Large-scale Geospatial Data Analysis using SparkSQL

Aditya Chayapathy
Arizona State University
achayapa@asu.edu

Kevin Thomas
Arizona State University
kthoma46@asu.edu

Kunwar Chowhan
Arizona State University
kchowhan@asu.edu

ABSTRACT

Large scale geospatial data analysis using the Spark SQL module of the Apache Spark cluster computing framework. New York City taxi data is analyzed to calculate the fifty most significant hot zones within the city to conform to the requirements of the ACM SIGSPATIAL Cup 2016[1].

KEYWORDS

Apache Hadoop, Apache Spark, SparkSQL, GeoSpark, Geospatial Analysis, Distributed and Parallel Database Systems

1 INTRODUCTION

Geospatial data consists of data that intrinsically contains locational information with attributes that vary depending on the use cases. This data is most often very dense i.e. containing a large number of data points which requires a great deal of computational and processing effort to gather a better understanding by converting this to high level. This is therefore a hard problem due to the scale of the data under consideration due to subjective considerations such as spatial joins or hull analysis. [2]. As an example, in spatial join operations [3], attributes are assigned based on locational information such as gathering insight from postcode information. With the advent of the Internet age, there has been an emergence of large scale distributed data processing frameworks that efficiently handle these workloads with characteristics such as fault tolerance and replication to ensure data integrity. Apache Hadoop and Spark are two such open source frameworks and this report describes the implementation of a system that leverages these frameworks in a distributed cloud computing cluster to analyze geospatial data with a dataset that contains taxi trip information from New York City (NYC) yellow cab data from the years 2009 to 2012. The purpose of this analysis is to calculate the hot zones for pickups in the given area of the city[4,5].

This system uses 3 Virtual Machine (VM) instances of the Google Compute Engine with the Apache Spark and Hadoop frameworks installed and spatial analysis is performed with the GeoSpark Application Programming Interface on Apache Spark on a sample data set.

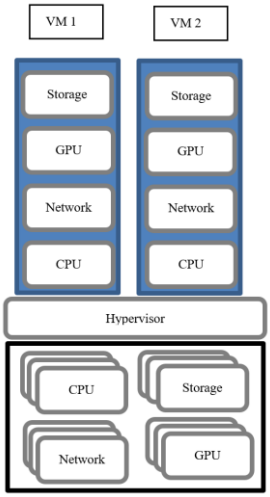

Figure 1: Virtual Machine abstraction in a typical Infrastructure as a Service (IaaS) setup

The querying of data with the SparkSQL module of Apache Spark is then performed on this system with queries on a dataset containing points and rectangles to ascertain which set of points are in the given set of rectangles.

Calculating the hot zones by the number of customer pickups at a location is defined in ACM SIGPATIAL Cup 2016 using the Getis – Ord G_i* statistic. The taxi trip data set is pre-processed into a space time cube of latitude by longitude by day in the month when the pickup took place. A cell in this space time cube consists of a range of locations per day and the Getis – Ord G_i* statistic considers the spatial proximity of each cell to the other by appropriately weighting these values.

2 SYSTEM ARCHITECTURE

2.1 Connectivity and Infrastructure

It is important that a system used in large scale data analysis can adapt to workloads of differing scale and computational intensity. The Google Compute Engine is chosen as the cloud infrastructure for this project as it has intuitive capabilities for disk imaging and snapshots; adaptive control and monitoring for VM instances; and is cost-effective. Three Ubuntu Linux VM instances are setup

with one master and two workers, with the */etc/hosts* file setup so that instances can connect to each other by the names "master", "worker-1" or "worker-2" instead of their internal IP addresses [6]. Another requirement of this system is that it must have password less SSH access between instances for Apache Hadoop and Spark. To connect to this system for configuration and monitoring: SSH remote access to instances and port forwarding of web interface ports was done using command line tools (CLI) provided in the Google Cloud Software Development Kit (SDK) [9]. Figure 2 describes the topology and connectivity for these VM instances.

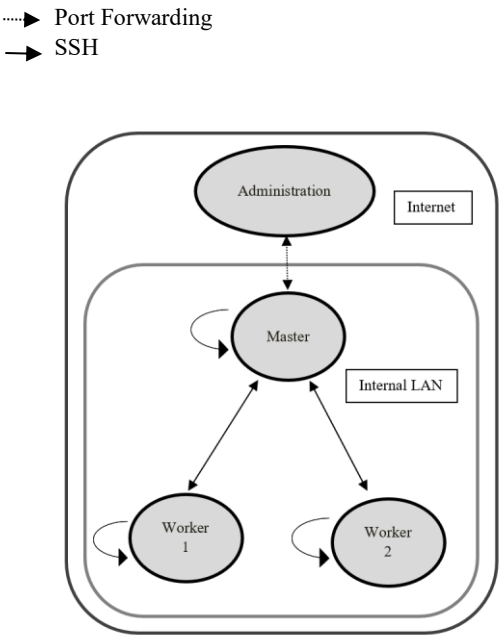

Figure 2: Google Compute Engine VM Networking

2.2 Frameworks and Modules

2.2.1 Apache Hadoop: is a framework that involve distributed data processing tools and the Hadoop Distributed File System (HDFS) is used here [7]. This file system ensures data integrity through replication and is set up with a replication factor of 3 (each block of data has 3 replicas). Data from the HDFS is managed in a Master – Worker format (Figure 3).

Figure 3: Google Compute Engine VM Instances

Arbitrating access on the HDFS is done by the NameNode and Secondary NameNodes, Resource Manager and Node Manager on the *master* which communicates with the Node Managers on the *workers* who serve data on the Data Nodes.

Once this system is set up, data in comma separated value format (CSV) is loaded onto the file system using the CLI on the *master.* A successful data load is smoke tested using the web interface (accessed over port forwarding) and common Linux CLI commands such as *tail* or *cat.*

2.2.2 Apache Spark: is an in-memory distributed computing framework that performs operations in the form of transformations and actions on a Resilient Distributed Dataset (RDD) abstraction. The Spark system represents a workload as a Directed Acyclic Graph (DAG) of these transformations and actions a key difference to distributed computing paradigms such as MapReduce [8]. This system also uses a Master – Worker architecture and the cluster used for Hadoop has Spark installed on the same VM (Figure 3). This system can read data from the HDFS with minimal configuration and referenced in the Application Programming Interface (API). Once the Spark environment is correctly set up (Figure 4) the *master* process is started and listens for *worker* processes to register. As a functional test of this system the GeoSpark API is used on the *Spark Interactive Shell.*

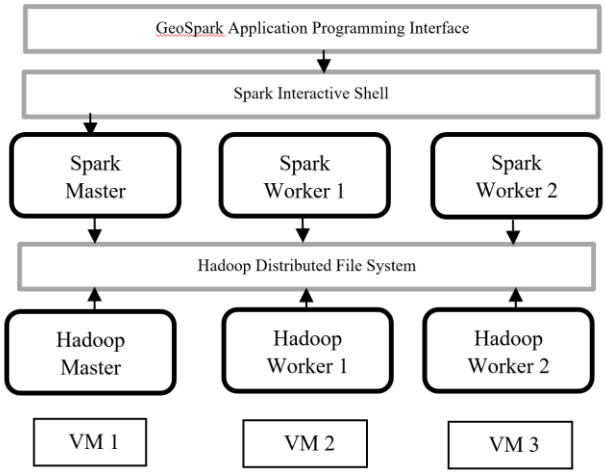

Figure 4: Functional description of GeoSpark running on Apache Hadoop and Spark in a 3 VM instance setup

2.3 GeoSpark

Before spatial queries are experimentally determined with the SparkSQL module alone, the GeoSpark API is chosen as an integration test of this software system because of its dependencies: SparkSQL, HDFS, Scala etc. GeoSpark also provides an intuitive means of executing spatial queries with additional optimizations such as indexing and spatial RDDs which are used to ascertain system performance. With a CSV input of

points given by (x, y) and *rectangles* given by (x1, y1) and (x2, y2), RDDs *point* and *rectangle* are created to query:

1. A window of size (x3, y3) (x4, y4) on *point*.
2. A window of size (x3, y3) (x4, y4) on *point* that is indexed with an R-Tree.
3. The 5 Nearest Neighbours in *point* of point (x5, y5).
4. The 5 Nearest Neighbours in *point* (that is indexed using an R-Tree) of point (x5, y5).
5. A Spatial Join for *point* and *rectangle* with spatial grid partitioning (equal grid).
6. A Spatial Join for *point* and *rectangle* with spatial grid partitioning (equal grid) and a R-Tree Index.
7. A Spatial Join for *point* and *rectangle* with spatial grid partitioning (R-Tree).

