Distributed and Parallel Databases: Project Report

Purushotham Swaminathan
Arizona State University
pswamin1@asu.edu

Suresh Gururajan Nolastname
Arizona State University
sgurura3@asu.edu

Thamizh Arasan Rajendran
Arizona State University
trajendr@asu.edu

Aravind Rajendran
Arizona State University
arajend6@asu.edu

Balaji Chandrasekaran
Arizona State University
bchandr6@asu.edu

ABSTRACT

In essence, our project involved familiarization phase with the GeoSpark API, run-time statistical analysis for distributed computation techniques like Spatial Range Query, Spatial KNN query, Spatial Join query and their variations obtained by with and without joining the PointRDD using Equal grid or building R-tree indices. We successfully implemented the tasks on a Hadoop cluster running spark shell with geo-spark jar included. The third phase involved spatio-temporal analysis of a subset of the NYC yellow-cab taxi dataset (Jan 2015 data), to compute the Getis-Ord statistic[2] as a z-score to identify 50 hot spots within the mentioned grid. We studied various approaches mentioned in [3, 4] and designed our approach towards this problem.

CCS Concepts

Information system → Data management systems → Database administration → Database performance evaluation,

Information systems → Data management systems → Database management system engines → Parallel and distributed DBMSs → MapReduce-based systems

Information systems →Information Systems applications → Spatial-temporal systems → Geographic information systems

Keywords

spatio-temporal analysis, Hadoop, spark, MapReduce, hotspot, getis-ord, GIS cup,

INTRODUCTION

Problem Statement and approach:

Our project involved 3 phases:

Phase1:

First to get a hands on experience with GeoSpark jar and spatial functionalities. It involved 3 tasks:
TASK 1: Setting up the Hadoop cluster and loading the GeoSpark jar file into the Spark environment and performing operations using Scala shell.

TASK 2.1: Spatial Range Query using query window [x1(35.08), y1(-113.79), x2(32.99), y2(-109.73)] and query after building R-Tree index on PointRDD.

TASK 2.2: Spatial KNN query using query point [x1(35.08), y1(-113.79)] for 5 Nearest Neighbors and query after building R-Tree index on PointRDD.

TASK 3. Spatial Join query and using it to join PointRDD with 3 variants:

3.1 Joining using Equal grid without R-Tree index.

3.2 Joining using Equal grid with R-Tree index.

3.3 Joining using RTree grid without R-Tree index.

Phase2:

Second, there were 2 components: one, was to implement a spatial join using basic Cartesian join algorithm which was done in java and it involved querying each rectangle from the window dataset against the points dataset using methods from the GeoSpark (spatial range query), and in the second task we got to provide a statistical comparison for the tasks mentioned in phase 1 and provide reasons for the difference using the following run-time statistics:

1. Execution time
2. Average memory
3. Average CPU utilization of the entire cluster

Phase3:

In this phase we are asked to use the GIS Cup dataset and the problem statement to identify 50 hotspots using Getis-Ord correlation.

About the GISCUP 2016 and data:

The data that involves transportation of people and their means has significant applications in many fields. Some of the sample fields are Urban planning and transportation management, etc.[3] Hotspots denotes the point coordinates or on a geographical system a place of intense activity that could pave way for newer solutions. A standard way of measuring the correlation of the hot-spots on spatio-temporal data is the Getis-Ord statistic. When we filter the data we can clearly identify the areas within s given geographic location that shows a similarity or correlation with another.

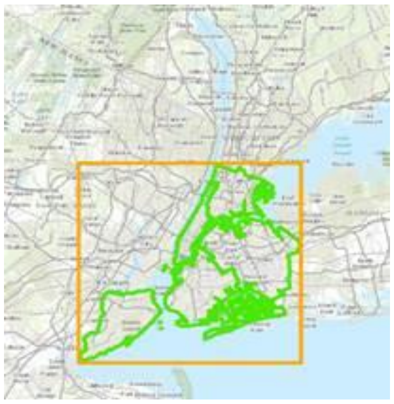


Figure 1: The figure shows the envelope of data that is considered for calculation purpose. [4]

As this data is periodic(temporal) and localized to spatial dimensions(spatial) we would have to run massive datasets for our computation and we resort to the Hadoop distributed computing using Spark environment that has the potential to distribute the computation and merge to form an overall result. Spatial data research and data mining is a specific field of research. GIS cup is a Geographical information systems focused competition that brings together academia and industries to incorporate efficient ways to process geographically obtained data. The dataset provided by the GIS cup organizers comprises vehicular data of NYC Yellow Cab taxi trip dataset that has been consolidated for a period of 6years from 2009-2015 and contains pickup and drop-off times and locations. The data set is provided as a uniform spatial grid and can be constructed into 3 dimensional cells that have both spatial and temporal data.

1.1 Phase I Solution

We completed a video submission that demonstrated the tasks of 1. Setting up the Hadoop cluster

2. Loading the GeoSpark jar file into the Spark environment and performing operations using Scala shell.

3.1. Spatial Range Query using query window [x1(35.08), y1(-113.79), x2(32.99), y2(-109.73)] and query after building R-Tree index on PointRDD.

3.2 Spatial KNN query using query point [x1(35.08), y1(-113.79)] for 5 Nearest Neighbors and query after building R-Tree index on PointRDD.

3.3. Spatial Join query and using it to join PointRDD with 3 variants

3.3.1 Joining using Equal grid without R-Tree index.

3.3.2 Joining using Equal grid with R-Tree index.

3.3.3 Joining using RTree grid without R-Tree index.

1.2 Phase II Solution

This project phase aims at comparing the various Spatial join, range and nearest neighbor operations on GeoSpark running on a Hadoop cluster, by monitoring cluster resources like memory usage, CPU usage and execution time, averaged over a few runs.

1.2.1 Summary of phase II

In this project, by analyzing the cluster CPU, Memory metrics and execution times of Spatial Range Query, KNN Query, Spatial Join query and Cartesian product based Join we find that — 1) queries that use indexes are faster albeit consuming more memory than regular queries that don't use indexes, 2) the Cartesian join query is considerably slower than the nested loop queries that have grid partitioning.

1.2.2 Cluster setup

We setup 3 Ubuntu machines running Hadoop 2.6.4 and Spark 2.0.1.

