Geospatial Operations

Group 20

Anoop Sahoo, Bernard Ngabonziza, Jon Lammers, Madhu Talasila, Magarjuna Myla, Vageesh Bhasin

**ABSTRACT**

In this paper, we described the implementation of Geospatial operations namely Geometry Union, Geometry Convex Hull, Geometry Farthest Pair, Geometry Closest Pair, Spatial Range and Spatial Join using Apache Spark and Hadoop distributed framework. Also, we analyzed the performance in terms of memory, CPU Load and Network Utilization of our Geospatial operations on multi-node cluster using Ganglia by varying number of nodes. We extended the implementation of project to find number of required objects in a given space.

**Keywords**

HDFS, Spark, Distributed Database System, Geospatial

# INTRODUCTION

## Motivation

Geospatial data has explicit Geographic positioning information and huge amount of data. Efficient analysis of this data provides many useful features.

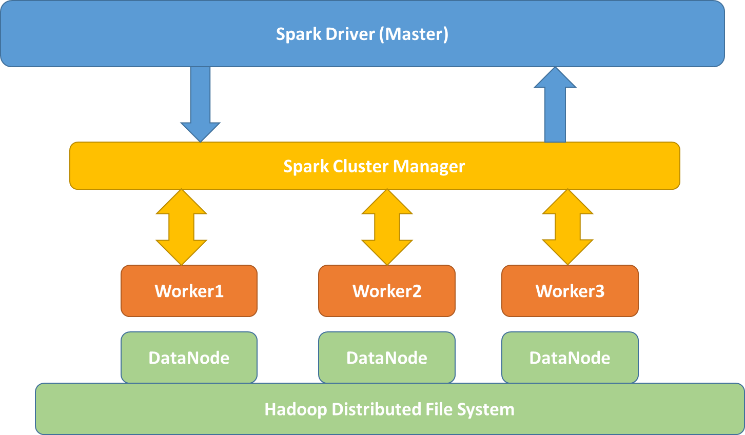
## Problem Statement

Centralized Systems and relational algebra often fails to give accurate results in limited time because of single point of failure, memory and scalability issues.

We made a distributed database system to which efficiently performs these functions. Our system uses Apache Spark for distributed in-memory processing of the functions and Hadoop Distributed File System (HDFS) to provide scalable distributed storage.

# SYSTEM ARCHITECTURE

Architecture of the project that we developed. We created a three node cluster with one master and three workers.



**Figure 1. System Architecture**

## HDFS and DataNode

“HDFS has master-slave architecture. HDFS cluster consists of single NameNode, a master that manages file system namespace and regulates access to files. There are three DataNodes one per node in the cluster which manages storage attached to the nodes they run on” [1]. In our project we used HDFS to store data. Application reads data from spark. After performing map-reduce using spark, data is stored into output file in Hadoop.

## Spark

Spark has an advanced DAG (Directed Acyclic Graph) execution engine. Spark supports in-memory cluster computing. Supports Generic programming model. One can quickly write applications in Java, Scala or Python. Spark has a feature of Run everywhere. Sparks increases the speed of the application by its in memory map reduce computations.

## Worker

Spark sends data to its workers i.e., partitions and partitions perform map reduce operations on the worker nodes and send combined data to one node. That node finally saves data to Hadoop.

## Spark Cluster Manager

We used Standalone cluster manager to manager our nodes in spark.

## Geospatial Application

The seven implemented functions are independent, except the Geometry Farthest Pair is dependent on the algorithm in the Geometry Convex Hull function to calculate the upper and lower hulls and Spatial Aggregation which is dependent on Spatial Join function, reads the output of Spatial Join and calculates the number of points/ rectangles that lie inside the given query rectangles.

Several classes were created that were used across the seven functions implemented in this package:

***GeoPoint*** – Implementation of a basic X, Y point class. This class provides the ability to manage points and calculated the distance between two points, the slope between two points, if the point lies within a given rectangle, and if two points are geometrically equal.

***GeoPointPair –*** This builds on the GeoPoint class to provide a pair of points.

***Rectangle –*** Similar to GeoPointPair, this function provides implementation of a rectangle and functions for containsRectangle and containsPoint.

***GeoSpatialUtils –*** This class provides a function that removes an hdfs files. This is useful for running tests on a system to delete the output file before trying to write to it.

***HullResult –*** This class contains two RDD’s. It is used as an intermediate value in the Geometry Convex Hull algorithm prior to writing the results to the required file. This is useful as the implementation of the Geometry Farthest Pair function uses upper and lower hulls from Convex Hull. With this intermediate results, Farthest Pair Implementation saves time by not writing and then reading the results from a file.

# IMPLEMENTATION

## Pre-requisites

We used virtual machines to create node of a cluster. Each node contains the following software

1. Operating System: Ubuntu 14.04.1 LTS 32-bit
2. OpenJDK 1.7.0
3. Hadoop 2.6.0
4. Spark 1.2.0
5. SSH
6. Maven2
7. Eclipse Luna
8. Ganglia

## System Configuration

Configuration of our system is as follows

1. **Master**

Processor: 4 cores

Memory: 3GB

HD: 30GB

Network Adapter: NAT

1. **Worker1**

Processor: 4 cores

Memory: 2GB

HD: 30GB

Network Adapter: NAT

1. **Worker2**

Processor: 4 cores

Memory: 2GB

HD: 30GB

Network Adapter: NAT

1. **Worker3**

Processor: 4 cores

Memory: 2GB

HD: 30GB

Network Adapter: NAT

1. **Virtual Machine**

Virtual Player

1. **Host Machine**

Processor: 4 cores

Memory: 16GB

HD: 1TB + 128GB SSD

## Geospatial Operations

### **Geometry union**

**Definition:** The union of a set S of polygons is the set of all points that lie in at least one of the polygons in S, where only the perimeter of all points is kept while the inner segments are removed.

Example: The Figures 2 and 3 show the input and output of a union operation. The input is a set of polygons and the output is the perimeter of the area which is composed of these polygons.

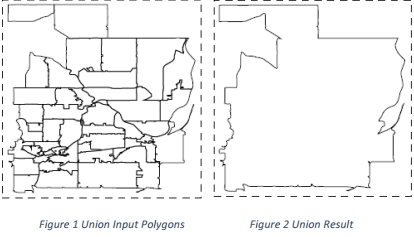


Figure 1. (a) Union input polygons (b) Union result

**Function Name**: GeometryUnion

**Arguments**:

(1) String InputLocation: the location of the input in HDFS

(2) String OutputLocation: the location of the output in HDFS

**Requirement**: Load a set of polygons, output the union result of this set.

**Input Dataset Schema:**

x1, y1, x2, y2: Every row is a point (longitude, latitude) which defines a rectangle.

**Output Dataset Schema:**

x, y: Every row is a point (longitude, latitude). This dataset has a bunch of points. The result polygon is composed of all these points.

**Algorithm:**

The algorithm, PolygonUnion[2], is based from a research paper by Avraham Margalit and Gary D. Knott. The initial algorithm works on oriented polygons, i.e., polygons having edges with orientation. However, our input data does not fit this criteria, and hence we will apply the generalized algorithm. We start by finding the intersection points between the polygons.

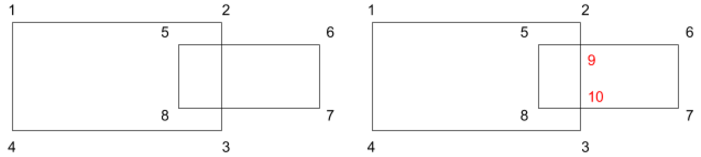


Figure 2. (a) Initial Overlapping polygons, (b) Intersection points

We then classify the edge fragments to be either inside, outside or on the boundary of the polygons. An edge fragment is classified inside if at least one of its endpoints is an inside vertex, or the two endpoints are boundary vertices and all the other points on the edge are inside points. An edge fragment is classified outside if atleast one of its endpoints is an outside vertex, or the two endpoints are boundary vertices and all the other points on the edge are outside points. After classification of the edge fragments, we select the fragments based on the operation. In case of Geometry Union, we select all the edge fragments that have been classified as outside. Finally, we construct the result polygon using the selected edge fragments.

