under SparkContext.
7.Return the result RDD;

The data is first loaded into the Hadoop file system and spark uses the data in the hdfs to perform the spatial join query. The SpatialJoinQueryUsingCartesianProduct was then added into the Geospark library under the JoinQuery class.

The following section discusses the performance metrics and comparison of the performance metrics for the queries in phase 1 and the spatial join queries from phase 1 with the naïve spatial join query with cartesian product program.

3.2.1. Performance Comparisons

3.2.1.1. Comparing Spatial Range Query and Spatial Range Query with R-Tree Index

For this comparison, we ran the spatial range query and then spatial range query with r-tree indexing. As seen from the figures and the table below, indexing helps in performing efficient spatial range query by reducing the average runtime and CPU utilization in comparison to the spatial range query without indexing. The memory usage slightly increases in the case of spatial range query with r-tree indexing. The r-tree indexing groups nearby objects together and therefore makes the spatial range query more efficient [2].

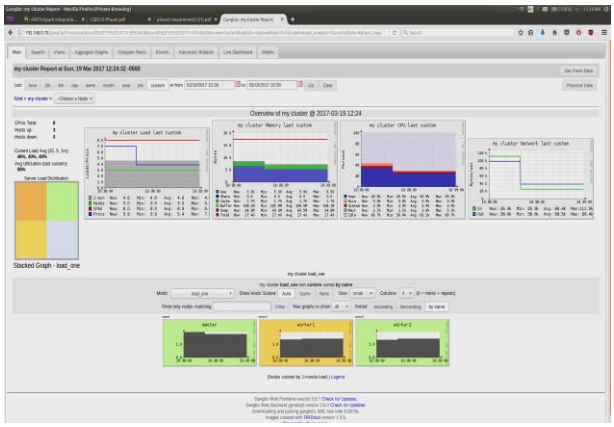

Figure 1 : Avg. CPU, Memory and Runtime for spatial range query

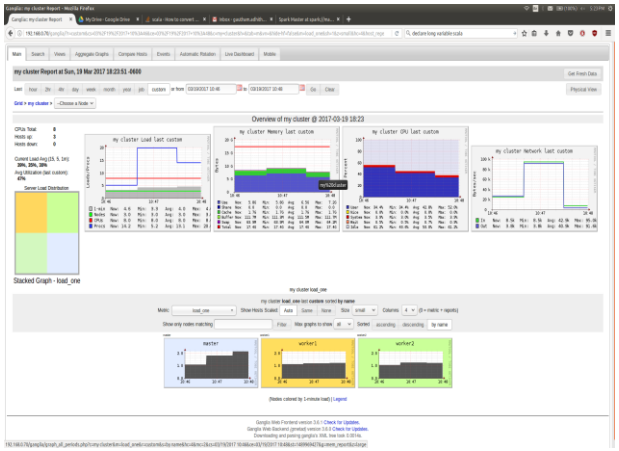

Figure 2 : Avg. CPU, Memory and Runtime for spatial range query with r-tree indexing

Table 1 : Comparison of performance metrics for spatial range query

	Spatial Range Query	**Spatial Range Query (Index)**
Avg. Runtime	5.1 seconds	4.7 seconds
Avg. CPU Usage	43.7%	30.9%
Avg. Memory Usage	5.7G	5.9G

3.2.1.2. Comparing Spatial KNN query and Spatial KNN query with R-Tree Indexing

For this comparison, we ran the spatial knn query to find 5 nearest neighbors of a input query point and then ran the spatial knn query with r-tree indexing to find the 5 nearest neighbors for the same input query point. With the comparison of the performance metrics of both the queries, we can argue that by using r-tree indexing , the CPU utilization and average runtime has decreased although the memory utilization has increased slightly in the case of spatial knn query. The r-tree indexing groups nearby objects together and therefore makes the spatial knn query more efficient [2].