2.4 Setup Considerations

2.4.1 VM, Hadoop and Spark: The setup that is described does not use images with prebuilt versions of Hadoop and Spark. The base configuration for these frameworks was found to be challenging and requires a systematic approach to configure network settings, environment variables, Bash scripts and XML setting files. In addition, as VM instances on IaaS providers typically do not have a desktop user interface and distributed frameworks use web interface dashboards to allow an intuitive view of settings, port forwarding using the Google Cloud SDK of these web ports is selected as the preferred method of administration.

2.4.2 Java ARchive (JAR) testing: The Spark framework provides many methods of running workloads and as an initial test, the Spark Interactive Shell (which opens with the Scala programming language by default) was chosen which allows the loading of an API as a single JAR. The GeoSpark API is loaded this way and relevant libraries are imported that allow the loading of data, and RDD spatial operations.

2.4.3 System Performance and Cost Effectiveness: It is also important to consider that Google Compute Engine VM Instances vary in cost depending on virtual hardware resources allocated by the hypervisor and the activity of these resources. However, IaaS providers allow the fine-tuned control of the resources used by VM instances in this system which means that time intensive set up operations can be done with minimal resources and increased as required. As an example, loading data on HDFS and testing dependencies on Spark can be done with $1/4^{th}$ of the resources necessary to perform spatial queries with the GeoSpark API.

The following are the set of steps performed to obtain password-less SSH between the 3 machines.

1. Generate public and private keys for each machine using the command “ssh-keygen”.
2. Copy the public keys of all the workers onto the master using the command “ssh-copy-id user@machine“.
3. Also, copy the public key of each machine onto itself using “ssh-copy-id user@machine”.
4. Test that you can login to the workers from the master without password using the command “ssh user@machine”.

The following are the set of steps performed to setup the Hadoop cluster on the 3 machines:

1. Update the following files with the changes as described on Table 1 all the machines:

Table 1: List of file changes to be made

File	Property	Value
hadoop-env.sh	JAVA_HOME	/usr/lib/jvm/YOUR-JAVA-FOLDER
core-site.xml	fs.default.name	hdfs://master:54310
mapred-site.xml	mapred.job.tracker	master:54311
hdfs-site.xml	dfs.replication	3

2. On the master machine, add the following lines in HadoopFolder/etc/Hadoop/slaves:
   master
   worker2
   worker3
3. Format the HDFS system using the command “hadoop namenode -format”
4. Start the process on the master machine using the following command “sbin/start-all.sh”.
5. Check the status of the hadoop cluster by entering “localhost:50070” on the web browser.

The following are the set of steps performed to setup the Spark cluster on the 3 machines:

1. On the master machine, add the following line in “SparkFolder/conf/spark-env.sh”:
   SPARK_MASTER_IP=master
2. Start the spark master using the command “sbin/start-master.sh”.
3. Check the status of Spark master by entering “localhost:8080” on the web browser.
4. On the worker machines, run the following line in SparkFolder/bin/ folder to register and run the worker in Spark master.
   ./spark-class org.apache.spark.deploy.worker.Worker spark://master:7077

3 GEOSPATIAL DATA ANALYSIS

3.1 SparkSQL

The SparkSQL module of the Apache Spark framework allows SQL-like queries to be executed by the Spark sub system.

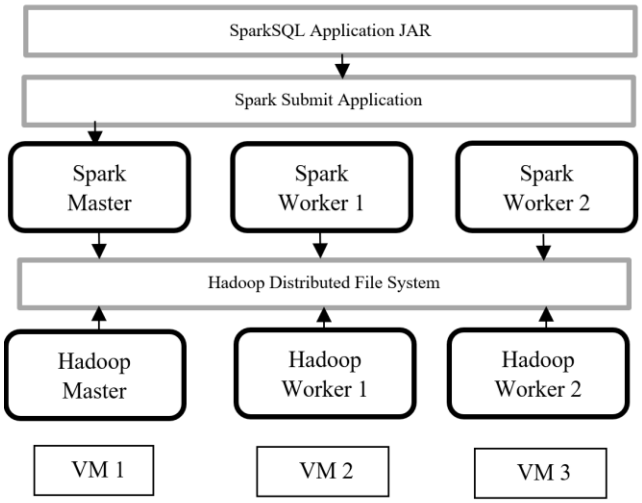

Figure 5: Functional description of experimental setup used for SparkSQL applications.

Spark performs extra optimizations using the information provided by SparkSQL on the data and the computation. It uses the same computing engine irrespective of which API or programming language is used to express the queries, thereby making it easier for developers to express queries in different ways. [12]

SparkSQL allows developers to define user-defined functions which work on the data in the SQL query itself. These functions need to be registered with SparkSQL before it is used using the udf.register() method. [12]

SparkSQL uses Datasets and Dataframe to provide its functionalities. A Dataset is a distributed collection of data. It combines the benefit of RDDs such as strong typing and usage of lamba functions and that of the Spark execution engine. A Dataframe is a Dataset organized into named columns. This is equivalent to a table in relational databases. In Scala, a Dataframe is a Dataset of Rows [12].

It is required to implement two functions in Scala which the SparkSQL could use in optimizing SQL queries called ST_Contains and ST_Within. The algorithms are as follows:

ST_Contains(point, rectangle):

1. Extract x and y coordinates of the point. Call it (x,y).
2. Extract the x and y coordinates of the diagonal points of the rectangle. Call is (x_1, y_1) and (x_2, y_2).
3. Find the minimum and maximum x-value from x_1 and x_2. Call it minX and maxX respectively.
4. Find the minimum and maximum y-value from y_1 and y_2. Call it minY and maxY respectively.
5. If minx <= x <= maxX and minY<= y <= maxY, then return TRUE else return FALSE.

ST_Within(point1, point2, distance):

1. Extract x and y coordinates of point1 and point2. Call it (x_1,y_1) and (x_2,y_2) respectively.
2. Calculate the Euclidean distance(d) between the two points using the formula:
$$d = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$$
3. If d <= distance, then return TRUE else return FALSE.

