Identifying GeoSpatial Hotspots using Apache Spark

Group 15 Members:

Aditi Pothuganti
Karthik Srinivaasan
Kawshik Karur Rangaraju
Raghav Shakthidharan
Revanth Patil

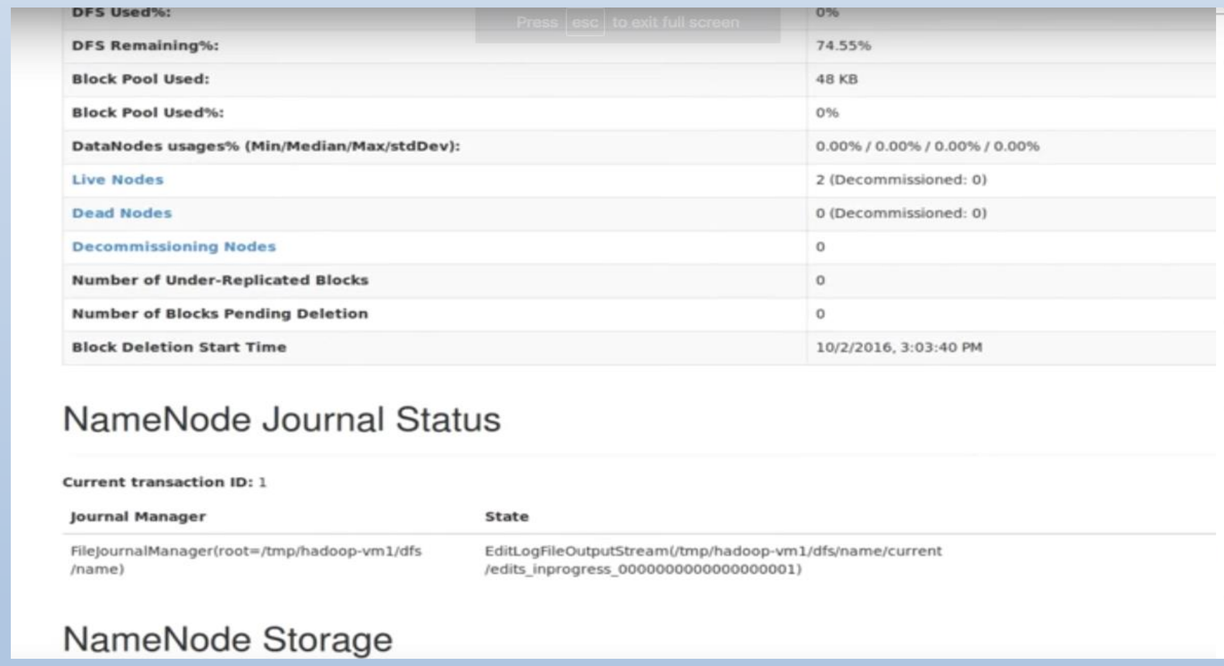
INTRODUCTION

To demonstrate the various geospatial operations that can be performed by GeoSpark, a spatio-temporal framework for distributed in-memory processing of large-scale spatial data. The performance of each geospatial operations is analyzed by using Ganglia monitoring tool. We also extend these operations from Geospark library to identify spatial hotspots from a given NYC Taxi trip dataset. The Getis-Ord statistic (G_i^*) used for identifying spatial patterns is calculated by using the Java API of Apache Spark.