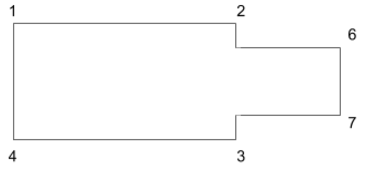


Figure 3. Final resultant polygon

The above algorithm works well for small sets of polygons, however using this on a larger set would not be an optimal solution. Another algorithm, called CascadingUnion[3], is a modification of this algorithm for faster processing. It breaks the input into smaller, manageable sets of polygon, and distributes the union computing across these subsets. It then takes the result of this operation, and recursively performs the union operation on intermediate results until it finds the result polygon.

**Pseudocode:**

1. Load file InputLocation and read data into RDD
2. Map the data into different partitions, using mapPartitions()
   1. Pass in the PolygonUnion() function to perform the Union operation on the partitioned set of polygons
   2. Save the results into an intermediate RDD
3. Gather the intermediate result polygons from the RDD using repartition()
4. Map the data again and perform a final union
   1. Pass the PolygonUnion() function again to get the union of the intermediates polygons, to form the final result polygon
   2. Save the result polygon into a file, at OutputLocation, on the HDFS
   3. Return true
5. Handle exceptions, by logging error and returning false

**How to run:**

./spark-submit \

--class edu.asu.cse512.Union \

--master <Spark Master IP> \

--jars convexHull-0.1.jar,jts-1.13.jar \

union-0.1.jar \

hdfs://<HDFS Master IP>/inputFile \

hdfs://<HDFS Master IP>/outputFile

### **Geometry convex hull**

**Definition**: The convex hull of a set of points P is the smallest convex polygon that contains all points in P. The output of the convex hull operation is the points forming the convex hull ordered in a clockwise direction.

Example: In Figures 4 and 5 show the input and output of a convex hull operation. The input is a set of points and the output is the points which compose the convex hull.

**Function Name:** GeometryConvexHull

**Arguments:**

(1) String InputLocation: the location of the input in HDFS

(2) String OutputLocation: the location of the output in HDFS

**Return Value:** Boolean: If the function works well, return True otherwise return False.

**Requirement:** Load a set of points, output the convex hull of this set.

**Input Dataset Schema:**

x, y: Every row is a point (longitude, latitude). This dataset has a bunch of points.

**Output Dataset Schema:**

x, y: Every row is a point (longitude, latitude). This dataset has a bunch of points. A convex hull is composed of all these points.

**Algorithm/ Pseudocode:**

The algorithm was developed use Apache Spark based on the convexHull algorithm in the papers by Andrew**Error! Reference source not found.**.

The implementation provided for two layers of interfaces. The outer layer is

1. Sort the points by X, then Y in ascending order
2. Until no points are removed,
   1. Add first point to upperHull
   2. If pnt1 to pnt2 is vertical and pnt1 to pnt3 then
      1. Add pnt2 to upperHull
      2. Replace pnt1 with pnt2
   3. If slope of pnt1 to pnt2 is greater than slope of pnt1 to pnt3
      1. Add pnt2 to upperHull
      2. Replace pnt1 with pnt2
   4. Replace pnt2 with pnt3, get new pnt3
3. Sort the points by X, then Y in descending order
4. Until no points are removed,
   1. Add first point to lowerHull
   2. If pnt1 to pnt2 is vertical and pnt1 to pnt3 then
      1. Add pnt2 to lowerHull
      2. Replace pnt1 with pnt2
   3. If slope of pnt1 to pnt2 is greater than slope of pnt1 to pnt3
      1. Add pnt2 to lowerHull
      2. Replace pnt1 with pnt2
   4. Replace pnt2 with pnt3, get new pnt3
5. Combine upper and lower hulls
6. Remove non-unique points
7. Sort the results

**How to run:**

./spark-submit \

--class edu.asu.cse512.convexHull \

--master <Spark Master IP> \

--jars convexHull-0.1.jar \

hdfs://<HDFS Master IP>/inputFile \

hdfs://<HDFS Master IP>/outputFile

### **Geometry farthest pair**

**Definition:** The farthest pair is a set of points P1 and P2 which are most far apart from each other in the given input set of points in 2D Euclidean space. The output of the farthest pair is two points written to file in sorted order one line per point sorted by X coordinate and then by Y coordinate.

Example: Figure 4 shows the input points and Figure 5 shows the output of a farthest point operation.



Figure 4 Input points



Figure 5. Output of Closest, Farthest & Convex Hull

**Function Name:** FarthestPair.operation()

**Arguments:**

(1) String InputLocation: the location of the input in HDFS

(2) String OutputLocation: the location of the output in HDFS

**Requirement:** Load a set of points, output the convex hull of this set.

**Input Dataset Schema:**

x, y: Every row is a point (longitude, latitude). This dataset has a bunch of points.

**Output Dataset Schema:**

x, y: Every row is a point (longitude, latitude). This dataset has a two point’s one in each row in sorted order.

**Algorithm/Pseudocode:**

The algorithm is developed on the top of convex Hull algorithm. As we know the farthest points should reside on the convex Hull we can leverage that and compute only pair wise distance between points returned by the convex Hull operation.

Convex Hull implementation is written in such a way that it saves the temporary RDD and results it which can be used in Farthest Pair operation so that we can eliminate file writing and reading from final convex Hull output.

To compute the farthest points we can use brute-force approach by calculating the pairwise Euclidean distance between all points in P let say there are n such points. This will give a time complexity of O(n2). As distance between (*x*1, *y*1) and (*x*2, *y*2) is same as (*x*2, *y*2) and (*x*1, *y*1) calculating it once is sufficient so we can reduce the number of loops by half. Even then the time complexity stills remains same because O(n2/2) reduces to O(n2).

Instead of calculating the pairwise distances between all the points we can reduce the problem space to number of points that lie on the convex hull because the farthest points should always lie on the convex. We can compute convex hull in O(nlogn) using Andrew’s Monotone Chain[6] convex hull algorithm which is discussed above. We can apply brute force approach on these points which will be faster.

**Steps**:

1. Load the input data set using *SparkContext.textFile()* and store in inputRDD

2. Pass inputRDD to Convex Hull algorithm which is explained above and store the results in convexHullPointsRDD

3. Create a cartesian product using sparks cartesian method of all the points on convex Hull RDD.

4. Using map() on the cartesian product points compute distances between points.

5. Now we have distance along with points as key value pair.

6. Combine all the pairs obtained using *reduce()* method and get the most distant pair.

7. Sort the two points obtained from the reducer

8. Save the output points into a outputpath file in HDFS using *saveAsTextFilePath()*.

**How to run:**

./spark-submit \

--class edu.asu.cse512.FarthestPair \

--master <Spark Master IP> \

--jars convexHull-0.1.jar \

farthestPair-0.1.jar hdfs://<HDFS Master IP>/inputFile hdfs://<HDFS Master IP>/outputFile

### **Geometry closest pair**

**Definition:** Given a set of points P, the closest pair is the pair of points that have the smallest Euclidean distance between them.

Example: Figure 5 shows the two points contributing to the closest pair have to lie on the convex hull. The input of the farthest pair operation is a set of points and the output is a pair of points which have the closest distances between each other.

**Function Name:** GeometryClosestPair

**Arguments:**

(1) String InputLocation: the location of the input in HDFS

(2) String OutputLocation: the location of the output in HDFS

**Requirement:** Load a set of points, output the closest pair of this set.

**Input Dataset Schema:**

x, y: Every row is a point (latitude, longitude).

**Output Dataset Schema:**

x, y: Every row is a point (latitude, longitude). This dataset has two points. The closest pair is composed of them.

**Algorithm:**

The algorithm uses a Divide-and-Conquer technique, where each dataset is divided and we recursively find the closest pair in the divided data set. Initially, we sort the dataset w.r.t x and y coordinates, and create two different datasets. We run the recursive algorithm on these two datasets. For the recursion, we define a base case and a recursive call. The base case stops the recursive calls when number of points in the list are less than or equal to 3. For the recursion, we find the median and split the datasets, which are then sent as inputs for the recursion call.