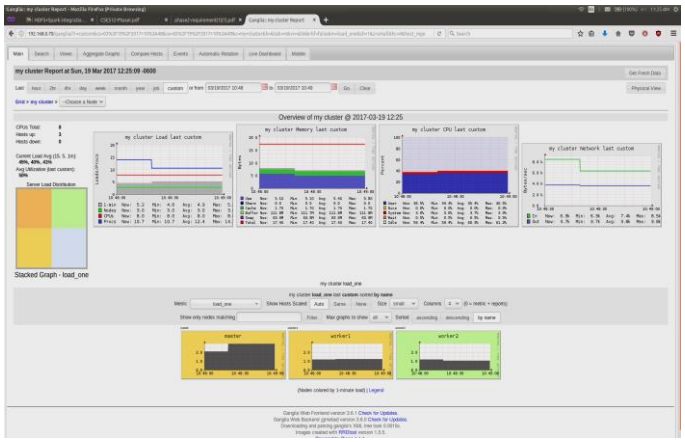

Figure 3 : Avg. CPU, Memory and Runtime for spatial K-NN query

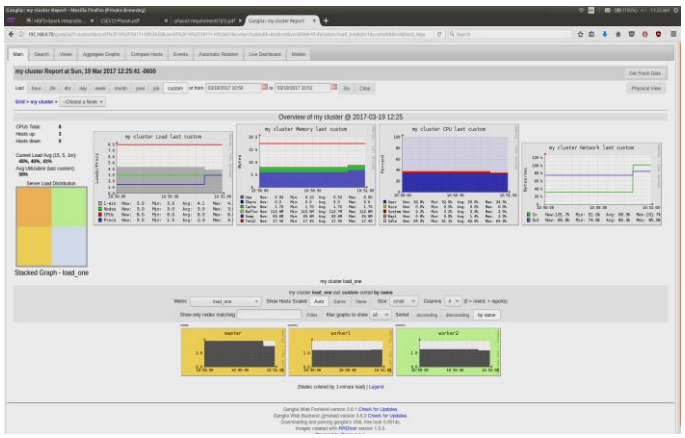

Figure 4 : Avg. CPU, Memory and Runtime for spatial K-NN query with r-tree indexing

Table 2 : Comparison of performance metrics for spatial K-NN query

	Spatial KNN Query	**Spatial KNN Query (Index)**
Avg. Runtime	5.9 seconds	5.1 seconds
Avg. CPU Usage	35.4%	33.6%
Avg. Memory Usage	5.1G	6.5G

3.2.1.3. Comparing Spatial Join Query with Equal Grid and Spatial Join with Equal Grid and R-Tree Indexing

For this comparison, we ran the spatial join query with equal grid on given set of rectangles envelopes & the given set of data points, and the spatial join query with equal grid and r-tree indexing on the same set of envelopes and data points. By comparing the performance metrics of both the queries, it can be argued that by using the r-tree indexing, the CPU utilization and the average runtime has reduced substantially, although the memory utilization due to r-tree indexing has increased slightly in comparison with the spatial join query and equal grid without r-tree indexing. The r-tree indexing groups nearby objects together and therefore makes the spatial join query more efficient [2].

Table 3 : Comparison of performance metrics for spatial join query with equal grid

	Spatial Join Query (Equal Grid)	**Spatial Join Query (Equal Grid with Index)**
Avg. Runtime	313 seconds	227 seconds
Avg. CPU Usage	36.8%	35.8%
Avg. Memory Usage	6.1G	7.3G

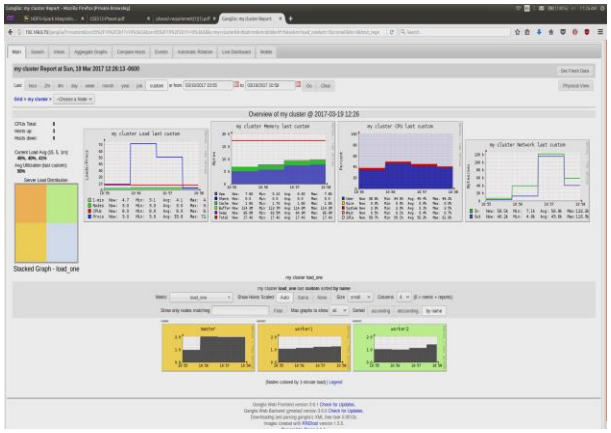

Figure 5 : Avg. CPU, Memory and Runtime for spatial join query with equal grid

Figure 6 : Avg. CPU, Memory and Runtime for spatial range query with equal grid and indexing