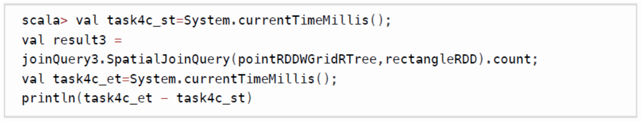
System	Memory	CPU	Cores
Master	2.9 GB	Intel i5	2
Worker1	6.7 GB	Intel i5	2
Worker2	6.7 GB	Intel i7	4

Table 1

1.2.3 Metrics Collection

In order to collect metrics, we installed ganglia daemon on the master node and ganglia monitoring on the worker nodes.

From the ganglia UI, we created a window in the monitor dashboard that would capture each query's execution. For e.g. the queries 2a, 2b, 3a, 3b which run within a minute, we set a 1-minute window and collected the average value for the cluster's memory and CPU utilization. Similarly, for 4a, 4b, 4c we set a 5-min window and for the phase 2 query we set a 3-hr. window. We run the queries multiple times and set window sizes accordingly and collect averages over the window. For execution time, we used the age-old method of collecting start and end time in the Scala shell and taking their difference to yield the query execution time.



```
scala> val task4c_st=System.currentTimeMillis();
val result3 =
joinQuery3.SpatialJoinQuery(pointRDDWGridRTree,rectangleRDD).count;
val task4c_et=System.currentTimeMillis();
println(task4c_et - task4c_st)
```

Figure 2

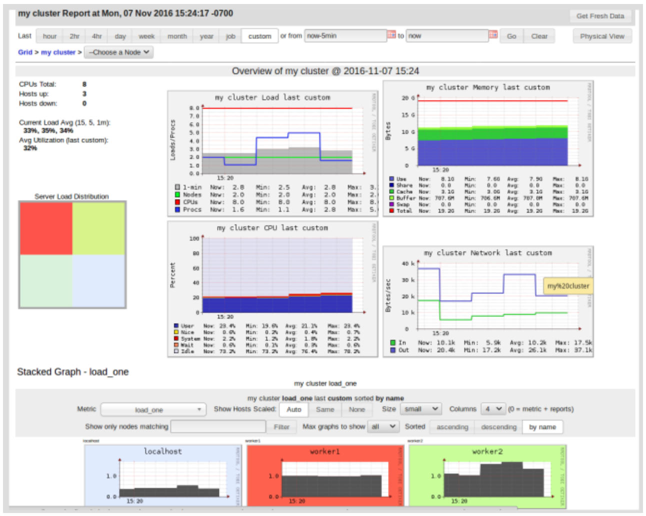

my cluster Report at Mon, 07 Nov 2016 15:24:17 -0700
Overview of my cluster @ 2016-11-07 15:24
my cluster Load last custom
my cluster Memory last custom
my cluster CPU last custom
my cluster Network last custom
Server Load Distribution
Stacked Graph - load_one
localhost
worker1
worker2

Figure 3

1.2.4 Queries comparison and explanations

With the above setup, we ran the queries from Project phase I and collected the metrics below:

1. Comparison of 2a and 2b:

Phase 1	Average Execution time (in milliseconds)	Average cluster memory in GB	Average CPU usage %
2b	1284.333333	9.1	17.26666667
2a	2756.333333	9.2	17.4

Table 2

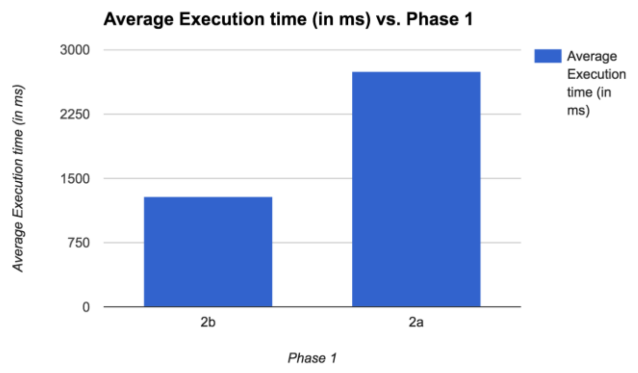


Figure 4

Observation:

Here we see that the indexed range query takes less time than the non-indexed one, which is as expected.

2. Comparison of 3a and 3b:

Phase 1	Average Execution time (in milliseconds)	Average cluster memory in GB	Average CPU usage %
3a	4411.666667	6.7	20.7333333
3b	1778.666667	7.9	18.3

Table 3

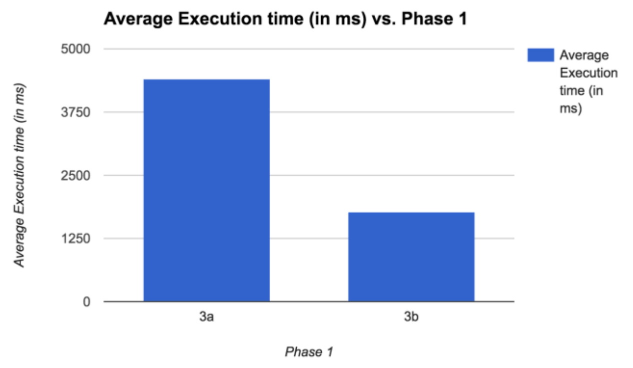


Figure 5

Observation:

Since the index is precomputed in 3b, the KNN query will have lesser number of shuffle stages than the 3a query. This is why 3b is faster than 3a.

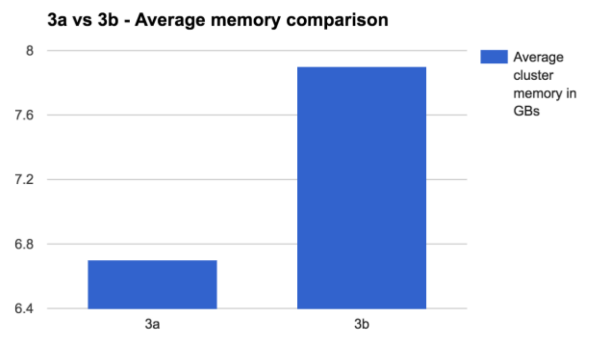


Figure 6

Observation:

For the above queries, the query with index 3b consumes more memory for constructing the index, but the computation is faster hence the execution time is smaller than the non-index query 3a.