**Pseudocode**:

1. Load file InputLocation and read data into RDD
2. Map the data into different partitions, using mapPartitions()
   1. Pass in the LocalClosestPair() function to perform the Closest Pair operation on the partitioned set of polygons
   2. Save the results into an intermediate RDD
3. Gather the intermediate result closest pairs from the RDD using repartition()
4. Map the data again and perform a final closest pair
   1. Pass the FinalClosestPair() function again to get the closest pairs from the intermediate pairs
   2. Save the resultant pair into a file, at OutputLocation, on the HDFS
   3. Return true
5. Handle exceptions, by logging error and returning false

### **Spatial range query**

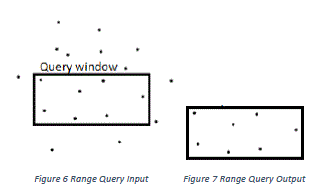
Definition: Spatial Range Queries are used to inquire about certain spatial objects which lie inside a certain query window.

Figure . Spatial range query

**Function Name:** SpatialRangeQuery

**Arguments:**

1. String InputLocation1: location of the input1 in HDFS
2. String InputLocation2: location of the input2 in HDFS
3. String OutputLocation: location of the output in HDFS

**Requirement:** Load a set of polygons, output the query result of this set.

**Input1 Dataset Schema:**

id, x1, y1: Every row represents one point (longitude, latitude).This set has a bunch of points.

**Input2 Dataset Schema**:

x1, y1,x2, y2: This dataset has a pair of points (longitude, latitude) which defines a rectangle. This dataset is the query window of range query.

**Output Dataset Schema:**

id: Every row is the id of a point. This is a set has a bunch of ids.

**Algorithm/ Pseudocode:**

1. Creates a spark configuration
2. Creates java spark context using spark configuration
3. Reads the inputfile1 from Hadoop and creates JavaRDD of type GeoPoint(class defined to map points)
4. Reads the inputfile2 from Hadoop and creates JavaRDD

of type Rectangle (class defined to map points to create a rectangle)

1. Creates a broadcast variable from the rectangle i.e. the query window created in step 4.
2. Now each GeoPoint is checked against the broadcasted rectangle to verify whether it is contained in that or not.
3. Result is combined using coalesce function and output is saved to Hadoop

**How to run:**

./spark-submit \

--class edu.asu.cse512.RangeQuery \

--master <Spark Master IP> \

--jars convexHull-0.1.jar \

rangeQuery-0.1.jar hdfs://<HDFS Master IP>/inputFile1 \

hdfs://<HDFS Master IP>/inputFile2

hdfs://<HDFS Master IP>/outputFile

### **Spatial join query**

Definition: Spatial join operation is used to combine two or more datasets with respect to a spatial predicate. Given a set of rivers and set of locations, spatial join returns all the rivers that lie inside or on the particular location.

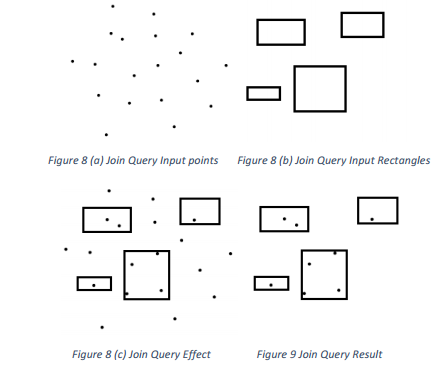


Figure . Spatial join query

**Function Name:** SpatialJoinQuery

**Arguments:**

1. String InputLocation1: location of the input1 in HDFS
2. String InputLocation2: location of the input2 in HDFS
3. String OutputLocation: location of the output in HDFS
4. String input1Type: indicates input1 is point or rectangle.

**Requirement:** Load two sets of polygons, output the join query result of these sets.

**Input1 Dataset Schema:**

1. A-id, x1, y1, x2, y2: Every row is a pair of points which defines a rectangle. This set has bunch of rectangles. Both contained and overlapped are considered.

(or)

1. A-id, x1, y1: Every row defines a point. This set has bunch of points. Points on and inside the rectangle are considered.

**Input2 Dataset Schema**:

B-id, x1, y1, x2, y2: Every row is a pair of points which defines a rectangle. This set has bunch of rectangles.

**Output Dataset Schema:**

B-id, A-id1, A-id2… A-idn: Every row has one B-id and 0~ n A-id. This dataset has bunch of rows.

**Algorithm/ Pseudocode:**

1. Creates a spark configuration
2. Creates java spark context using spark configuration
3. Reads the inputfile2 from Hadoop and creates JavaRDD of type Rectangle (class defined to map points to create a rectangle). Rectangle Definition: Bottom Left, Top Right.
4. Sorts the inputfile2 rectangles by overriding the default sortBy function
5. If input type is “point” then
   1. Reads the inputfile1 from Hadoop and creates JavaRDD of type GeoPoint (class defines to map points)
   2. Creates of broadcast variable of list of GeoPoint
   3. Then the inputfile2 JavaRDD is partitioned to different nodes in cluster and inputfile1 is broadcasted.
   4. On each node: For every rectangle in inputfile2 JavaRDD, GeoPoint Ids in broadcasted points are taken.
   5. Code to check if Rectangle contains GeoPoint:

this.x >= input2Rectangle.getX1() &&

this.x <= input2Rectangle.getX2() &&

this.y >= input2Rectangle.getY1() &&

this.y <= input2Rectangle.getY2()

* 1. GeoPoint Ids are sorted.
  2. String of inputfile2 Rectangle id and set of inputfile1 GeoPoint ids are returned as a result from mapped partitions.
  3. Result is combined using coalesce function and output is saved to Hadoop.

1. If input type is “rectangle” the
   1. Reads the inputfile1 from Hadoop and creates JavaRDD of type GeoPoint (class defines to map points)
   2. Creates of broadcast variable of list of Rectangles
   3. Then the inputfile2 JavaRDD is partitioned to different nodes in cluster and inputfile1 is broadcasted.
   4. On each node: For every rectangle in inputfile2 JavaRDD, Rectangle Ids in broadcasted rectangles are taken.
   5. Code to check if Rectangle contains Rectangle:

this.x1 <= rectangle.x2 &&

this.y1 <= rectangle.y2 &&

this.x1 >= rectangle.x1 &&

this.y1 >= rectangle.y1 &&

this.x2 <= rectangle.x2 &&

this.y2 <= rectangle.y2 &&

this.x2 >= rectangle.x1 &&

this.y2 >= rectangle.y1

* 1. Rectangle Ids in broadcasted rectangles are sorted.
  2. String of inputfile2 Rectangle id and set of inputfile1 Rectangle ids are returned as a result from mapped partitions.
  3. Result is combined using coalesce function and output is saved to Hadoop.

**How to run:**

1. If input type is “rectangle”

./spark-submit \

--class edu.asu.cse512.Join \

--master <Spark Master IP> \

--jars convexHull-0.1.jar \

joinQuery-0.1.jar hdfs://<HDFS Master IP>/inputFile1 \

hdfs://<HDFS Master IP>/inputFile2 \

hdfs://<HDFS Master IP>/outputFile rectangle

1. If input type is point

./spark-submit \

--class edu.asu.cse512.Join \

--master <Spark Master IP> \

--jars convexHull-0.1.jar \

joinQuery-0.1.jar hdfs://<HDFS Master IP>/inputFile1 \

hdfs://<HDFS Master IP>/inputFile2 \

hdfs://<HDFS Master IP>/outputFile point

### **Spatial aggregation query**

**Function Name:** SpatialAggregationQuery

**Arguments:**

1. String InputLocation1: location of the input1 in HDFS
2. String InputLocation2: location of the input2 in HDFS
3. String OutputLocation: location of the output in HDFS
4. String input1Type: indicates input1 is point or rectangle.

**Requirement:** Load two sets of polygons, output the join query result of these sets.

**Input1 Dataset Schema:**

1. A-id, x1, y1, x2, y2: Every row is a pair of points which defines a rectangle. This set has bunch of rectangles. Both contained and overlapped are considered.