3.2.1.4. Comparison of Spatial Join Query with Equal Grid and Spatial Join Query with R-Tree Grid

For this comparison, we ran the spatial join query with equal grid on given set of rectangles envelopes & the given set of data points, and the spatial join query with r-tree grid on the same set of envelopes and data points. By comparing the performance metrics of both the queries, it can be argued that by using of r-tree grid, the CPU utilization and the average runtime has reduced substantially, although the memory utilization due to r-tree grid has increased slightly in comparison with the spatial join query and equal grid. The r-tree grids group nearby objects together and therefore makes the spatial join query to look up only few nearby grids for the points more efficient [2].

Table 4 : Comparison of performance metrics for spatial join query

	Spatial Join Query (Equal Grid)	**Spatial Join Query (R-Tree Grid with Index)**
Avg. Runtime	313 seconds	243 seconds
Avg. CPU Usage	36.8%	35.5%
Avg. Memory Usage	6.1G	7.2G

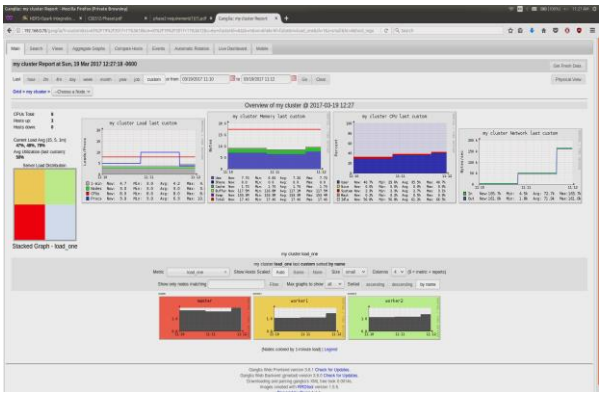

Figure 7 : Avg. CPU, Memory and Runtime for spatial join query with r-tree grid

3.2.1.5. Comparison of Naïve Spatial Join Query using Cartesian Product Algorithm and Spatial Join Query with Equal Grid, Spatial Join Query with Equal Grid and R-Tree Indexing and Spatial Join Query with R-Tree Grid

For this comparison, we ran the custom Java program that we created using the Naïve spatial join query using cartesian product algorithm discussed in the implementation section, to perform spatial join over a given set of rectangle envelopes and a set of data points. We then ran the spatial join query with r-tree grid, spatial join query with equal grid and spatial join query with equal grid and r-tree indexing over the same set of envelopes and data points. It was found that cartesian product algorithm is very poor to perform spatial join query since the runtime, memory usage and cpu utilization for the cartesian product algorithm was much higher when compared to the other spatial join query algorithms discussed above. The cartesian product algorithm find every possible combination of data points and the envelope and then performs joins if they can be joined, thereby taking more computation time and hence is not efficient.

Table 5 : Comparison of performance metrics for spatial join query

	Spatial Join Query	**Spatial Join Query (Index)**	**Spatial Join Query (Equal Grid)**	**Spatial Join Query (R-Tree Grid with Index)**
Avg. Runtime	313 seconds	227 seconds	243 seconds	6450 seconds
Avg. CPU Usage	36.8%	35.8%	35.5%	36%
Avg. Memory Usage	6.1G	7.3G	7.2G	8.2G

3.3 Phase 3 – Identifying Statistically Significant Hot Spots by Applying Spatial Statistics to Spatio-Temporal Data

In Phase 3 of our project, we have used Apache Spark to identify the statistically significant hot spots from the given data set. The input data was contained of New York Yellow Cab taxi trip records for the January 2015. We calculated the fifty most significant hot spot cells in space and time by using Getis-Ord G_i^* statistic. "When identifying statistically significant clusters (often termed Hot Spot Analysis), a very commonly used statistic is the Getis-Ord statistic. It provides a z-score and p-values that allow users to determine where features with either high or low values are clustered spatially." [1]. The formula to calculate the Getis-ord statistic is specified as