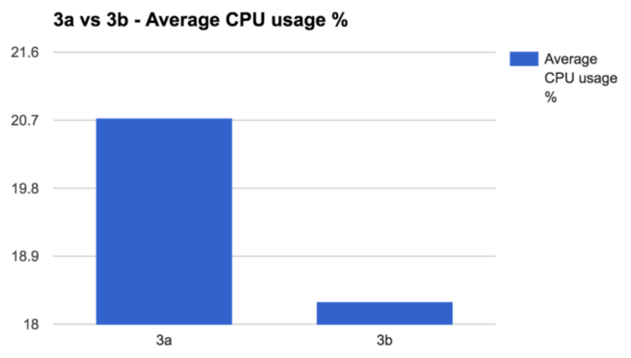


Figure 7

3. **Comparison of 4a and 4b:**

Phase 1	Average Execution time (in ms)	Average cluster memory in GBs	Average CPU usage %
4a	279419.3333	8.4	20.93333333
4b	76769.33333	9.5	18.93333333

Table 4

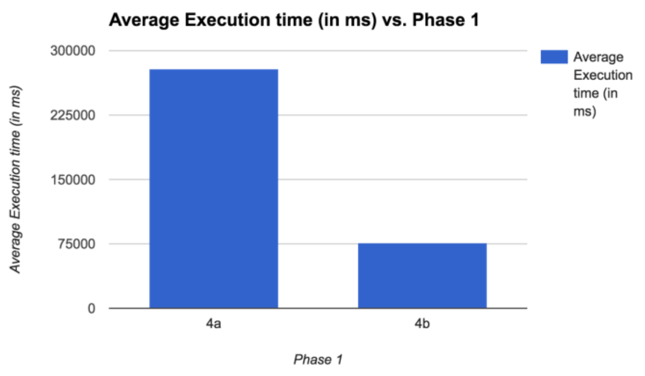


Figure 8

Observation:

Query 4a is a spatial join query with equal grid partitioning and Query 4b is the same query with an RTree index on the PointRDD. The index speeds up the Spatial Join query as it would make 4b an index nested loop join instead of a simple nested loop which is what 4a would do.

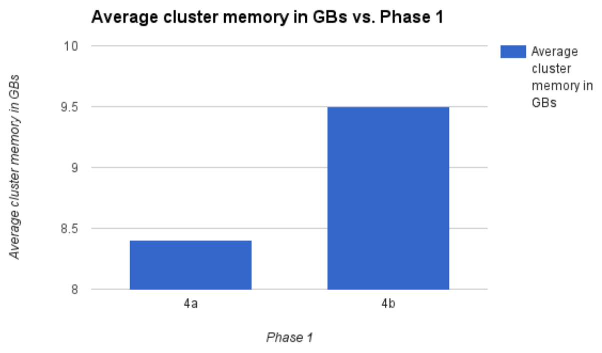


Figure 9

Observation:

On the memory front, building an index will consume more memory than the non-indexed join query. Hence we see an increase in memory usage on the cluster for 4b over 4a.

4. Comparison of 4a and 4c:

Phase 1	Average Execution time (in ms)	Average cluster memory in GBs	Average CPU usage %
4a	279419.3333	8.4	20.93333333
4c	208806	9.6	24.13333333

Table 5

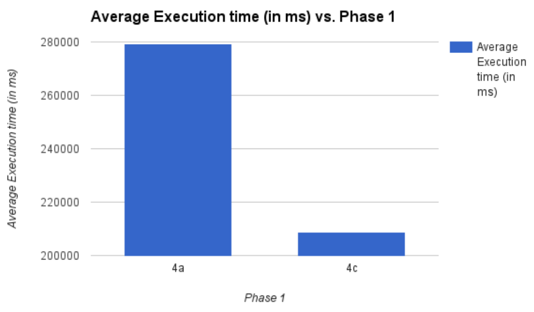


Figure 10

Observation:

Query 4c builds an RTree partitioning over the PointRDD. While both queries are slower than query 4b which has an RTree index, query 4c is better in terms of execution time to query 4a. This is probably because the RTree partitioning has better access time to points during the nested loop join query than the equal grid, hence speeding up the query comparatively.

5. Comparison of Phase 2 Cartesian join with Phase 1 4(a), (b), (c)

	4a	4b	4c	Phase 2
Execution Time(in ms)	279419.3333	76769.33333	208806	12021774.5

Table 6

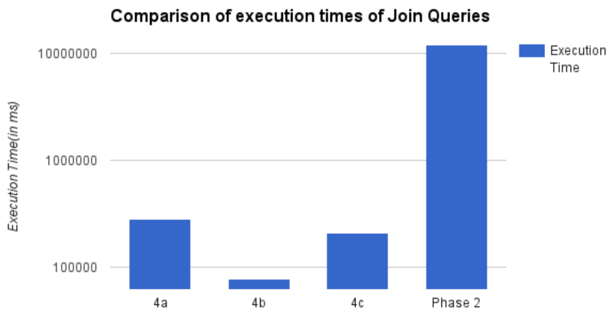


Figure 11

Observation:

The execution time of the Cartesian join query is almost 40 times slower than all the other join queries. This is since the nested loop queries in 4a,4b,4c have grid partitioning which partitions the PointRDD and utilizes spark's parallelism, while the Cartesian join query collects all rectangles and runs a spatial range query and further collects the PointRDD results. This slows down the spark processing since it doesn't utilize spark's parallelism.

	4a	4b	4c	Phase 2
Memory	8.4	9.5	9.6	9.7

Table 7

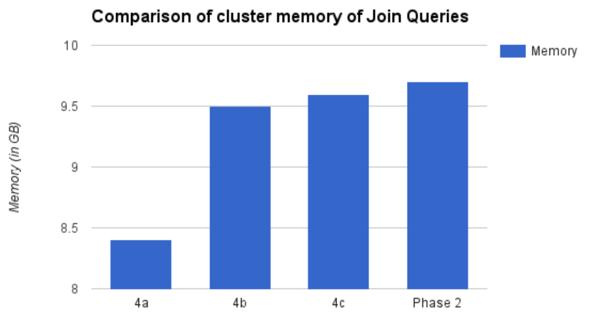


Figure 12

Observation:

The phase 2 query takes more memory on average than the phase 1 queries as it collects the results at the master. Also 4a takes less memory than 4b, 4c as it doesn't build an index like 4b and utilizes only the equal grid partitioning.