(or)

1. A-id, x1, y1: Every row defines a point. This set has bunch of points. Points on and inside the rectangle are considered.

**Input2 Dataset Schema**:

B-id, x1, y1, x2, y2: Every row is a pair of points which defines a rectangle. This set has bunch of rectangles.

**Output Dataset Schema:**

B-id, count: Every row is a pair which represents the Rectangle ID and count represents the number of points lie in that rectangle.

**Algorithm/ Pseudocode:**

1. Creates a spark configuration
2. Creates java spark context using spark configuration
3. Invoked SpatialJoinQuery which is explained in detail above
4. The output files generated from step 3 are read using *context.textFile().*
5. Each line in each file is converted to a Iterable JavaRDD<String> using the flatMap().
6. Step 5 JavaRDD Iterable is converted to JavaPairRDD using *maptToPair()* by split each line into key value pair where key is the rectangle id and the value is count of array obtained by splitting line by comma and subtracting 1 which gives total number of points in the rectangle.
7. The obtained key value pairs in JavaPairRDD are reduced by key using *reduceByKey()* which counts the total count for the same rectangle id.
8. The results are then sorted by key using *sortbyKey()*.
9. The fineal JavaPairRDD is then converted back to JavaRDD and key value are combined to string using a comma as separator.
10. Result is combined using coalesce function and output is saved to Hadoop.

**How to run:**

./spark-submit \

--class edu.asu.cse512.aggregation \

--master <Spark Master IP> \

--jars convexHull-0.1.jar,joinQuery-0.1.jar \

spatialAggregation-0.1.jar \

hdfs://<HDFS Master IP>/inputFile1 \

hdfs://<HDFS Master IP>/inputFile2 \

hdfs://<HDFS Master IP>/outputFile rectangle

# EXPERIMENTAL SETUP

Our experiments were run on a single laptop with an Intel Core I7 processor and 16GB of RAM as a host machine. The master and slave computer were all run on this host machine, in Ubuntu Linux virtual machines. Network communications was between these virtual machines but completely within the host machine.

# EXPERIMENTAL EVALUATION

The following sections detail the experimental evaluation of the original six functions and the added Geospatial Aggregation function.

The displays of the Spark Status screens and the Ganglia screens are moved to Section 0 -

Addendum – Run Time Figures as they required full width to display properly.

### **Geometry union**

**Run Time:** As seen from Figure 8, the run-time of the functions increases as we increase the number of nodes in the cluster. This is an expected behavior for the cluster configuration, multiple low-resourced virtual machines on a single host. If the cluster comprised of multiple high-resourced physical machines, we would have seen a drop in run time, until the network overhead grows above the increase in computing power.

Figure . Geometry union run time

**CPU Used:** As seen from Figure 9, the CPU utilization of the master node remains constant when we increase the number of nodes in the cluster. However, the sharp drop in percentage of CPU cycles used in a slave (worker) node directly reflects that the total work was chunked into smaller work assignments for the slave nodes.

Figure . Geometry union CPU used (% Cycles spent in user mode v/s # of nodes)

**Memory Used:** As seen from Figure 10, it is quite evident that the memory utilized by the master increases (slightly) as it has to perform more overhead processes to keep track of distributed work. Another interesting point to note is that the master is utilizing its allocated memory completely. Thus suggesting that the master node has to be high-resourced when compared to the slave nodes. For the slave (worker) nodes, there is hardly any change in memory utilized, possibly due to the limited test data set.

Figure . Geometry union memory used

**Network Traffic:** As the number of nodes in a cluster increases, we see an increase in the network traffic. This is an expected behavior, since the master has to send data to more number of slave (worker) nodes, and the slave nodes have to send the status and/or output back to the master node. If you notice the graph, there is an anomaly when number of nodes is 3. This is not because the network traffic was less for this configuration, but because the data monitored had a drop in data point. Figure 11 can explain this anomaly better.

Figure . Geometry union network traffic

The data that is shown in Figure 12 is analyzed base on the readings from ganglia and by further analyzing this data above graphs are generated.

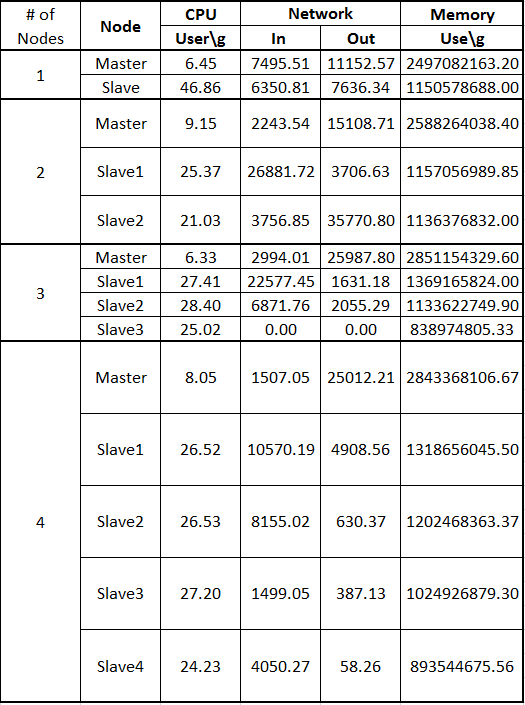


Figure . Geometry union raw data

### **Geometry convex hull**

Figure 13. Convex hull run time

As expected, the run-time of the function increases as the number of nodes in the cluster increases as seen in Figure 133. This is due to the nature of the experimental system which consists of multiple virtual machines running on a single host system. If the experiment were run in a cluster of multiple physical machines, then a decrease of elapsed time would be expected as the cluster size increases until the communications overhead overwhelms the advantage of increasing processing power.

Figure 14. Convex hull CPU Used

As shown in Figure 14, the master node uses about the same amount of CPU as the number of nodes increases. There is a very small drop-off. This is likely due to the increased run-time. The master node will perform the same tasks over a longer period, lowering the average CPU load.

Figure 15 shows the memory used on the master and the average of the slave nodes. The master shows a small increase as the number of slaves increases. This is likely due to the need to keep track of the work from these slave nodes. The slaves show almost no change in memory usage as the number of nodes increases. One possible explanation is the limited size of the experimental data set.

Figure 15. Convex hull memory used

Figure 16. Convex Hull Network Bytes Sent

The increase in network traffic shown in Figure 16 as more nodes are added to the cluster is expected, as more coordination between the master and slave nodes to perform the algorithm is required as the cluster gets more complicated.

Figure 17 shows the raw results of running the Geometry Convex Hull algorithm on the class provided point data set. One apparent anomaly in the raw data is the last line, showing more than three times the network traffic of any other item, including five times as much output as input. This somewhat skews the remaining results.

This Figure 17 also illustrates a possible anomaly seen in Figure 16. The network bytes sent shows a consistent increase as the number of nodes increases. The increase with four slave nodes is dramatically more than the rest. Looking at the raw results in Figure 17, the last node in each cluster shows the highest network output values of any node.

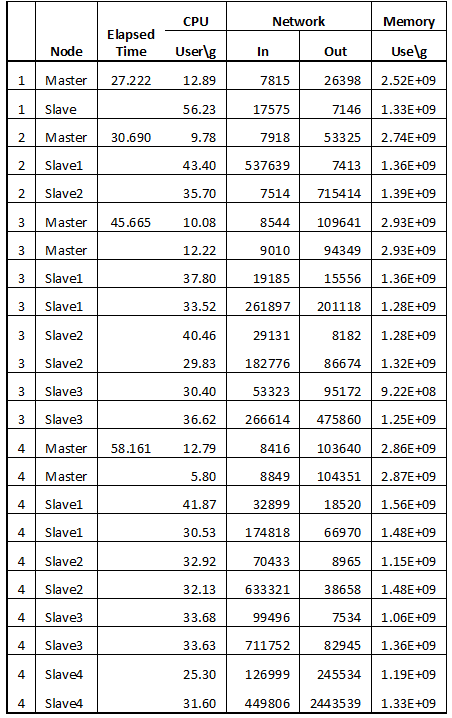


Figure convex hull raw data

### **Geometry farthest pair**

As farthest pair internally uses convex hull to compute farthest pairs the implementation analysis to convex hull should able applicable here and same reasoning hold for each of them.