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

Where, $w_{i,j}$ is the spatial weight between cell i and j

x_j is the attribute value for cell j

And n is total number of cells present in data

$\bar{X}$ and S are calculated as followed:

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

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

We filtered the data based upon the following factors:

1. Latitude ranges from 40.5N to 40.9N
2. Longitude ranges from -74.25 to -73.70
3. To create the 3-D matrix, the unit size of each cell should be 0.01*0.01 (latitude and longitude respectively.
4. The Time-Stamp size should be 1 day (3rd dimension of the matrix).
5. Only pick up locations need to be considered.

3.3.1. Architecture

The architecture is depicted in Figure 8. The Java Spark driver initiates the worker processes. The worker processes get the data from HDFS and perform Map Reduce on the data to generate the results.

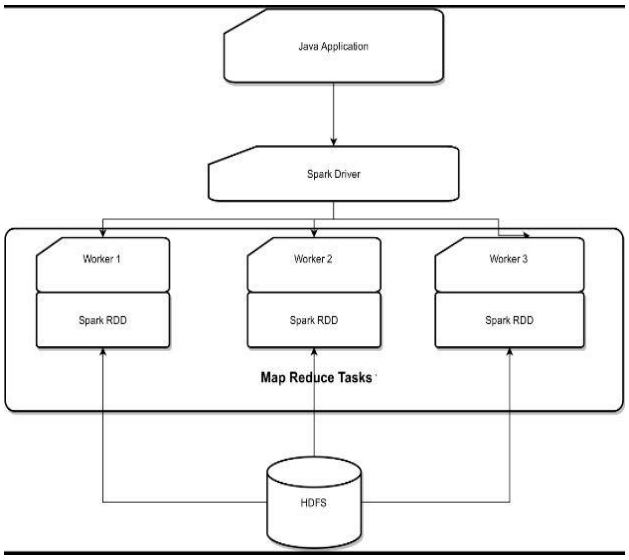

Figure 8. Architecture for Hotspot analysis

3.3.2 Solution Explanation

To identify the hot spots in the given data, we read the data from CSV file, filtered it per the requirement, and processed it. We created a 3-Dimensional matrix having latitude, longitude and time (specifically day of the month starting from 0 to 30) as its dimensions. We stored the number of rows in input data having the current dimensions as their values for latitude, longitude and date respectively, as the value of the matrix.

Once we processed the data, we computed the z-score using the Getis-Ord formula and stored the results in a priority queue. We sorted the priority queue in descending order and stored the top 50 results representing the top 50 hottest cells into the output file.

3.3.2.1 Node class to represent Result Structure

We created a node class to represent an object of following type:

Latitude, Longitude, Date, Score

3.3.2.2 Computing Neighbors

We assumed four different cases to compute the number of neighbors a cell has.

1. If a cell lies somewhere in the middle of the cube (not touching any border), it will have 27 neighbors.

2. If it is touching one border, it will have 18 neighbors.

3. If it is touching two borders, it will have 12 neighbors.

4. If the cell is in one of the corners of space-time cube, it will have only 8 neighbors.

3.3.2.3 Algorithm

Assumptions:

1. The weight of the neighboring cell = 1
   The weight of the non-neighboring cell = 0

Step 1: Define the following values as constants, to be used in algorithm while creating a 3-D matrix to represent the space-time cube.

Minimum Latitude = 40.50

Maximum Latitude = 40.90

Minimum Longitude = -74.25

Maximum Longitude = -73.70

Step 2: Create and calculate the following fields based on the values defined in step 1, to create a dimension matrix as:

Latitude Count = (Maximum Latitude – Minimum Latitude + 0.01) / 0.01

Longitude Count = (Maximum Longitude – Minimum Longitude + 0.01) / 0.01

Cell Count = Latitude Count * Longitude Count * 31

Step 3: Create a 3-Dimensional matrix with following dimension sizes:

1st dimension size = Latitude Count

2nd dimension size = Longitude Count

3rd dimension size = 31 (represents 31 days in January)

Step 4: Read the data from csv line by line, and get the values of latitude, longitude, and date for each row. Filter only those rows where the values of longitude and latitude fall between the range given in problem specification.