	4a	4b	4c	Phase 2
CPU	20.93333333	18.93333333	24.13333333	21.9

Table 8

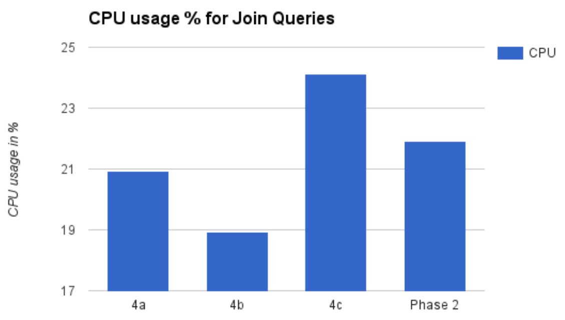


Figure 13

Observation:

On the CPU front, the RTree indexed query 4b consumes the least CPU time among all queries.

1.3. Phase III Solution

1.3.1 Dataset Description:

The trip data consists of NYC yellow taxi pickup and drop times, pickup and drop locations, trip distances and other miscellaneous data about the fare, rate and payment types, etc. In addition, it also consists of the number of passengers that were picked up and dropped during the trip. The entire trip dataset spans from 2009 to 2015 consisting of all the trip during every time of the day.

The data that is relevant and important a for our project are the pickup coordinates (latitude and longitude) and the time of pickup using statistical measures. We need only date for the month of January of 2015. The entire data spans for 31 days for the month of January.

1.3.2 Getis-Ord formula info

Getis ord Statistic is a kind of z-score that is used to denote the clustering pattern of some spatial type data [8]. There are many other similar z-score calculation methods but Getis Ord score is one of the most standard and is highly used as well.

As the trip dataset consists of points of pickup and time of pick-up. The entire area under consideration can be taken to contain smaller regions of pick up or drop operation. The structure is represented as a cell and it is taken as a unit to denote the entire span of the given area. The hotspot calculation is dependent on the number of trips that the passengers were picked up inside each cell.

Each cell data structure is allotted a score based on the Getis-Ord statistic calculated as follows.

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

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

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

_X – Specifies the mean of all the scores of cells.
$W_{i,j}$ -- denotes the neighbor parameter of the cell w.r.t. c_i
x_i -- denotes the value of cell c_i (the sum of all trips within cell c_i)
n -- denotes the number of cells considered overall
S -- Standard Deviation of a cell c_j

1.3.3 Approach:

We have followed the generic approach mentioned in the paper "Spatio temporal hotspot computation on Apache Spark" by Paras Mehta, Christian Windolf, Anges Voisard [3] for the GIS CUP competition.

The approach consists of 4 stages in total:

1. **Pre-Processing Data from the taxi trip dataset:**

a. Reading the data from the CSV into the HDFS for further processing. The CSV contains the data in the format [taxi pickup and drop times, pickup and drop locations, trip distances and other miscellaneous data about the fare, rate and payment types.] information.

b. Removing the header from the csv as the basic cleaning step to obtain raw data to process.

c. Removing the unnecessary columns from the raw data, and to retain only the necessary ones like latitude, longitude and time of the trip.

d. Removing outliers. This involves making the envelope of latitude and longitude concise enough such that it is made to occupy the data the is necessary and remove all the area where there is no area cover. This is just to consider only the areas which show any kind of trip activity. The envelope considered is 40.5, –40.9, -74.25, -73.7.

2. **Cell Creation**

The entire area under consideration has to be split into smaller units for easy computation, hence the unit of cell is considered. A cell is typically 0.01 latitude high and 0.01 longitude wide. The trips within the cell are the score of the cell. The cells have a concept of neighborhood associated with them.

The neighbors of a cell are defined as the adjacent cells to a particular cell under consideration in the spatio temporal domain. Spatial because, a cell is related to nearby cells in the same temporal plane, temporal because, a particular cell is related to the activity during different times of activity. We have implemented a naïve form of neighborhood mapping and facilitating the Getis Ord score calculation for each cell based on the neighborhood.

3. **Score Computation:**

The Getis Ord statistic computation for each cell is taken as a reduce stage and information about the neighbors of the cell is taken into consideration to do so. A cell under consideration could be one that is complete, surrounded by other cells on all sides (26 neighbors) or could be an edge of the boundary, consisting of (17 neighbors) or could be a corner (with 11 neighbors). Thus 3 different cases of cell neighborhood calculation is important.

There is a map stage that localizes each point of activity to a specific cell, both in the spatial and temporal planes and assigns a key that comprises the spatio-temporal coordinates and assigns a value of 1.

There is a reduce phase where the cell score are collected and grouped into single cell key and values as the sum of all the scores. This is done inside the RDD inside partitions and different partitions are unaware of the other's data.

4. **Top hotspots retrieval:**

For the hotspots retrieval, we need the hash map consisting the cell value as the key and Getis Ord Score as the value to be sorted based on Getis Ord Score. This task of sorting the hash map is taken as a shuffle stage and reduced to a list of values from which the information about the top k hotspots can be extracted.