Figure 18. Farthest pair run time

As expected, the run-time of the function increases as the number of nodes in the cluster increases as seen in Figure 138. This is due to the nature of the experimental system which consists of multiple virtual machines running on a single host system. If the experiment were run in a cluster of multiple physical machines, then a decrease of elapsed time would be expected as the cluster size increases until the communications overhead overwhelms the advantage of increasing processing power.

Figure 19. Farthest pair CPU used

As shown in Figure 19, the master node uses about the same amount of CPU as the number of nodes increases. There is a very small drop-off. This is likely due to the increased run-time. The master node will perform the same tasks over a longer period, lowering the average CPU load.

Figure 20. Farthest pair memory used

Figure 20 shows the memory used on the master and the average of the slave nodes. The master shows a small increase as the number of slaves increases. This is likely due to the need to keep track of the work from these slave nodes. The slaves show almost no change in memory usage as the number of nodes increases. One possible explanation is the limited size of the experimental data set.

Figure 21. Farthest pair network bytes sent

The increase in network traffic shown in Figure 21 as more nodes are added to the cluster is expected, as more coordination between the master and slave nodes to perform the algorithm is required as the cluster gets more complicated.

Figure 22 shows the raw results of running the Geometry Convex Hull algorithm on the class provided point data set. One apparent anomaly in the raw data is the last line, showing more than three times the network traffic of any other item, including five times as much output as input. This somewhat skews the remaining results.

This table also illustrates a possible anomaly seen in Figure 21. The network bytes sent shows a consistent increase as the number of nodes increases. The increase with four slave nodes is dramatically more than the rest. Looking at the raw results in Figure 22, the last node in each cluster shows the highest network output values of any node. This may just be the

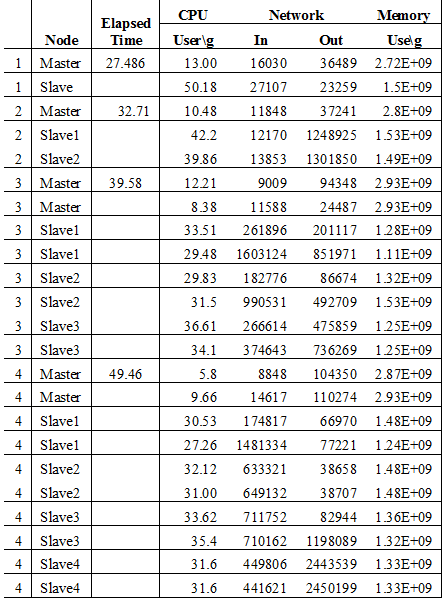


Figure Farthest pair raw data

### **Geometry closest pair**

**Run Time:** As expected, the run-time of the operation increased when the number of nodes increased. The reason that we expected this behavior was that our configuration had virtual machines running on a single host machine. Secondly, the machines are low-resourced. We expect the run-time to decrease if the system configuration was with high-resourced physical machines, until the network overheads exceed the gained computing power.

Figure 23. Geometry closest pair run time

**CPU Used:** As seen in Figure 24, the CPU utilization of the master increases (slightly) when the system setup included 4 worker nodes. We expect this behavior due to the overhead costs of maintaining the 4 worker nodes. Also, it is evident that there is a sharp drop in the worker node’s CPU utilization, because the initial data set has been distributed. Now, the worker nodes have less data to process and run the algorithm on.

Figure 24. Geometry closest pair CPU used (% Cycles spent in user mode v/s # of nodes)

**Memory Used:** As seen Figure 25, the master is almost running at its memory capacity. This is because the master has to perform overhead processes for the cluster to maintain its integrity. Also, the workers are running at their memory capacity since the operation utilized an in-memory divide-and-conquer algorithm.

Figure 25. Geometry closest pair memory used

**Network Traffic:** As the number of nodes in a cluster increases, we should see an increase in the network traffic. However, looking at the first two data points in the Figure 26, we see that the network traffic has dropped. This anomaly is caused by missing data points from Ganglia, especially for the second data point. The reason the network traffic should increase is that the master has to communicate more as the worker nodes increase. Table can explain this anomaly better.

Figure 26. Geometry closest pair network traffic

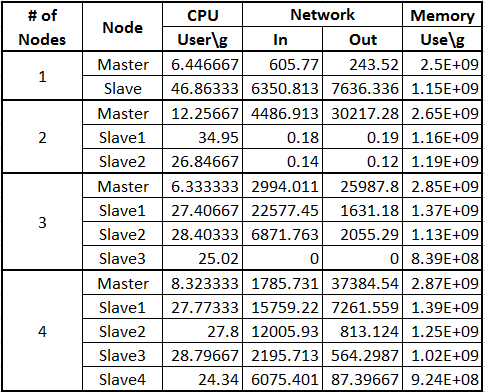


Figure 27. Geometry closest pair raw data (Averaged for data points)

### **Spatial range query**

Figure 28. Spatial range query run time

As expected, the run-time of the function increases as the number of nodes in the cluster increases as seen in Figure 28. This is due to the nature of the experimental system which consists of multiple virtual machines running on a single host system. If the experiment were run in a cluster of multiple physical machines, then a decrease of elapsed time would be expected as the cluster size increases until the communications overhead overwhelms the advantage of increasing processing power.

Figure . Spatial range query CPU used

As shown in Figure 149, the master node uses about the same amount of CPU as the number of nodes increases. There is a very small drop-off. This is likely due to the increased run-time. The master node will perform the same tasks over a longer period, lowering the average CPU load.

Figure . Spatial range query memory used

Figure 30 shows the memory used on the master and the average of the slave nodes. The master shows a small increase as the number of slaves increases. This is likely due to the need to keep track of the work from these slave nodes. The slaves show a tweaked change in memory usage, it increased when there are two nodes and then almost remained constant. One possible explanation is the limited size of the experimental data set and also because of some other issue with the memory utilization.

Figure . Spatial range query network bytes sent

The increase in network traffic shown in Figure 31 as from going to more than one node due to the increase in coordination between the master and slave nodes to perform the algorithm is required as the cluster gets more complicated. But after a certain point we don’t see much difference. One possible explanation could be the limited size of the experimental data.

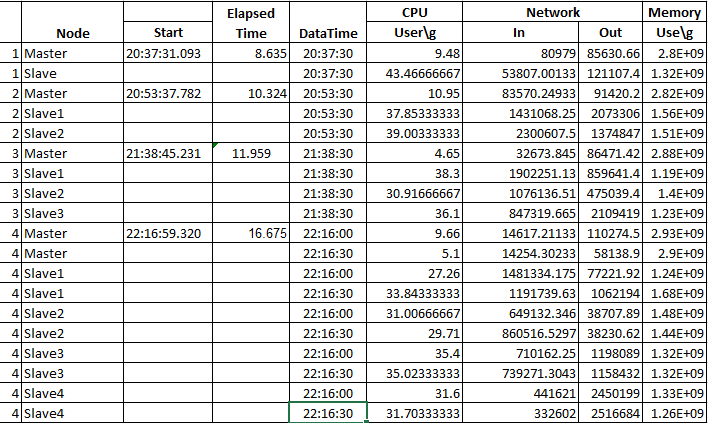


Figure . Spatial range query raw data

### **Spatial join query**

Figure . Spatial join query run time

As expected, the run-time of the function increases as the number of nodes in the cluster increases as seen in Figure 33. This is due to the nature of the experimental system which consists of multiple virtual machines running on a single host system. If the experiment were run in a cluster of multiple physical machines, then a decrease of elapsed time would be expected as the cluster size increases until the communications overhead overwhelms the advantage of increasing processing power.

Figure . Spatial join query CPU used

As shown in Figure 34, the master node uses about the same amount of CPU as the number of nodes increases. This is likely due to the increased run-time. The master node will perform the same tasks over a longer period, lowering the average CPU load.