We are asked to apply these two user defined functions on four different spatial queries:

1. *Range query*: Given a query rectangle R and a set of points P, find all the points within R. ST_Contains is used for this query.

```
+--------------------+
|                 _c0|
+--------------------+
|-93.579565,33.205413|
|-93.417285,33.171084|
|-93.493952,33.194597|
|-93.436889,33.214568|
+--------------------+
```

Figure 6: Range Query Output

2. *Range join query*: Given a set of Rectangles R and a set of Points S, find all (Point, Rectangle) pairs such that the point is within the rectangle [10]. ST_Contains is used for this query.

```
+--------------------+--------------------+
|                 _c0|                 _c0|
+--------------------+--------------------+
|-93.63173,33.0183...|-93.579565,33.205413|
|-93.63173,33.0183...|-93.417285,33.171084|
|-93.63173,33.0183...|-93.493952,33.194597|
|-93.63173,33.0183...|-93.436889,33.214568|
|-93.595831,33.150...|-93.491216,33.347274|
|-93.595831,33.150...|-93.477292,33.273752|
|-93.595831,33.150...|-93.420703,33.466034|
|-93.595831,33.150...|-93.571107,33.247214|
|-93.595831,33.150...|-93.579235,33.387148|
|-93.595831,33.150...|-93.442892,33.370218|
|-93.595831,33.150...|-93.579565,33.205413|
|-93.595831,33.150...|-93.573212,33.375124|
|-93.595831,33.150...|-93.417285,33.171084|
|-93.595831,33.150...|-93.577585,33.357227|
|-93.595831,33.150...|-93.441874,33.352392|
|-93.595831,33.150...|-93.493952,33.194597|
|-93.595831,33.150...|-93.436889,33.214568|
|-93.595831,33.150...|-93.437081,33.360932|
|-93.442326,33.248...|-93.242238,33.288578|
|-93.442326,33.248...|-93.224276,33.320149|
+--------------------+--------------------+
only showing top 20 rows
```

Figure 7: Range Join Query Output

3. Distance query: Given a point location P and distance D in km, find all points that lie within a distance D from P. ST_Within is used for this query.

```
+--------------------+
|                 _c0|
+--------------------+
|-88.331492,32.324142|
|-88.175933,32.360763|
|-88.388954,32.357073|
| -88.221102,32.35078|
|-88.323995,32.950671|
|-88.231077,32.700812|
|-88.349276,32.548266|
|-88.304259,32.488903|
| -88.182481,32.59966|
|-87.534883,31.934442|
| -87.49702,31.894541|
|-88.153618,33.261297|
|-87.586341,31.959751|
| -87.43091,31.901283|
|-87.989825,33.138512|
|-88.279714,33.056158|
|-87.849593,32.514133|
|-87.727727,32.072313|
|-87.997666,32.067377|
|-87.754018,31.933427|
+--------------------+
only showing top 20 rows
```

Figure 8: Distance Query Output

4. Distance join query: Given a set of Points S1 and a set of Points S2 and a distance D in km, find all (s1, s2) pairs such that s1 is within a distance D from s2 (i.e., s1 belongs to S1 and s2 belongs to S2) [10]. ST_Within is used for this query.

```
+--------------------+--------------------+
|                 _c0|                 _c0|
+--------------------+--------------------+
|-88.331492,32.324142|-88.331492,32.324142|
|-88.331492,32.324142|-88.388954,32.357073|
|-88.331492,32.324142|-88.383822,32.349204|
|-88.331492,32.324142| -88.384664,32.34299|
|-88.331492,32.324142|-88.401397,32.341222|
|-88.331492,32.324142|-88.414987,32.338364|
|-88.331492,32.324142|-88.277689,32.310778|
|-88.331492,32.324142|-88.382818,32.319915|
|-88.331492,32.324142|-88.366119,32.402014|
|-88.331492,32.324142|-88.265642,32.359191|
|-88.175933,32.360763|-88.175933,32.360763|
|-88.175933,32.360763| -88.221102,32.35078|
|-88.175933,32.360763|-88.158469,32.372466|
|-88.175933,32.360763|-88.133374,32.367435|
|-88.175933,32.360763|-88.265642,32.359191|
|-88.388954,32.357073|-88.331492,32.324142|
|-88.388954,32.357073|-88.388954,32.357073|
|-88.388954,32.357073|-88.383822,32.349204|
|-88.388954,32.357073| -88.384664,32.34299|
|-88.388954,32.357073|-88.401397,32.341222|
+--------------------+--------------------+
only showing top 20 rows
```

Figure 9: Distance Join Query Output

3.2 New York Taxi Cab Data Geospatial Analysis

The input dataset is a collection New York City Yellow Cab taxi trips records from 2009 to 2012. This dataset is limited to latitude 40.5N – 40.9N and longitude 73.7W – 74.25W in order to remove noise. Figure 10 shows this geographical rectangle with the blue dots representing every taxi pickup location.

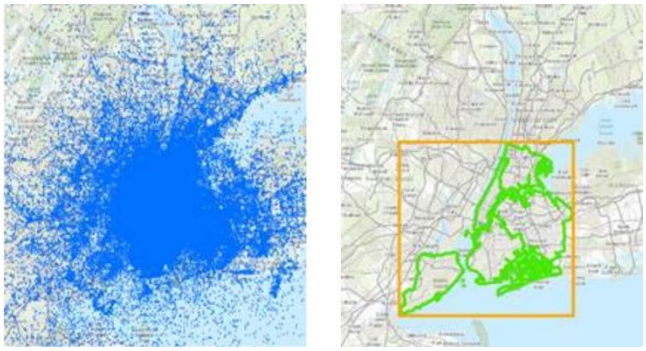

Figure 10: New York Taxi Cab Pickup Locations

It is required to perform spatial hot spot analysis on this New York taxi trip dataset. Specifically, the two following tasks:

3.2.1 *Hotzone Analysis*

This task will need to perform a range join operation on a rectangle dataset and a point dataset. 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.

The algorithm used is as follows:

Hotzone(SparkSession, pointPath, rectanglePath):

1. Fetch the list of points from file stored at pointPath.
2. Fetch the list of rectangles from file stored at rectanglePath.
3. Using ST_Contains, count the number of points which exists in the pointPath file in each rectangle in the rectanglePath of the file.
4. Return the data containing each rectangle and the number of points in each rectangle.

rectangle	count
-73.789411,40.666...	1
-73.793638,40.710...	1
-73.795658,40.743...	1
-73.796512,40.722...	1
-73.797297,40.738...	1
-73.802033,40.652...	8
-73.805770,40.666...	3
-73.815233,40.715...	2
-73.816380,40.690...	1
-73.819131,40.582...	1
-73.825921,40.702...	2
-73.826577,40.757...	1
-73.832707,40.620...	200
-73.839460,40.746...	3
-73.840130,40.662...	4
-73.840817,40.775...	1
-73.842332,40.804...	2
-73.843148,40.701...	2
-73.849479,40.681...	2
-73.861099,40.714...	21

only showing top 20 rows

Figure 11: Hotzone Analysis Output

cellX	cellY	cellZ
-7399	4075	15
-7399	4075	29
-7399	4075	22
-7399	4075	28
-7399	4075	14
-7399	4075	30
-7398	4075	15
-7399	4075	23
-7399	4075	16
-7398	4075	29
-7399	4075	21
-7398	4075	28
-7399	4075	27
-7398	4075	22
-7399	4074	30
-7399	4074	15
-7398	4075	14
-7399	4074	23
-7399	4074	29
-7399	4075	13

only showing top 20 rows

Figure 12: Hotcell Analysis Output

3.2.2 Hotcell Analysis

This task will focus on applying spatial statistics to spatio-temporal big data to identify statistically significant spatial hot spots using Apache Spark. The topic of this task is from ACM SIGSPATIAL GISCUP 2016. It is required to create a space-time cube of latitude, longitude and day as the three axes and perform Getis – Ord G_i* Statistic on this cube and identify the most significant cells in the cube.