1.3.4 Flow Chart

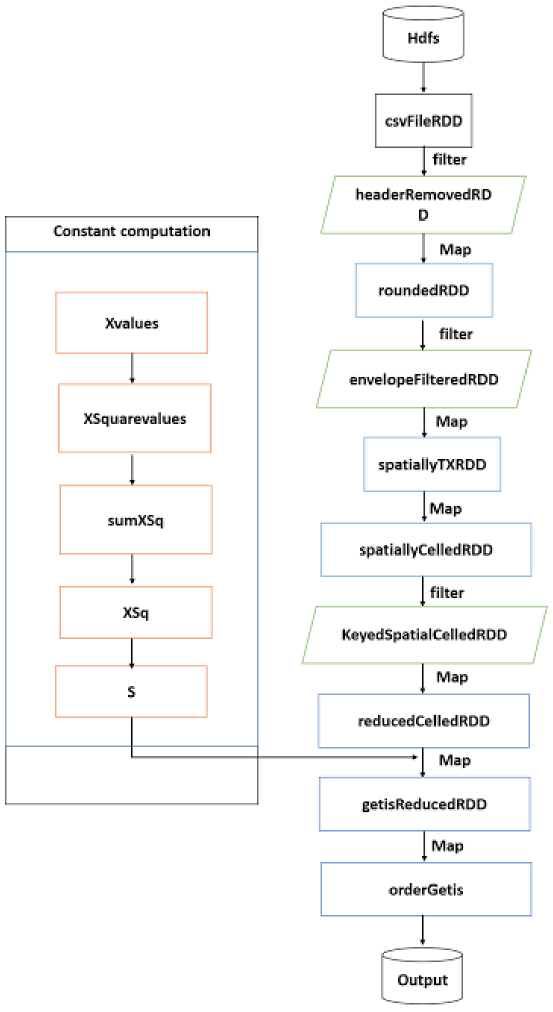


Figure 14: Flow chart of process

Algorithm explanation

Step 1. Load data file from HDFS and filter columns: In this process, we use a map operation to parse the data file, remove outliers and restrict our dataset to just three columns: pickup latitude, longitude and date. This is because we are interested in number of pickups per location per day as the attribute value.

Step 2. Remove outliers:
We then restrict our dataset to include data only within this range: latitude: (40.5N, 40.9N), longitude: (73.7W, 74.25W), which is done by a filter operation.

Step 3. Make pickup points relative to origin: Here, we map all the points in the dataset to start with the origin, i.e., the lower left corner of the dataset is set to origin and all other points are moved relative to this lower left corner to make it easier for cell identification. This is done by simply subtracting each data point with the lower-left corner of the data.

Step 4. Create and Flatten cells in row-major order: At this stage, we create and flatten cells so that all the cells corresponding to a single day are in row major order. This is done so that while assigning a cell ID, it is easier to identify neighboring cells using this ID.

Step 5. Map points to cells:

This process involves mapping each point to a cell ID. If a pickup occurred at a specific location on a specific day, the data point is mapped to the corresponding cell ID.

Step 6. Assign points to cells (Reduce operation):

In this stage, we perform a reduce operation by combining the value of a location (lat, lon) on the same day and the pickup counts are incremented. Since the reduce operation involves shuffling of data and sorting by key, this process takes the longest time. However, after this stage, we would be operating only on cell IDs rather than data points, so the amount of data that we would work on, is considerably less than the original dataset.

Step 7. Getis-ord score computation for each cell:

For each cell, we identify the neighboring cells and compute the G* statistic. We observed that the value of S (standard deviation) and mean (X-bar) remain the same for all cells. Hence, we pre-computed these values and used it for subsequent operations. The G* statistic was then calculated for each cell and stored in a hash map.

Step 8. Take top 50 hotspots:

Finally, we take the top 50 hotspots ordered by the z-score computed in the previous step. Though this step involves materialization of data, we take just the top 50 hotspots, so the amount of data that gets transferred is less.

1.3.5 Experimental results

As defined in the problem statement we obtained the top 50 values using the given dataset and have mapped the heat-map visualization from the obtained results.

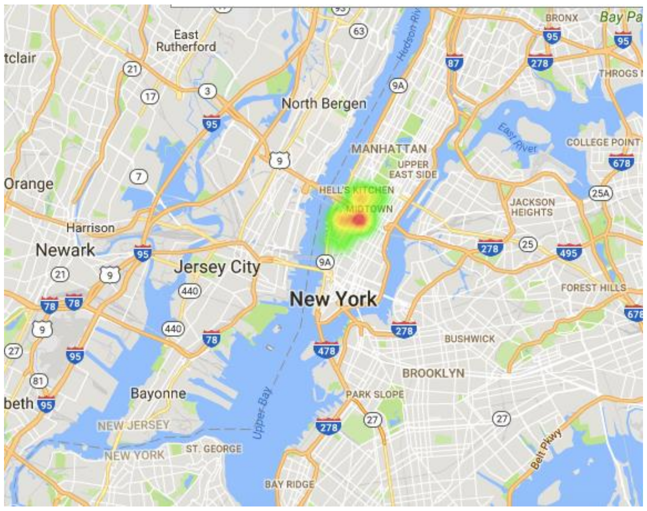


Figure 15: The figure shows the 50 top spots in a heat-map visualization.

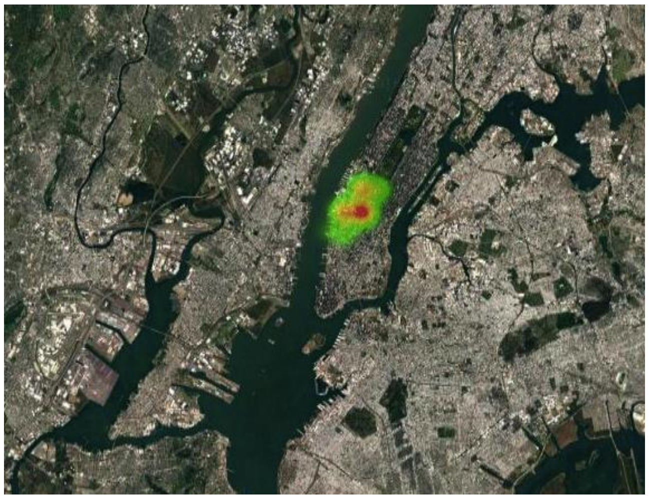


Figure 16: The figure shows the 50 top spots in a heat-map visualization in a satellite image.

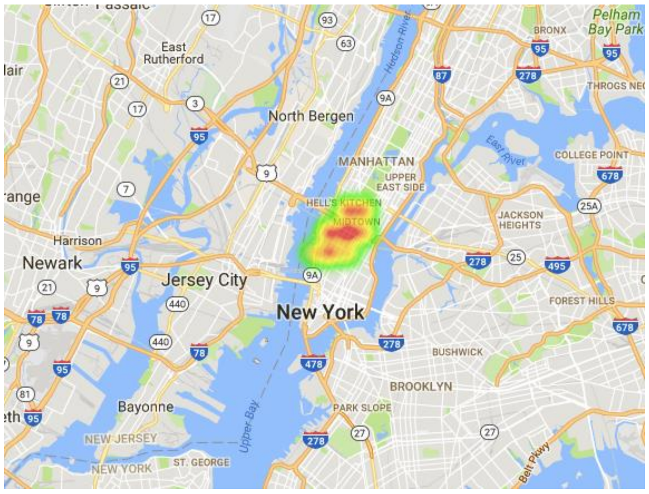


Figure 17: The figure shows the 150 top spots in a heat-map visualization.