Figure . Spatial join query memory used

Figure 155 shows the memory used on the master and the average of the slave nodes. The master shows a small increase as the number of slaves increases. This is likely due to the need to keep track of the work from these slave nodes. The slaves show a tweaked change in memory usage, it increased when there are 2 nodes and then almost remained constant. One possible explanation is the limited size of the experimental data set and also because of some other issue with the memory utilization.

Figure Spatial join query network bytes sent

The increase in network traffic shown in Figure 1636 as more nodes are added to the cluster is expected, as more coordination between the master and slave nodes to perform the algorithm is required as the cluster gets more complicated. In this for 2 and 3 mode cluster network traffic remained same because of limited data. But as expected for 4 node cluster it increased.

Figure 37 shows the raw results of running the Spatial join algorithm on the class provided point data set. This is analyzed from files taken from Ganglia and later processed to produce the above explained charts.



Figure . Spatial join query raw data

### **Geospatial Aggregation**

Figure 38. Geospatial aggregation run time

Figure 3838 shows the runtime for spatial aggregation drops dramatically with two slaves in the cluster. Then it increases with three, and then drops again with four. Looking at the raw data in Figure 42, it appears the algorithm is not fully utilizing the slaves in with two or four, like it is with one and three.

Figure 39. Geospatial aggregation CPU used

In Figure 39, we see the average slave CPU values drop with two, then slightly increase with three and four. This is likely due to the increased communications overhead as more slaves are added to the cluster without a real increase in CPU power. We also see the master CPU usage drop with two slaves, but then increase with three or four. This is probably due to the increased amount of communications with the slaves required as the cluster grows.

Figure 40. Geospatial aggregation Memory used

Figure 40 shows the memory used on the master and average memory across the slaves for each cluster size. The data in this chart doesn’t appear to give us any differences between the runs with different cluster sizes. Figure 41 show the total network bytes sent/received by all nodes in the cluster. The increasing network communications is expected as the master must coordinate the activities of more nodes. For a cluster with four slaves the amount of communication drops, which is not expected. This is explained by looking at Figure 42, which seems to show clusters three and four less active in this run than with smaller clusters. The raw data used to generate the graphs above is shown in Figure 42 The CPU and Memory usage is averaged across all readings for the master and all slaves. The network usage is totalled of all readings for master and slaves for the cluster.

Figure 41. Geospatial Aggregation Network Used

As discussed above, the Slave2 in the cluster with two slaves, and Slave3 and Slave4 in the cluster with four slaves do not appear to be taking their share of the load. This may be due to the way this geospatial aggregation function was designed and implemented.



Figure . Geospatial aggregation raw data

# CONCLUSION

Our experiments showed several interesting factors. Unfortunately our experiments suffered from several limitations that seriously affected the results. The first is the limitation of executing the experiments on a single host system with the cluster implemented on multiple VMs on that system. The second is the limited data size for the test data. The rectangle data had only 50k records and the point data had less than 150k records. These data sets resulted in run times that never exceeded 60 seconds. This limited the available metric collection time, as the nodes didn’t really have time to settle before the execution was complete.

One factor discovered was the generally expanding the cluster size showed increases in elapsed time for computation. This result is easily explained as the clusters were all executing on a single physical host machine. Therefore increasing the size of the cluster did not add any real capability to the system, but introduced latency of communication between the clustered nodes.

The results also showed the CPU and memory load decreased on the individual slaves as more slaves shared the load. Also the network communications total went up dramatically as the cluster size grew. These two results are exactly as expected, with the load being shared between more machines therefore requiring more network transmissions to disperse, synchronize, and collect the results.

If the experiment were conducted with a cluster where adding nodes actually added additional processing power we would expect to see some differences in the results. One expectation would be to see a decrease in the processing time as nodes were added. This is due to the addition of more physical CPUs to provide enhanced processing power. Another expectation would be the larger impact of network traffic on the overall run-time. This is due to the requirement of the physically separate nodes to communicate over a physical network, instead of the virtual network between VMs on the same host.

The results would be a general decrease in processing time as the number of nodes increased, until the network latency impact overwhelmed the advantage of multiple processors, then the overall elapsed time would increase.

## Contributions

Anoop Sahoo – Spatial Range Query

Bernard Ngabonziza – Geometry Closest Pair

Jon Lammers – Geometry Convex Hull

Madhu Meghana Talasila – Spatial Join Query

Nagarjuna Myla – Geometry Farthest Pair, Spatial Aggregation

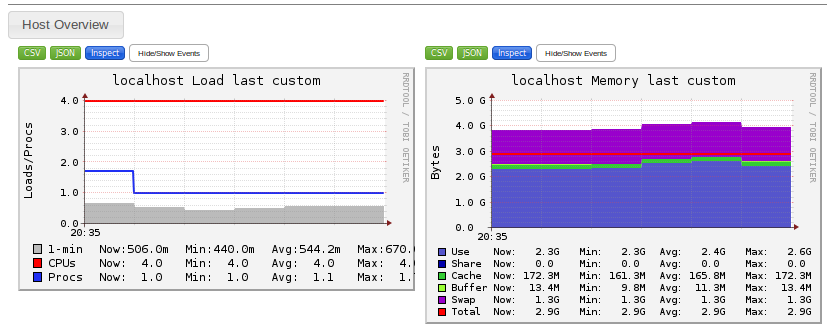
Vageesh Bhasin – Geometry Union

# REFERENCES

1. HDFS Architecture, 2015, November 30, online: https://hadoop.apache.org/docs/r1.2.1/hdfs\_design.html
2. Margalit A., Knott G.D., *An algorithm of computing the Union, Intersection or Difference of two polygons*, 1989, <http://www.cc.gatech.edu/~jarek/graphics/papers/04PolygonBooleansMargalit.pdf>
3. Davis M , *Fast polygon merging in JTS using Cascaded Union*., 2007, <http://lin-ear-th-inking.blogspot.com/2007/11/fast-polygon-merging-in-jts-using.html>
4. A. M. Andrew, "Another Efficient Algorithm for Convex Hulls in Two Dimensions", Info. Proc. Letters 9, 216-219 (1979)