The algorithm used is as follows:

Hotcell(SparkSession, pointPath)

1. Fetch the list of points from file stored at pointPath.
2. Parse the points to fetch the latitude, longitude and day on which the taxi pickup occurred.
3. Create the space-time cube by parsing the latitude, longitude and day of each point. The three axes of the space-time cube are the latitude on the x-axis, longitude on the y-axis and day on the z-axis. Each cell in the space-time cube contains the count of the number of taxi pickups that occurs in that latitude and longitude range on that given day.
4. Calculate the Getis – Ord G_i* Statistic for every cell in the space-time cube.
5. Return the values of fifty cells in the descending order of the Getis – Ord G_i* Statistic.

3.3 Space-Time Cube

The space-time cube is a three-dimensional cube with the latitude on the x-axis, the longitude on the y-axis, the day on the z-axis. Two cells on the x and y-axis are separated by a unit of 0.01 degrees and two cells on the z-axis are separated by a unit of 1 day. The value in each cell is the count of the number of taxi pickups that occurred in that day in that range of latitude and longitude.

Figure 13: Space-Time Cube

3.4 Adjacency

Two cells are adjacent to each other if they share either an edge or a vertex with each other. For example, in the Figure 14 which contains a square of 16 cells (a two-dimensional representation for easier understanding), let the cell in red be our cell in consideration.

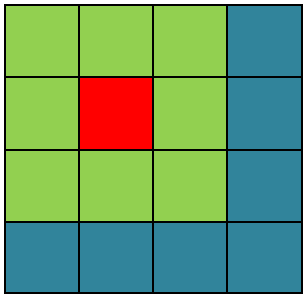

Figure 14: A 16-cell square

All the cells in green are adjacent to red as the they all have either an edge or a vertex in common with the red cell. They all are assigned the spatial weight of 1. All the cells in blue are not adjacent with the red cell as they do not share an edge or a vertex with it. They all are assigned the spatial weight of 0.

3.5 Getis – Ord G_i^* Statistic

It is now required to assign values to measure the significance of each cell. There are a lot of attributes which go into deciding how significant a cell is. A cell with a high value is interesting but may not be a statistically significant hot spot. To be a statistically significant hot spot, a cell will have a high value and be surrounded by other cell with high values as well. The local sum for a cell and its neighbours is compared proportionally to the sum of all cells; when the local sum is much different than the expected local sum, and that difference is too large to be the result of random chance, a statistically significant Z-score results. [11]

The Getis-Ord Gi* statistic is used to identify the most significant hot spot cells in the space-time cube. It is a Z-score; hence no more calculations are required. The Z-Score represents the statistical significance of clustering for a specified distance. For statistically significant positive Z scores, the larger the Z score is, the more intense the clustering of high values (hot spot). For statistically significant negative Z scores, the smaller the Z score is, the more intense the clustering of low values (cold spot). [11]

The formula for the Getis-Ord Gi* statistic is as follows:

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

where,

x_j : attribute value of cell j

n : total number of cells

$w_{i,j}$: spatial weight between cell i and j.

In this case, the spatial weight is defined as follows:

$$w_{i,j} = \begin{cases} 1, & if\ cell\ i\ and\ j\ are\ adjacent\ to\ each\ other \\ 0, & if\ cell\ i\ and\ j\ are\ not\ adjacent\ to\ each\ other \end{cases}$$

$\bar{X}$: mean of the attribute value of all the cells. The formula is as follows:

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

S : standard deviation of the attribute value of all the cells. The formula is as follows:

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

4 EXPERIMENTAL SETUP

4.1 Cluster Settings

The system uses Google Compute Engine's n1-standard-1 VM instances which have the following resource configurations per machine:

vCPU count: 1
Memory: 3.75GB
Base Image: Ubuntu 16.04.3 LTS
Apache Hadoop Version: 3.0.0
Apache Spark Version: 2.2.1

Jobs are run by being submitted on the Master machine within each job, the command line arguments determine the HDFS IO, and actions. As an example, in the Hotzone and Hotcell Analysis Job, the command line submission in the following format:

/usr/local/spark-submit --master spark://10.142.0.2:7077 /home/kunwar_aiesec/CSE512-Hotspot-Analysis-Template-assembly-0.1.0.jar /user/kunwar/phase3/output hotzoneanalysis /user/kunwar/phase3/input/point-hotzone.csv /user/kunwar/phase3/input/zone-hotzone.csv hotcellanalysis /user/kunwar/phase3/input/yellow_tripdata_2009-01_point.csv

4.2 Data and Query Workload

Size	Last Modified	Replication	Block Size	Name
206.56 KB	Mar 18 16:34	3	128 MB	arealm10000.csv
409.97 KB	Mar 18 16:35	3	128 MB	zcta10000.csv

Table 2: Preliminary Spatial Query Analysis Dataset

Size	Last Modified	Replication	Block Size	Name
1.76 MB	Apr 15 17:24	3	128 MB	point_hotzone.csv
2.37 GB	Apr 19 21:12	3	128 MB	yellow_tripdata_2009-01_point.csv
11.73 KB	Apr 15 17:25	3	128 MB	zone-hotzone.csv

Table 3: Hotzone and Hotcell Analysis Dataset

The tables above show the sizes of the datasets used on the HDFS and as the yellow trip data set is a considerable size, querying information takes more time than other analyses. Furthermore,

calculating the Getis – Ord G_i^* score requires several runs of exhaustively checking cell adjacency, which is a query task with high selectivity.

4.3 Monitoring Setup

4.3.1 Spark Analysis: When a job is submitted to the Spark system, the Master device provides an execution monitoring web interface on port 4040. This port is then forwarding to a local machine using the Google SDK.

4.3.2 Google Cloud Monitoring: In this setup, Google Compute Engine provides the facility to monitor the CPU, Network and Disk usage across VM instances and this information is seen in the VM System Workloads sections.

5 EVALUATION OF RESULTS

5.1 SparkSQL Spatial Query Analysis

5.1.1 Spark Analysis: For the processing of points within rectangles with a *point* and *rectangle* dataset (discussed in section 3.1). The following figures show Spark execution details for this job:

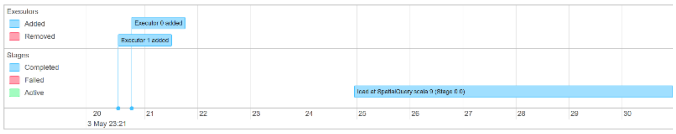

Figure 15: Event timeline – Load stage of SpatialQuery.scala

Figure 15 describes the various stages of the spark job executed as a part of Phase 2 tasks. During the experiment, this timeline shows operations performed such as range, range join, distance and distance join queries.

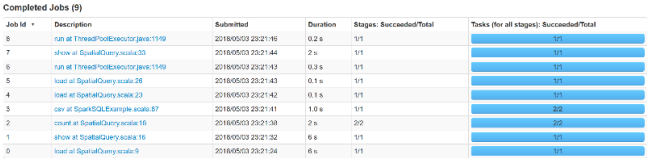

Figure 16: Dynamic Jobs List

Figure 16 describes the various jobs executed as a part of phase 2 task execution. It shows the time taken to complete each job along with the order in which the various jobs that were executed.

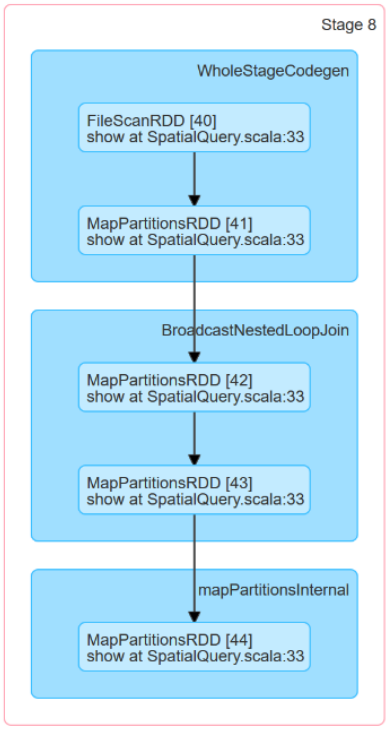

Figure 17: DAG - Map transformation stages in SpatialQuery.scala

Figure 17 describes the Spark DAG that is constructed for phase 2 task. The blocks denote the nodes of the graph with their respective operations while the arrows denote the edges of the graph and signify the order of the operations executed.