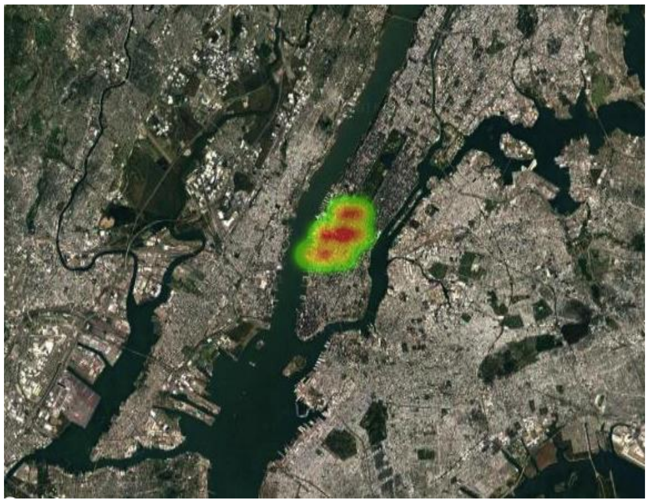


Figure 18: The figure shows the 150 top spots in a heat-map visualization in a satellite image.

2 Using Apache Spark for distributed computing

2.1 Introduction to Apache Spark

Architecture

Spark applications are run from the Spark driver program. The Spark driver is any program that has an associated SparkContext object. Once the spark process is started on a standalone machine or on the master in a cluster, the driver program can obtain the SparkContext in a compiled Scala program by using:

```
val sc = new
SparkContext(master="spark://master:7077")
```

or by running a spark shell through:

```
$ spark-shell --master spark://master:7077
```

Each machine in a Spark cluster have Executors associated with. Executors are JVM threads that are designed to operate on the RDDs associated with that node in the cluster.

The Spark driver program with the SparkContext is responsible for creation, execution of jobs by utilizing the DAGScheduler and Task Schedulers, which are responsible for spinning threads on worker nodes to execute the Spark Job.

Resilient Distributed Datasets (RDD)

RDDs or Resilient Distributed Datasets are the fundamental unit of data in the Spark environment. RDDs are built on top of the Scala collections framework [9] and they extend on them by adding distributed properties and concurrency to them.

RDDs are:

• Resilient — RDDs manage a lineage graph related to the transformations and actions performed on them. This makes them fault tolerant and ability to recover from failures. Also, RDDs can be replicated on nodes to increase availability and increase fault tolerance and reliability.

• Distributed — RDDs are distributed across nodes, i.e. partitioned across nodes in a cluster.

• Dataset — extended on Scala collections that represent lists, arrays and maps of primitive types and serializable objects.

In addition, RDDs are in-memory partitioned data structures. They exist solely in memory, spilling to disk optionally when memory is limited on the node.

RDDs are lazily evaluated and are cacheable. Spark's speed is due to the in-memory property and the lazy evaluations of RDDs.

RDDs are partitioned in memory, they exist across the cluster residing one per JVM. In addition, RDDs can use the native distributed filesystems like HDFS and utilize their partitioning and replication frameworks.

Operations on RDDs, DAGs, transformations and actions

RDDs have two fundamental operations:

— Transformations and,

— Actions

Transformations are operations that are evaluated lazily. Transformations convert the RDD from one form to another. For each transformation on a RDD, a new RDD object is created, and they are linked through a DAG as shown below:

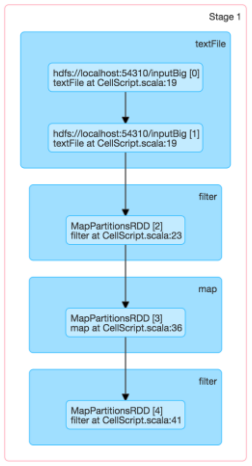

Stage 1
textFile
hdfs://localhost:54310/inputBig [0]
textFile at CellScript.scala:19
hdfs://localhost:54310/inputBig [1]
textFile at CellScript.scala:19
filter
MapPartitionsRDD [2]
filter at CellScript.scala:23
map
MapPartitionsRDD [3]
map at CellScript.scala:36
filter
MapPartitionsRDD [4]
filter at CellScript.scala:41

Figure 19

Any RDD stores a list of properties:

— List of parent RDDs from which the current RDD is generated from by using transformations

— Partition indices

— A function object that contains the operation to be done on the partitions

Actions are operations that trigger all operations above it(parent RDDs and computations) to finally execute. Actions are those operations that kick the lazy transformations into life and execute the whole DAG of tasks associated with a Spark job.

Examples of transformation are - `map, filter, flatten, flatMap, flatMapToPair, reduceByKey, join, cogroup` etc.

A few examples of actions are `count, collect and saveAsTextFile`.

DAGs

All spark operations are pushed into a Directed Acyclic Graph of tasks. The scheduler at the Driver program is responsible for this.

Example DAG –

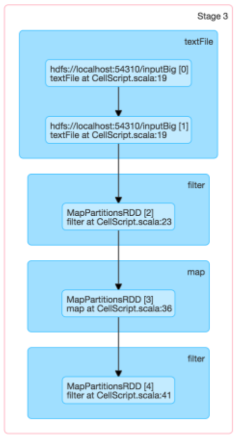

Stage 3
textFile
hdfs://localhost:54310/inputBig [0]
textFile at CellScript.scala:19
hdfs://localhost:54310/inputBig [1]
textFile at CellScript.scala:19
filter
MapPartitionsRDD [2]
filter at CellScript.scala:23
map
MapPartitionsRDD [3]
map at CellScript.scala:36
filter
MapPartitionsRDD [4]
filter at CellScript.scala:41

Figure 20

When an action is triggered the graph is traversed in reverse from the bottom-most RDD by seeking parents and finally executing them down the DAG till the action that triggered the evaluation of lazy transformation above it.

3 Note on Spark Performance

Stages

When an action is called on an RDD, a job is created for all operations till the action. The job is then transformed into stages submitted to the TaskScheduler for execution.

Stages are individual groups of operations that can be pipelined. For example, in our program, we have a bunch of maps and filters applied on our initial Hadoop data RDD(created using sc.textFile("hdfs://master:54310/input")). Then a reduceByKey is called on top of these operations. Further maps and filters are applied to get the Getis-ord statistic. The action that invokes the overall job is the takeOrdered(50) operation.