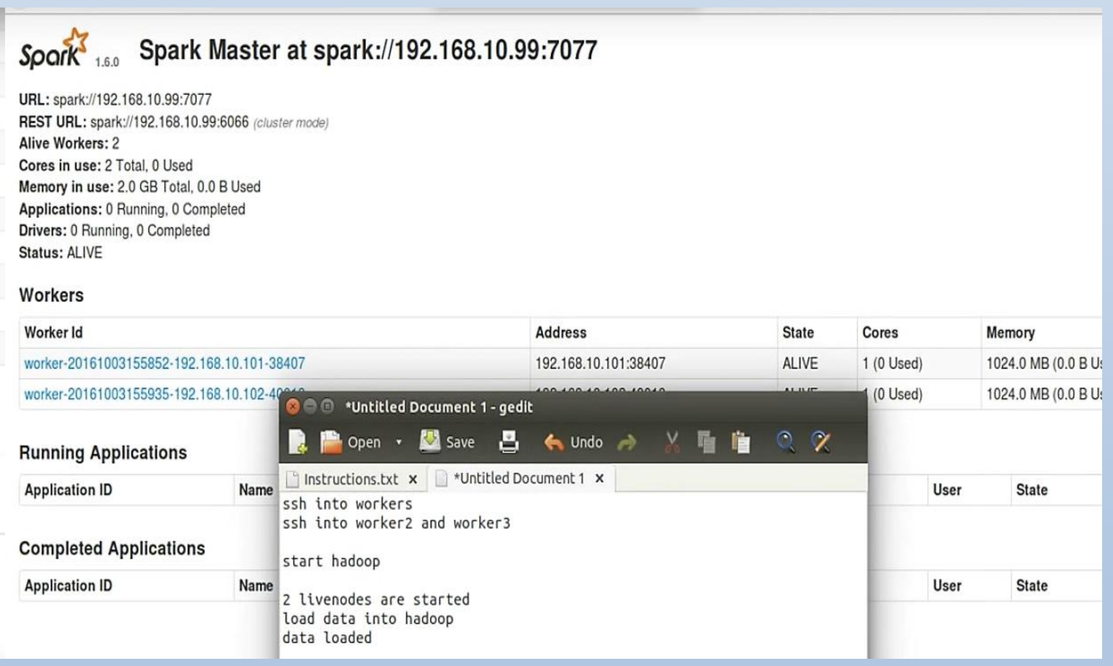
PHASE 1

Problem Statement: To load GeoSpark jar into Apache Spark and execute the following geospatial operations using Scala:

- Spatial Range Query with and without using an R-tree index.
- Spatial KNN Query with and without using an R-tree index.
- Spatial Join Query with and without using an R-tree index.



Hadoop Cluster



Spark Cluster

Result: The Hadoop and Spark cluster have been started with a master node and 2 worker nodes. The above queries have been implemented on the Spark cluster.

PHASE 2

Problem Statement: To implement a Spatial Join Query in Java using a Simple Cartesian product algorithm. Using the Ganglia monitoring tool, compare the CPU utilization, execution time and memory utilization of the geo-spatial operations implemented in Phase 1 with and without using indexing. To assess and differentiate between the performances of Spatial Join Query performed using a Cartesian product algorithm vs Spatial Join Query performed using equijoin- with and without indexing.

Results(comparison between different querrying algorithms)

1) Spatial Range Query without using R-tree index vs. Spatial Range Query with an R- tree index

Parameters (for 1000 iterations)	Spatial Range Query without R-tree Index	Spatial Range Query using R-tree Index
Execution Time	58 secs	51 secs
Average Memory	5.0 G	5.1 G
Average CPU usage (in %)	31.2	31.2

- Spatial Range Query without R-tree index requires lesser memory than Spatial Range Query with R-tree index. R-tree indexing takes more memory as it stores metadata of the indexes of each node in order to improve execution time and performance.
- Indexing enables faster searching and hence, Spatial Range Query without an R- tree index has a comparatively higher execution time than running the Spatial Range Query indexed with R-tree.

2) *KNN Query without using an R- tree index vs. KNN Query with R-tree index*

Parameters (for 1000 iterations)	Spatial KNN Query	Spatial KNN Query using R-tree Index
Execution Time	59 secs	51 secs
Average Memory	5.2G	5.3G
Average CPU usage (in %)	34.5	31

- KNN Query without an R- tree index has a comparatively higher execution time than running the KNN Query indexed with an R-tree.
- KNN Query without R-tree index requires lesser memory than KNN Query with R-tree index.

  The results are as expected.