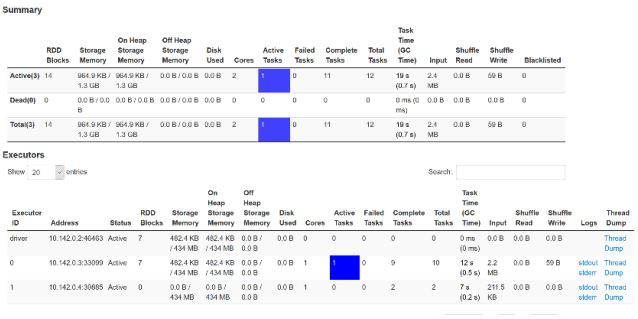

Figure 18: Executor Information that details current memory usage for Master and Workers

Figure 18 describes the memory usage on both the master and the worker machines during the execution of the phase 2 task.

Job Id	Description	Submitted	Duration
19	run at ThreadPoolExecutor.java:1149	2018/05/03 23:26:54	0.1 s
18	show at SpatialQuery.scala:63	2018/05/03 23:26:53	0.4 s
17	run at ThreadPoolExecutor.java:1149	2018/05/03 23:26:53	0.2 s
16	load at SpatialQuery.scala:57	2018/05/03 23:26:52	0.1 s
15	load at SpatialQuery.scala:54	2018/05/03 23:26:52	0.1 s
14	csv at SparkSQLExample.scala:87	2018/05/03 23:26:52	0.2 s
13	count at SpatialQuery.scala:49	2018/05/03 23:26:50	0.9 s
12	show at SpatialQuery.scala:47	2018/05/03 23:26:50	0.2 s
11	load at SpatialQuery.scala:40	2018/05/03 23:26:50	0.2 s
10	csv at SparkSQLExample.scala:87	2018/05/03 23:26:49	0.2 s
9	count at SpatialQuery.scala:35	2018/05/03 23:21:46	5.0 min
8	run at ThreadPoolExecutor.java:1149	2018/05/03 23:21:46	0.2 s
7	show at SpatialQuery.scala:33	2018/05/03 23:21:44	2 s
6	run at ThreadPoolExecutor.java:1149	2018/05/03 23:21:43	0.3 s
5	load at SpatialQuery.scala:26	2018/05/03 23:21:43	0.1 s
4	load at SpatialQuery.scala:23	2018/05/03 23:21:42	0.1 s
3	csv at SparkSQLExample.scala:87	2018/05/03 23:21:41	1.0 s
2	count at SpatialQuery.scala:18	2018/05/03 23:21:38	2 s
1	show at SpatialQuery.scala:16	2018/05/03 23:21:32	6 s
0	load at SpatialQuery.scala:9	2018/05/03 23:21:24	6 s

Figure 19: SparkSQL Total Completed Jobs List

From Figure 19, we can see that the total time taken to perform the range, range join, distance and distance join queries is approximately 5 minutes.

Summary

	RDD Blocks	Storage Memory	On Heap Storage Memory	Off Heap Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(3)	22	1.5 MB / 1.3 GB	1.5 MB / 1.3 GB	0.0 B / 0.0 B	0.0 B	2	1	0	26	27	5.4 min (23 s)	4.8 MB	0.0 B	177 B	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B / 0.0 B	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(3)	22	1.5 MB / 1.3 GB	1.5 MB / 1.3 GB	0.0 B / 0.0 B	0.0 B	2	1	0	26	27	5.4 min (23 s)	4.8 MB	0.0 B	177 B	0

Figure 20: Executor Information that details current memory usage information for Master and Workers (Job 9)

Figure 20 shows the memory usage information for both master and worker machines on executing job 9 (range, range join, distance, distance join queries) as described in Figure 19.

5.1.2 VM System Workload: To show the VM level usage metrics on Google Compute Engine

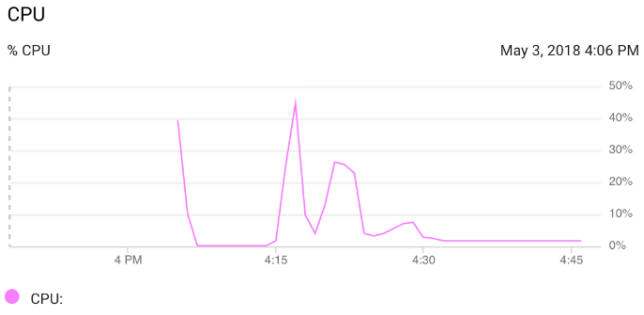

Figure 21: VM Instance 1 (Master) – CPU Usage

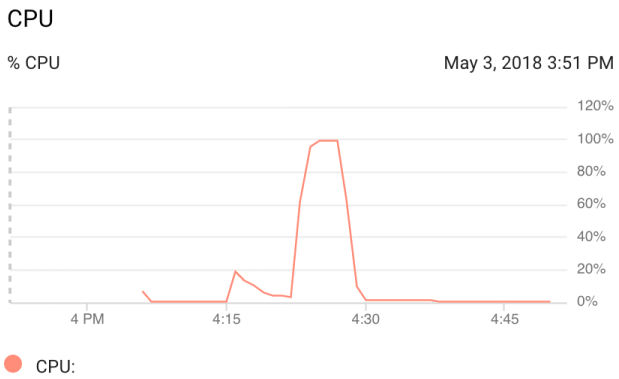

Figure 22: VM Instance 2 (Worker 1) – CPU Usage

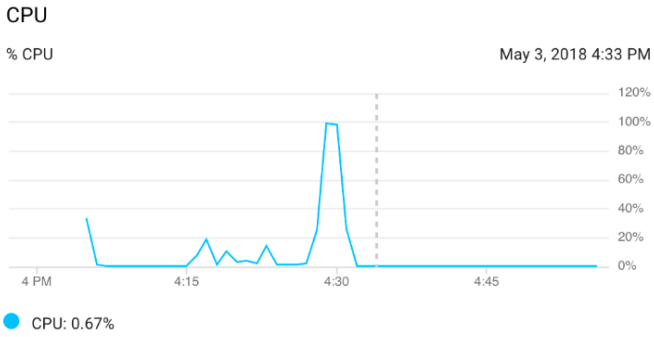

Figure 23: VM Instance 3 (Worker 2) – CPU Usage

Figures 21, 22 and 23 show the VM CPU usage on the master and the worker machines during the execution of jobs. As expected, once the job is submitted to the spark master, we see a sharp rise in the CPU usage on all the 3 machines owing to the execution of the job.