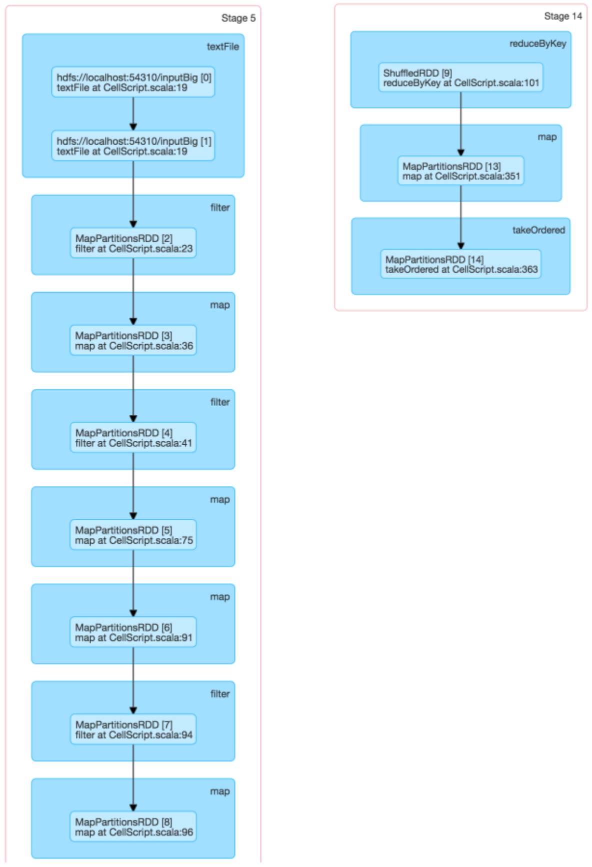

Stage 5
textFile
hdfs://localhost:54310/inputBig [0] textFile at CellScript.scala:19
hdfs://localhost:54310/inputBig [1] textFile at CellScript.scala:19
filter
MapPartitionsRDD [2] filter at CellScript.scala:23
map
MapPartitionsRDD [3] map at CellScript.scala:36
filter
MapPartitionsRDD [4] filter at CellScript.scala:41
map
MapPartitionsRDD [5] map at CellScript.scala:75
map
MapPartitionsRDD [6] map at CellScript.scala:91
filter
MapPartitionsRDD [7] filter at CellScript.scala:94
map
MapPartitionsRDD [8] map at CellScript.scala:96
Stage 14
reduceByKey
ShuffledRDD [9] reduceByKey at CellScript.scala:101
map
MapPartitionsRDD [13] map at CellScript.scala:351
takeOrdered
MapPartitionsRDD [14] takeOrdered at CellScript.scala:363

Figure 21

Here we see that the stages are split at the reduceByKey operation. The reason is that Spark splits stages based on shuffle operations. Each operation prior to the shuffle on reduceByKey operation can be pipelined across the RDD.

However the reduceByKey is a shuffle operation and hence has to be staged separately from the other operations.

Shuffles — Costliest operations

Shuffles are the costliest operations on Spark. Like hadoop-mapreduce, the shuffle stage involves sorting of data in some sort and this involves moving data across the network, involving high transfer costs. Also sorting is a O(nlogn) operation while maps, filters, reduce, are O(n) operations.

Hence to optimize a Spark job, one has to write operations in a way that reduces the number of shuffle operations as much as possible.

Other optimizations — serialization of objects

Another optimization that can be done with Spark is to reduce the overall memory footprint of objects and RDDs in the JVM. The idea is to use primitive types, Scala collections as much as possible as they are by default serialized by Spark and are optimized in RDD implementations [10]. In our application, we have reduced significant memory footprint compared to those implementations with GeoSpark by using Tuples of Integers and Longs for all our data-structures involving cells. Reducing memory footprint is important to avoid Spark from spilling into disk and hence incurring disk I/O costs.

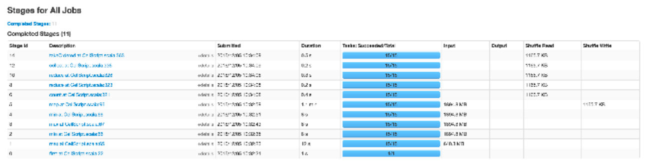
Operating on primitive data types by not making use of the objects, results in avoiding spillover of memory and helps in reducing number of bytes involved in the computation, whereas handled easily in case of primitive data types. So, sticking to primitive types helped us greatly reduce the run-time.

2. Reduce by key involves processing the shuffling data and combining them by incrementing number of pickup points occurring at the cell ID across the day. This step involves sorting data points present in all the RDD's, here network cost being the bottleneck during data access across the nodes, consumes most of the time during computation of the entire statistics. Hence, we have only one reduce by key operation across the entire program.

4. Performance Monitoring and Evaluation

Spark UI

The Spark UI on localhost:8080 is a useful tool to monitor the cluster spark resources, see scheduled jobs, see execution times and job failures.



Stages for All Jobs

Completed Stages: 11

Completed Stages (11)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
14	takeOrdered at CellScript.scala:363	[illegible]	0.5 s	15/15			1155.7 KB	
12	collect at CellScript.scala:356	[illegible]	0.2 s	15/15			1155.7 KB	
10	reduce at CellScript.scala:326	[illegible]	0.3 s	15/15			1155.7 KB	
8	reduce at CellScript.scala:323	[illegible]	0.2 s	15/15			1155.7 KB	
6	count at CellScript.scala:311	[illegible]	0.4 s	15/15			1155.7 KB	
5	map at CellScript.scala:96	[illegible]	1.1 min	15/15	1904.3 MB			1155.7 KB
4	map at CellScript.scala:[illegible]	[illegible]	8 s	15/15	1904.3 MB			
3	map at CellScript.scala:[illegible]	[illegible]	8 s	15/15	1904.3 MB			
2	map at CellScript.scala:[illegible]	[illegible]	5 s	15/15	1904.3 MB			
1	map at CellScript.scala:[illegible]	[illegible]	12 s	15/15	670.3 MB			
0	first at CellScript.scala:[illegible]	[illegible]	1 s	1/1				

Figure 22

We primarily leveraged the Spark UI to understand the execution framework, DAGs and Stages of RDDs to optimize on reducing the shuffle stage.