# 

# Addendum – Run Time Figures



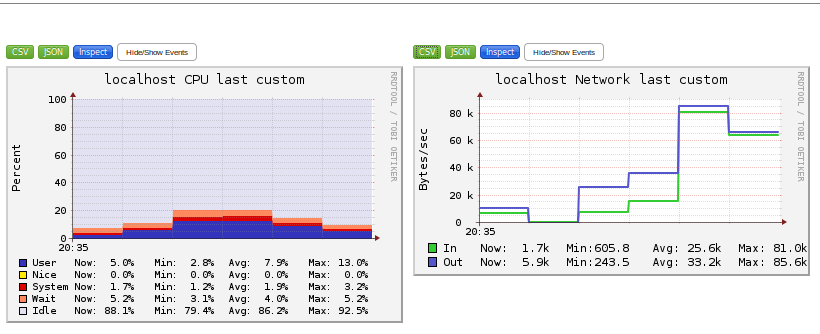


Figure 43. Original six functions – Master ganglia reports

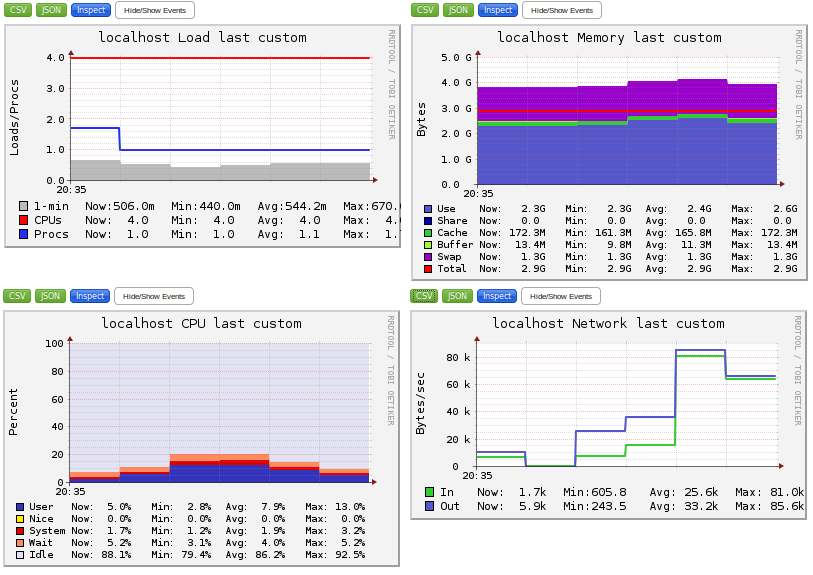


Figure 44. Original six functions - Slave ganglia reports

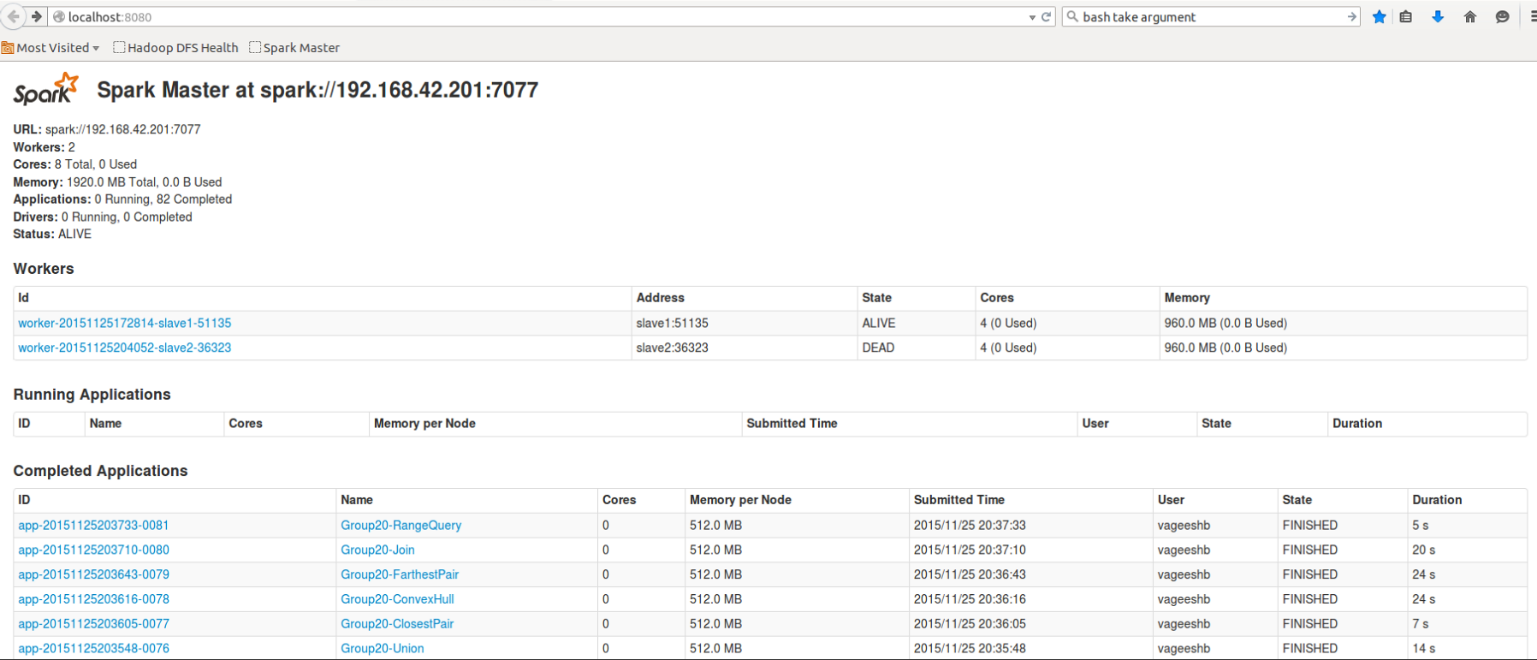


Figure 45. Original six functions - Spark status



Figure 46. Geospatial aggregation - Cluster 1 - Spark status

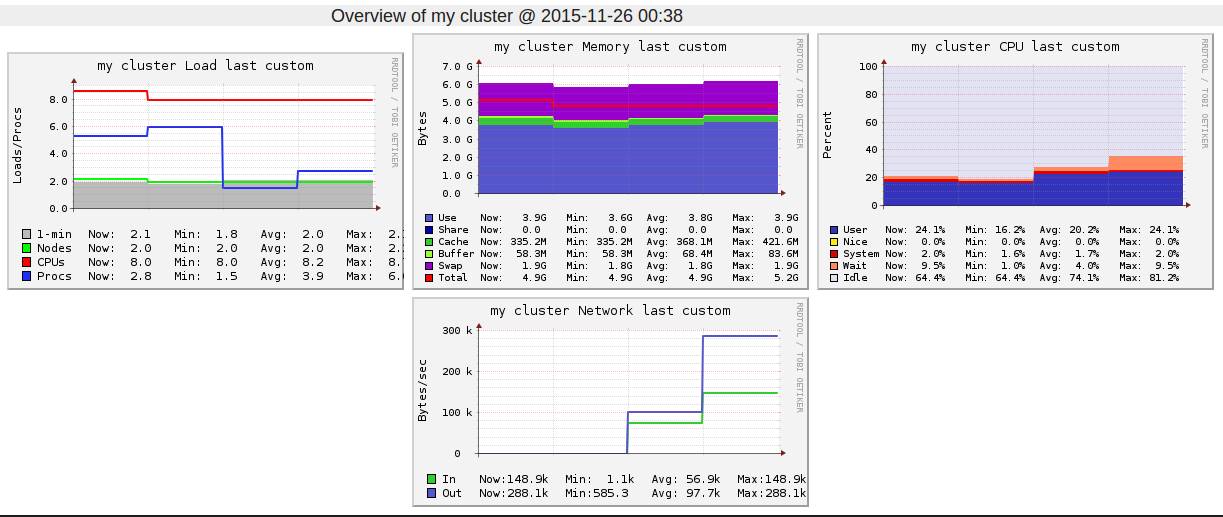