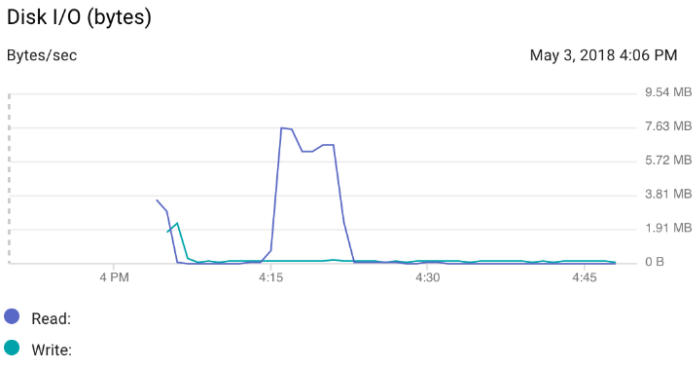

Figure 24: Master – Disk IO (in Bytes/sec)

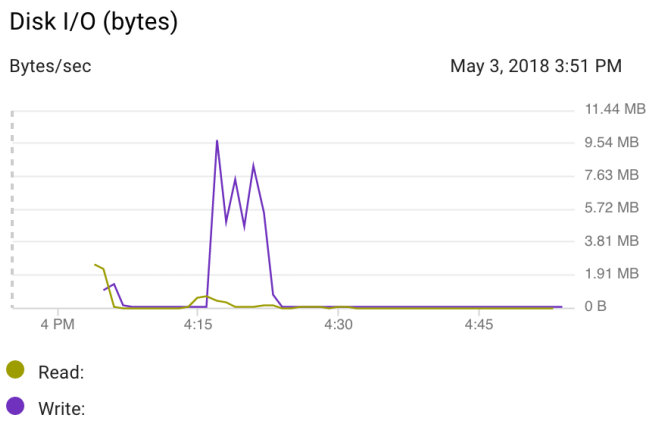

Figure 25: Worker 1 – Disk IO (in Bytes/sec)

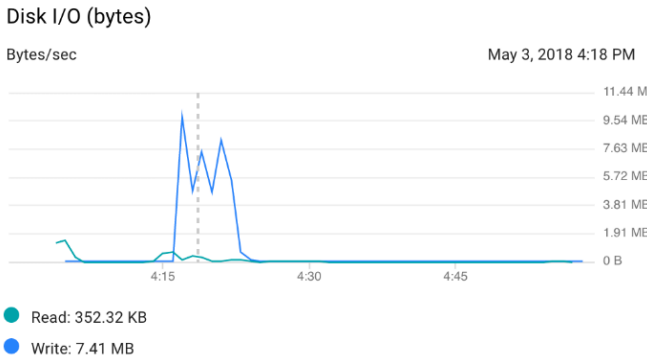

Figure 26: Worker 2 – Disk IO (in Bytes/sec)

Figures 24, 25 and 26 show the VM Disk IO usage on the master and the worker machines during the execution of jobs. As expected, once the job is submitted to the spark master, we see a sharp rise in the Disk IO on all the 3 machines. We can attribute this to the results being written to the Hadoop Distributed File System (HDFS).

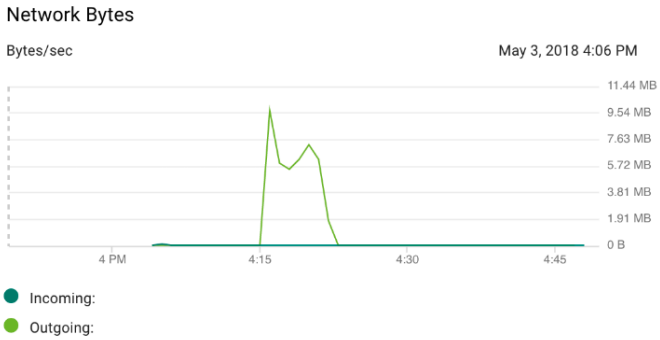

Figure 27: Master – Network Usage (in Bytes/sec)

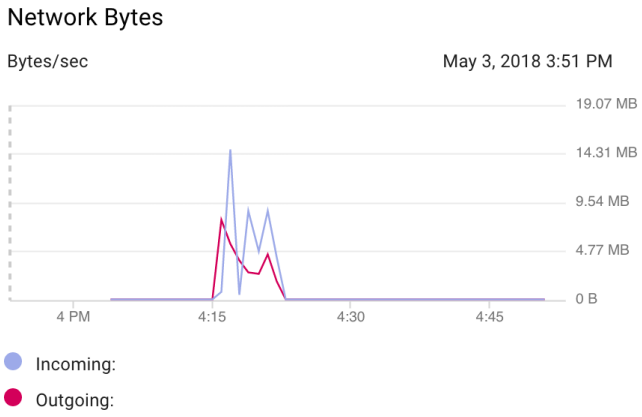

Figure 28: Worker 1– Network Usage (in Bytes/sec)

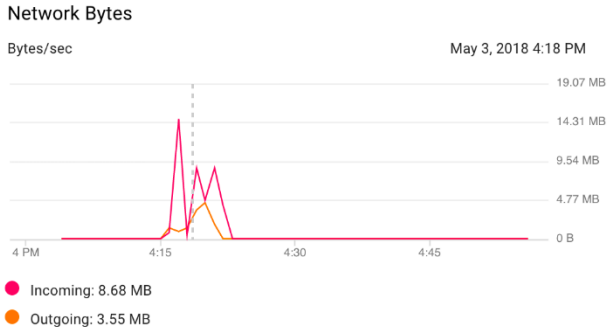

Figure 29: Worker 2 – Network Usage (in Bytes/sec)

Figures 27, 28 and 29 show the VM Network Usage on the master and the worker machines during the execution of jobs. Once the job is submitted to the spark master, we see a sharp rise in the Network Usage on all the 3 machines. We can attribute this to the data (residing in HDFS) being transferred between the machines during the execution of the job.

5.2 Hotzone and Hotcell Analysis

5.2.1 Spark Analysis: for the processing of hotzones (number of points in a rectangle) and hotcells (high Getis-Ord regions) . The following figures show the Spark execution details for this job:

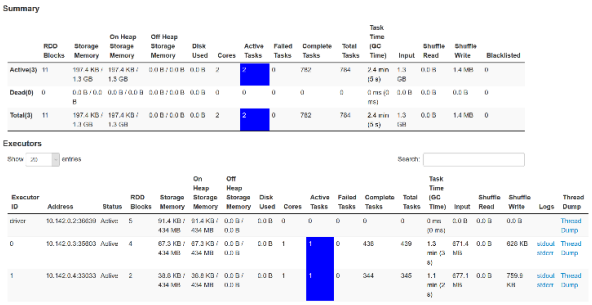

Figure 30: Executor Information: Master and Workers Memory for Hotzone Analysis

Figure 30 describes the memory usage on both the master and the worker machines during the execution of the Hotzone analysis task.

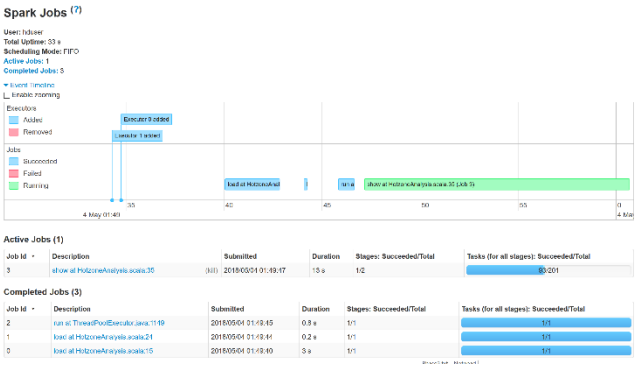

Figure 31: Event Timeline for Hotzone Analysis

Figure 31 describes the various stages of the spark job executed as a part of Hotzone analysis.

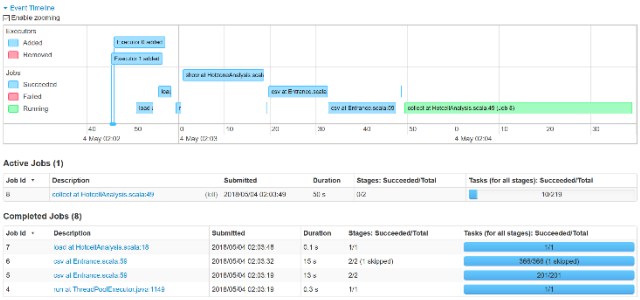

Figure 32: Event Timeline for Hotcell Analysis

Figure 32 describes the various stages of the spark job executed as a part of Hotcell analysis.