3) Spatial Join Query on Equal Grid without an R-tree index vs. Spatial Join Query on Equal Grid with an R-tree index vs. Spatial Join Query on R-tree Grid without an R-tree index

Parameters (for 10 iterations)	Equal grid without R-Tree index	Equal grid with R-Tree index	R-Tree grid without R-Tree index
Execution Time	12mins 38 secs	8 mins 02 secs	4 mins 14 secs
Average Memory	4.7 G	4.9 G	4.8 G
Average CPU usage (in %)	34.5	36.5	36.3

- Spatial join query with equal grid partitioning is the same as querying with an RTree index on the PointRDD. The index speeds up the Spatial Join query as it would make an index nested loop join instead of a simple nested loop as in the case of querying without an index..
- Spatial Join Query with R-tree grid without R-tree has highest performance amongst the three.
- The R-tree grid is constructed hierarchically by grouping the leaf boxes into levels where querying is faster as it is a bottom-up approach whereas in Equal Grid, searching takes a longer time as it is Top-Down

4) *Spatial Join Query using EquiJoin vs Spatial Join Query using Cartesian product*

Parameters	Equal grid without R-Tree index(for 10 iterations)	Equal grid with R-Tree index(for 10 iterations)	R-Tree grid without R-Tree index (for 10 iterations)	Spatial Join Query (Cartesian Product)
Execution Time	12mins 38 secs	8 mins 02 secs	4 mins 14 secs	29 mins 42 secs
Average Memory	4.7 G	4.9 G	4.8 G	4.6 G
Average CPU usage (in %)	34.5	36.5	36.3	39.5

- Spatial Join Query using Cartesian product utilizes lesser memory as compared to Spatial Join Query using equijoin or innerjoin as it does undergo R-tree indexing and hence, does not store node indexes.
- However, Cartesian product takes more time to perform than innerjoin or equijoin as we reference all the data records present in both the tables rather than referencing only those data records whose join keys are equal. In other cases, the run time was faster since the rectangle RDD was efficiently partitioned into grids covering maximum points, rather than checking each point with each envelope.

PHASE 3

Problem Statement: In this phase, we have to find a list of 50 significant spatial hotspots in space and time by using the Getis-Ord statistic (G_i^*) from a given input dataset.

Algorithm:

Step 1: Load the dataset from HDFS using the JavaSparkContext function.

Step 2: From the given dataset, a list of envelopes each with edge size 0.01*0.01.

Step 3: We now create a Map containing the days as keys and a list of coordinates which contains all taxi pick up locations for a particular day.

Step 4: Now for every day ,for every envelope we iterate through our list of points and create a JavaPairRDD of key ->date and value another JavaPairRDD which contains an Envelope and a value . We count the number of points that fall into a particular envelope and is mapped to the RDD

JavaPairRDDs<Day, JavaPairRDD<Envelope, HashSet<Point>>>

Step 5: Then we create a JavaPairRDD with envelope as key and a list containing the total no of pickups from the envelope and its neighbours. Step 6: The Getis-Ord statistic (G_i^*) is calculated for each cell by using the formula