Ganglia monitoring

Notes on memory usage:

Compare VM vs normal — more vs less memory

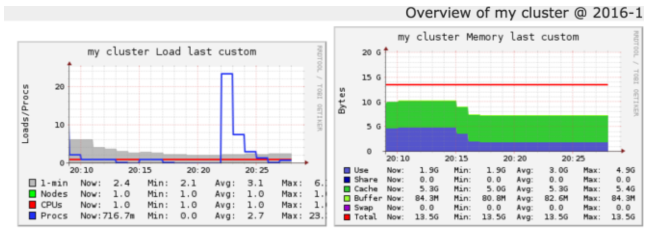


Figure 23

We see here that the memory usage is high initially and then there is a dip in the memory. This is due to the fact that the initially operations are performed on the point RDD which is a 1.9GB distributed RDD. Once we reduce the data by key and have cell RDD we see that the memory usage dips. This also correlates well with the network peak due to shuffle @t=20:15 in the network graph.

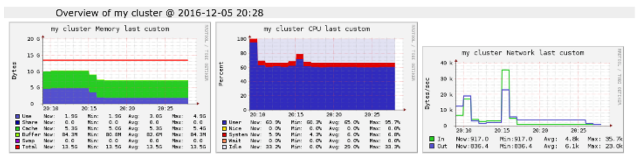
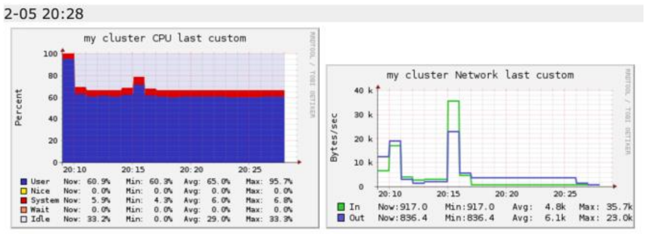


Figure 24

Notes on CPU usage



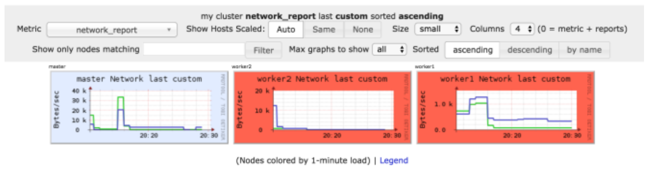
```
■ User   Now: 60.9%   Min: 60.3%   Avg: 65.0%   Max: 95.7%
□ Nice   Now:  0.0%   Min:  0.0%   Avg:  0.0%   Max:  0.0%
■ System Now:  5.9%   Min:  4.3%   Avg:  6.0%   Max:  6.8%
■ Wait   Now:  0.0%   Min:  0.0%   Avg:  0.0%   Max:  0.0%
□ Idle   Now: 33.2%   Min:  0.0%   Avg: 29.0%   Max: 33.3%
```

```
■ In   Now:917.0   Min:917.0   Avg:  4.8k   Max: 35.7k
■ Out  Now:836.4   Min:836.4   Avg:  6.1k   Max: 23.0k
```

Figure 25

It is seen that there is a spike in CPU usage right after t=20:15. This is due to the computationally intensive Getis-Ord function that computes with high-precision Double numbers.

Notes on network usage



my cluster **network_report** last **custom** sorted **ascending**

Metric: network_report — Show Hosts Scaled: Auto | Same | None — Size: small — Columns: 4 (0 = metric + reports)

Show only nodes matching — Filter — Max graphs to show: all — Sorted: ascending | descending | by name

(Nodes colored by 1-minute load) | Legend

Figure 26

The network usage peaks are due to the reduceByKey task. Since our application uses only one reduceByKey shuffle stage, there is the one peak in network activity during the whole execution.

5. CONCLUSION

In this project, we could implement spatial operations in RDDs in Spark in a distributed environment.

Phase1: Setting up Hadoop cluster and implementing various operations using the GeoSpark jar.

Phase2: Statistics gathering using indexing on the operations performed in Phase1.

Phase3: We performed Getis Ord Gi* score calculation on spatio-temporal data and implemented map reduce to obtain 50 hotspots of activity data.

Team Members and contribution

Group 25

1. Purushotham Kaushik Swaminathan
[pswamin1@asu.edu] – 1208626480: 20%
2. ARAVIND RAJENDRAN
[arajend6@asu.edu] – 1208684590: 20%
3. THAMIZH ARASAN RAJENDRAN
[trajendr@asu.edu]- 1209319666: 20%
4. Balaji Chandrasekaran
[bchandr6@asu.edu] – 1208948451: 20%
5. Suresh Gururajan Nolastname
[sgurura3@asu.edu] – 1208567798: 20%

REFERENCES

[1] Group 25 - DDS Project Phase 1, submission link: https://www.youtube.com/watch?v=ShfpMlbMFYo

[2] Ord, J.K. and A. Getis. 1995. "Local Spatial Autocorrelation Statistics: Distributional Issues and an Application" in Geographical Analysis 27(4).

[3] ACM SIGSPATIAL Cup 2016, Problem Definition (Online): http://sigspatial2016.sigspatial.org/giscup2016/problem Accessed: 12/04/2016

[4] Mehta, Paras, Christian Windolf, and Agnès Voisard. "Spatio-Temporal Hotspot Computation on Apache Spark (GIS Cup)."

[5] Jia Yu, Jinxuan Wu, Mohamed Sarwat. "GeoSpark: A Cluster Computing Framework for Processing Large-Scale Spatial Data". (short paper) In Proceeding of the ACM International Conference on Advances in Geographic Information Systems ACM SIGSPATIAL GIS 2015, Seattle, WA, USA November 2015

[6] PowerPoint poster templates for research poster presentations, (Online), accessed: 12/5/2016 at 00:14 AM http://www.posterpresentations.com/html/free_poster_templates.html

[7] The Scala Programming Language, https://www.scala-lang.org/

[8] Getis Ord Statistic for Geospatial operations: https://en.wikipedia.org/wiki/Geospatial_analysis

[9] Data Serialization (Online), accessed 12/05/2016, https://spark.apache.org/docs/latest/tuning.html#data-serialization

[10] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.