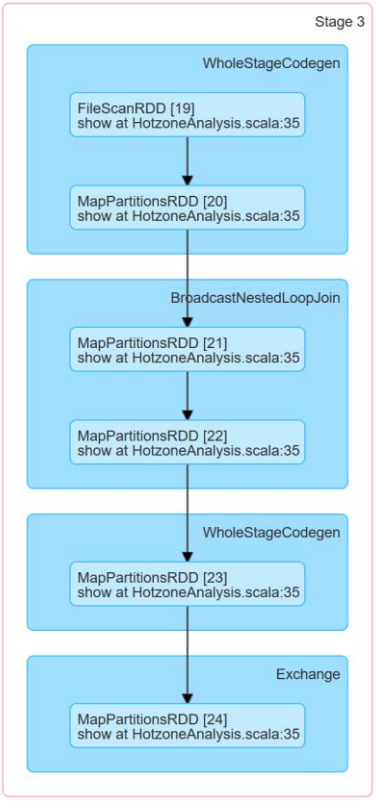

Figure 33: DAG Execution for Hotzone Analysis

Figure 33 describes the Spark DAG that is constructed for Hotzone analysis. The blocks denote the nodes of the graph with their respective operations while the arrows denote the edges of the graph and signify the order of the operations executed.

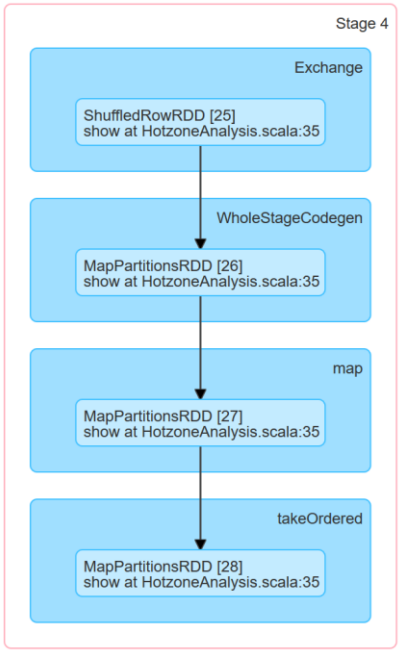

Figure 34: DAG Execution for Hotcell Analysis

Figure 34 describes the Spark DAG that is constructed for Hotcell analysis. The blocks denote the nodes of the graph with their respective operations while the arrows denote the edges of the graph and signify the order of the operations executed.

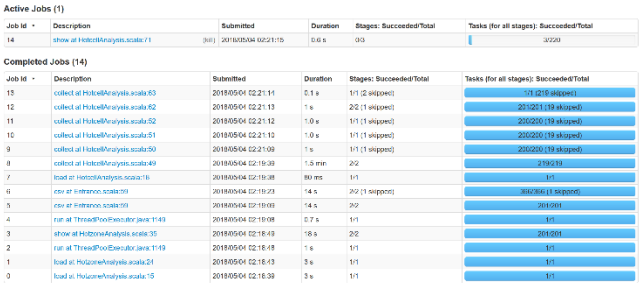

Figure 35: **Event Timeline - Completed Jobs**

Figure 35 describes the various jobs executed as a part of phase 3 task execution. It shows the time taken to complete each job along with the order in which the various jobs that were executed.

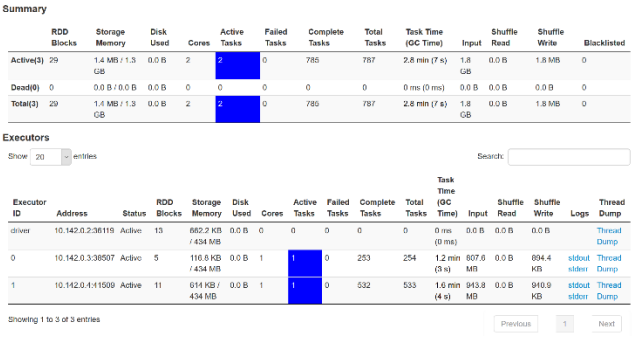

Figure 36: Executor Information for Hotcell Analysis

Figure 36 describes the memory usage on both the master and the worker machines during the execution of the Hotcell analysis task.

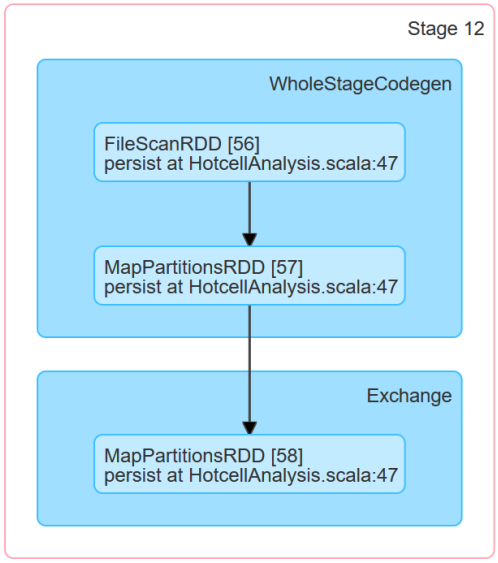

Figure 37: DAG Execution: FileScan and Map transformations for Hotcell analysis

Figure 37 shows the Map and FileScan transformations and the DAG preparing the RDD for analysis.

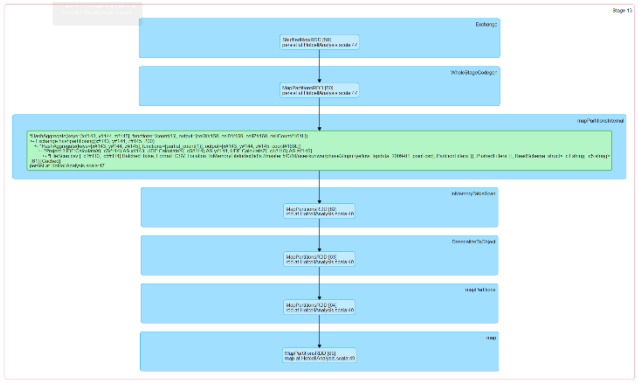

Figure 38: DAG Execution of Hotcell Analysis SparkSQL Query

Figure 38 shows the DAG for the task that involves constructing the space-time cube and calculating the Getis – Ord G_i^* for each cell.