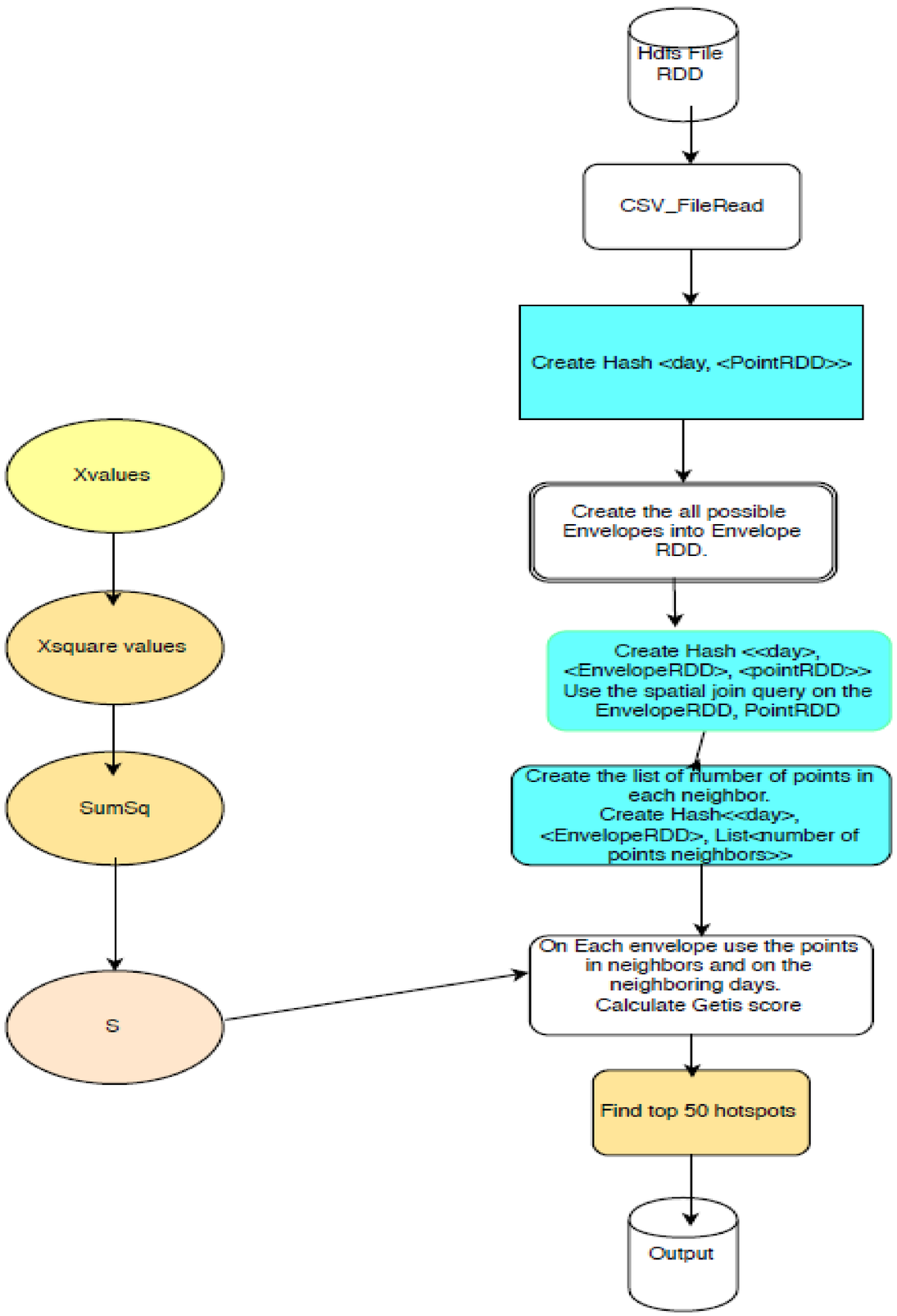
$$G_i^* = \frac{\sum_{j=1}^{n} w_{i,j}x_j - \bar{X}\sum_{j=1}^{n} w_{i,j}}{S\sqrt{\frac{\left[n\sum_{j=1}^{n} w_{i,j}^2 - \left(\sum_{j=1}^{n} w_{i,j}\right)^2\right]}{n-1}}}$$

where x_j is the attribute value for cell j.
$w_{i,j}$ is the spatial weight between cell i and j.
n is equal to the total number of cells

Step 6: Iterate through this and compute the GI score for every envelope for everyday and is put into a hashmap.

Step 7: The cells are sorted based on decreasing value of Getis-Ord statistic (G_i^*) from which the spatial hotspots can be obtained. These are the coordinates which serve as the pick-up location for maximum number of taxis.

FLOWCHART



Why Spark? Implementing geospatial operations on large datasets in a distributed environment using Hadoop MapReduce will reduce performance as it loads data from HDFS whereas Spark provides an efficient abstraction for in-memory cluster computing called Resilient Distributed Datasets (RDDs).

Leveraging runtime using Spark : The total run time of the algorithm for loading the big yellow taxi trip dataset to the HDFS, reading the data from HDFS and finally writing the Top 50 hotspots to the output file is very less (in few minutes). This is because we have used inbuilt spark functions like GroupByKey, ReduceByKey, maptopair, etc. all of which are considered to be code optimization techniques. We have used only three columns from the dataset: day, pickup latitude, pickup longitude and we are doing a spatial join query on RectangleRDD and PointRDD to get JavaPairRDD. Using these RDD representations makes use of Apache Spark's in-memory and parallel Map-Reduce paradigm.

Output: The top 50 spatial hotpots are obtained.