Step 5: Normalize the values of longitude, latitude, and date so that three values could be used as indices of 3-d matrix created in step 3.

Step 6: Once the normalization process is done, store the values in a map.

The keys of the map will be of the form: ‘Latitude, Longitude, Date’.

And values will be of the form: Number of entries having the same key.

Step 7: Then iterate through the map and store the values in dimension matrix using the following formula:

Dimension Matrix[Dimension1][Dimension2][Dimension3] = Map.getValue()

where Dimension1, Dimension2 and Dimension3 represents the latitude, longitude and date corresponding to the current Map Key respectively.

Step 8: Then calculate the Z-score for each cell of Dimension Matrix by using the formula given in problem description and store the values in a priority queue. The priority queue contains objects of type ‘Node’. Node is a user defined class, which contain the following 4 fields:

1. Latitude

2. Longitude

3. Date

4. Score.

Also provide an implementation of comparator to sort the priority queue values in descending order. So that the result can be sorted in the same order.

Step 9: Once we are done with calculation for all the cells, we write the first 50 values to a file, which will be our desired output. These first 50 cells will correspond to the hottest cells.

The results were written in following format:
cell_x, cell_y, time_step, zscore

4. RESULTS

The results are shown in Figure 9. The redness of the points corresponds to the hotness. The redder the spot is, the hotter it will be. The results were generated using the algorithm described in the previous section and the map was generated with the top 50 hottest points.

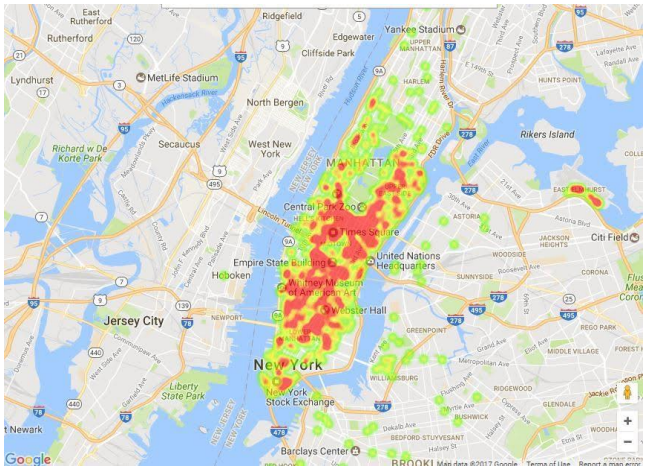

Figure 9. Result of Phase 3 in Map form

5. SKILLS LEARNED

While working on our project, we learnt how does the processing of large data sets in distributed and parallel networks take place. We worked on different technologies and frameworks and learnt the following skills:

Apache Spark, Hadoop, GeoSpark, MapReduce Framework

We will continue working on the above skills to expand our learnings.

6. CHALLENGES FACED

The most difficult part of our project has been the initial configuration part to setup the workers and masters into our systems. Once the setup was done, we faced a few other minor issues while working on different phases, as we had to code in Scala, using the RDD data structures, which were completely new concepts to us. We referred a few articles, documentation to get ourselves familiar with the technologies used in the project, and completed the entire project successfully.

7. ACKNOWLEDGEMENTS

We thank Dr. Sarwat for his guidance and support to help us learn the concepts and implement them successfully to ours project.

8. REFERENCES

[1] "ACM Sigspatial Cup 2016", http://sigspatial2016.sigspatial.org/giscup2016/problem

[2] R-tree. (2017, April 15). In Wikipedia, The Free Encyclopedia. Retrieved 23:47, April 30, 2017, from https://en.wikipedia.org/w/index.php?title=R-tree&oldid=775562765

[3] http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/

[4] http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/