5.2.2 VM System Workload: To show the VM level usage metrics on Google Compute Engine

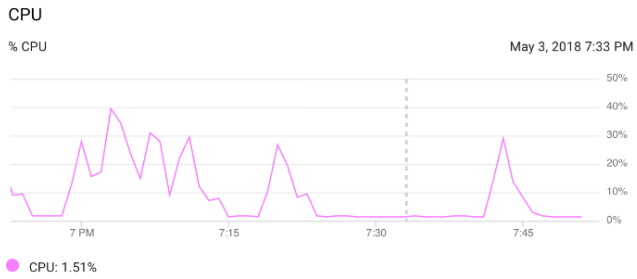

Figure 39: CPU usage percentage for Master

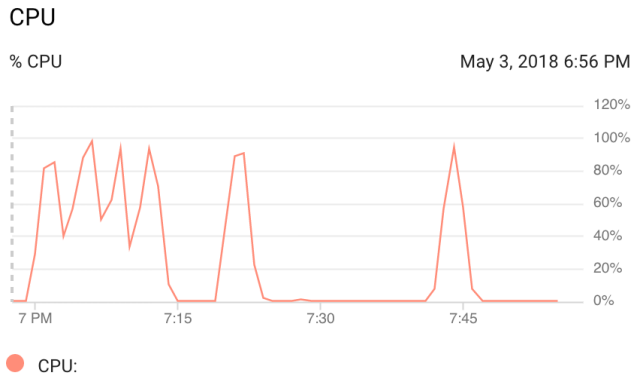

Figure 40: CPU usage percentage for Worker 1

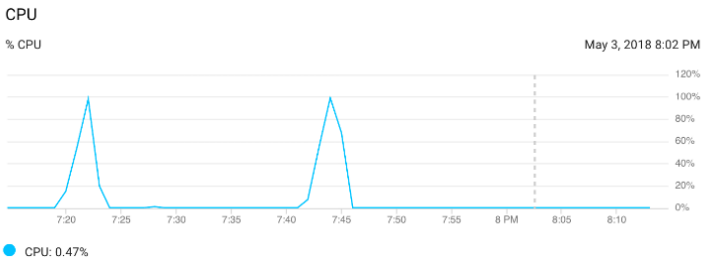

Figure 41: CPU usage percentage for Worker 2

Figures 39, 40 and 41 show the VM CPU usage on the master and the worker machines during the execution of phase 3 jobs. As expected, once the job is submitted to the spark master, we see a sharp rise in the CPU usage on all the 3 machines owing to the execution of the job.

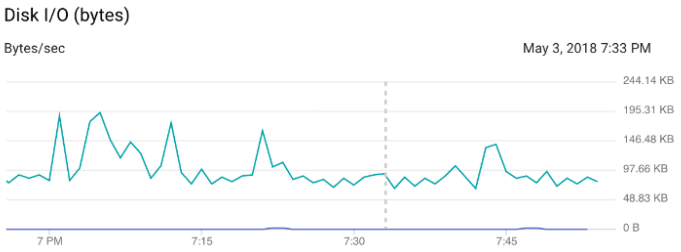

Figure 42: Disk I/O usage in bytes for Master

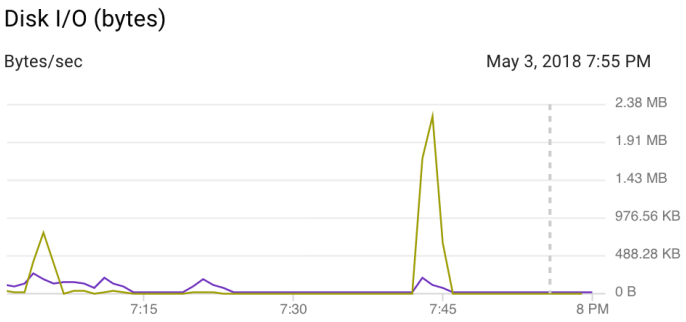

Figure 43: Disk I/O usage in bytes for Worker 1

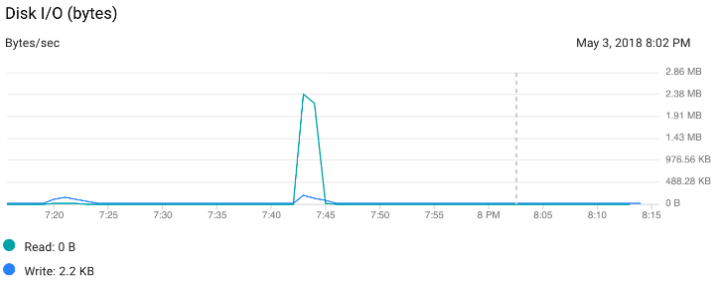

Figure 44: Disk I/O usage in bytes for Worker 2

Figures 42, 43 and 44 show the VM Disk IO usage on the master and the worker machines during the execution of phase 3 jobs. As expected, once the job is submitted to the spark master, we see a sharp rise in the Disk IO on all the 3 machines. We can attribute this to the results being written to the Hadoop Distributed File System (HDFS).

Figure 45: Network usage in bytes for Master

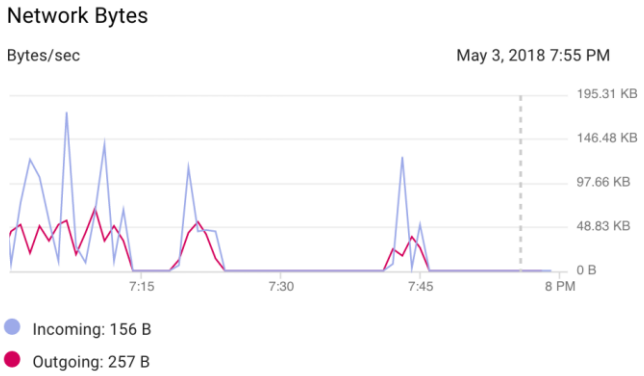

Figure 46: Network usage in bytes for Worker 1

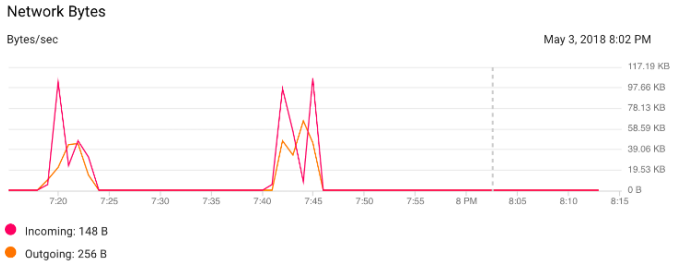

Figure 47: Network usage in bytes for Worker 2

Figures 45, 46 and 47 show the VM Network Usage on the master and the worker machines during the execution of phase 3 jobs. Once the job is submitted to the spark master, we see a sharp rise in the Network Usage on all the 3 machines. We can attribute this to the data (residing in HDFS) being transferred between the machines during the execution of the job.

7 CONCLUSIONS

When it comes to in-memory computations, caching and fault tolerance, Apache Spark is miles ahead. These features helped in reducing the run-time of our operations which otherwise for a large project like this would have taken a lot more time. Spark coupled with HDFS is a great storage platform for storing large files of data. Understanding Apache Spark coupled with Hadoop really helped us comprehend the need for a Distributed and Parallel Database System and of its advantages. The SparkSQL APIs on Scala helped us understand how to deal with geospatial data.

8 ACKNOWLEDGMENTS

We would like to thank Professor Mohamed Sarwat and the Teaching Assistant, Yuhan Sun for giving us a project that helped in improving our understanding of the distributed and parallel database systems and frameworks such as Apache Spark, Hadoop, and GeoSpark.

9 REFERENCES

[1] http://sigspatial2016.sigspatial.org/giscup2016/problem
[2] https://github.com/DataSystemsLab/GeoSpark
[3] https://hadoop.apache.org/
[4] https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html
[5] https://ieeexplore.ieee.org/document/7498357/
[6] https://medium.com/google-cloud/apache-spark-cluster-on-google-cloud-platform-c99d8ebfc248
[7] http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/
[8] https://blog.tinned-software.net/understand-the-basics-of-rrdtool-to-create-a-simple-graph/
[9] https://cloud.google.com/community/tutorials/ssh-port-forwarding-set-up-load-testing-on-compute-engine
[10] http://www.qgistutorials.com/en/docs/performing_spatial_joins.html
[11] http://resources.esri.com/help/9.3/arcgisengine/java/gp_toolref/spatial_statistics_tools/how_hot_spot_analysis_colon_getis_ord_gi_star_spatial_statistics_works.htm
[12] https://spark.apache.org/docs/2.1.0/sql-programming-guide.html#overview