Figure 47. Geospatial aggregation - Cluster 1 - RRDTool

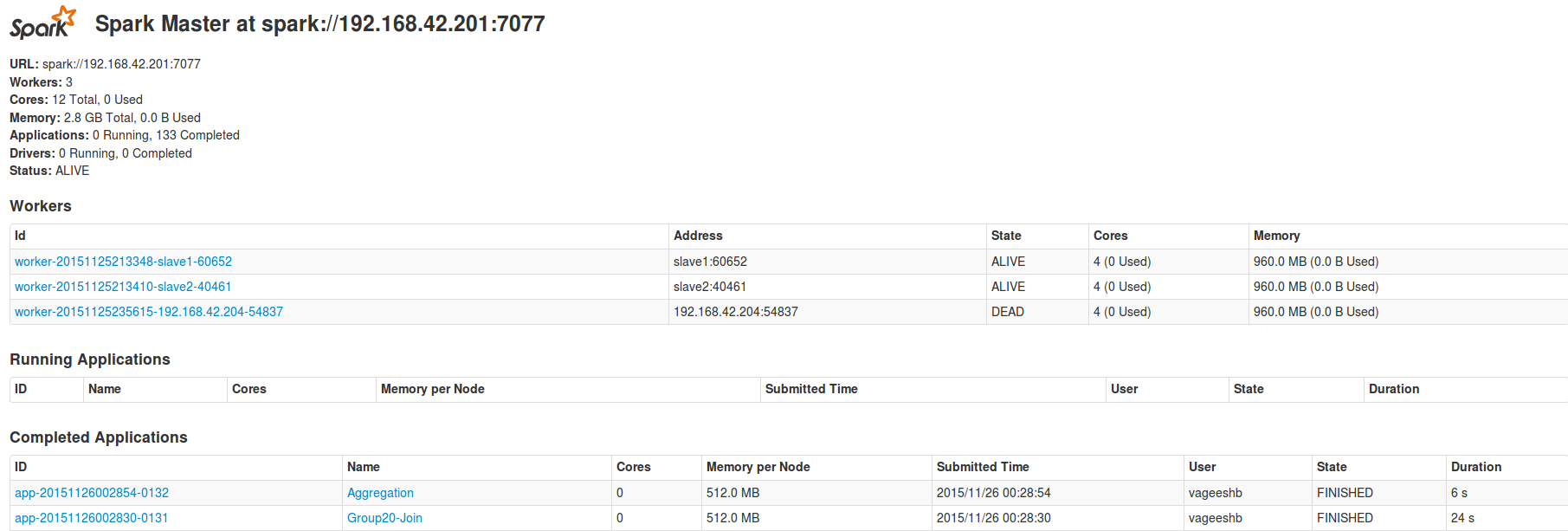


Figure 48. Geospatial Aggregation - Cluster 2 - Spark Status

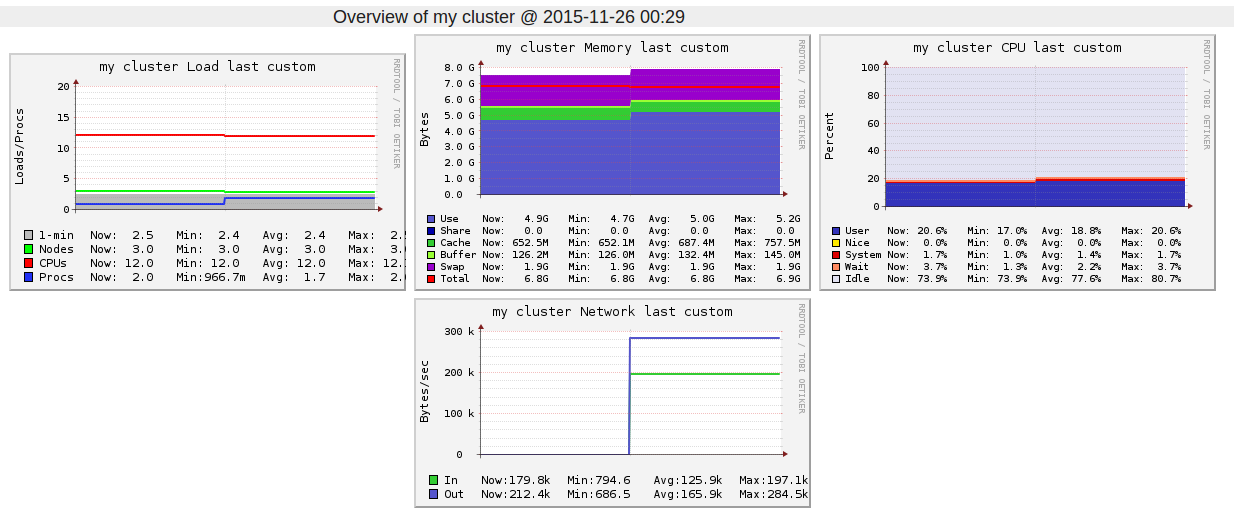


Figure 49. Geospatial Aggregation - Cluster 2 – RRDTool

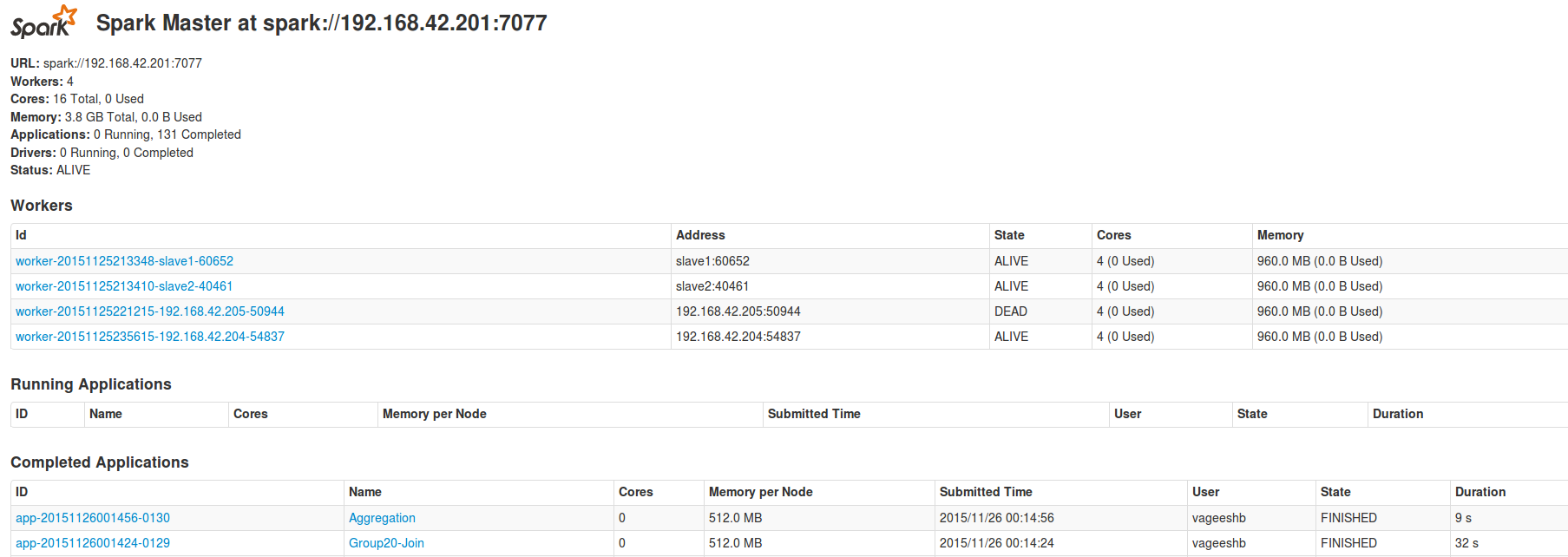


Figure 50. Geospatial aggregation – Cluster 3 - Spark Status

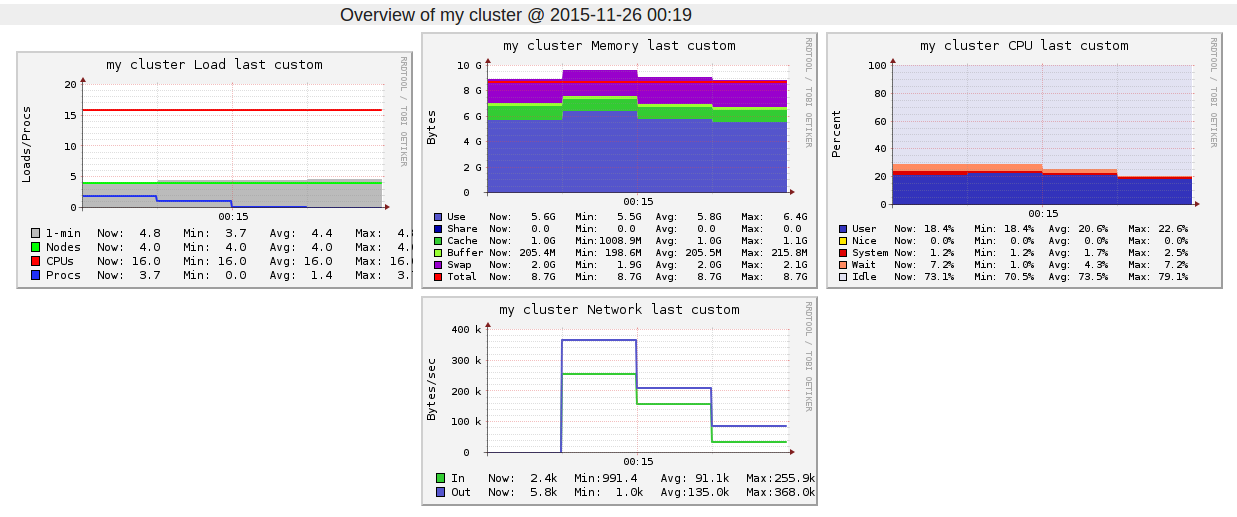


Figure 51. Geospatial aggregation - Cluster 3 – RRDTool

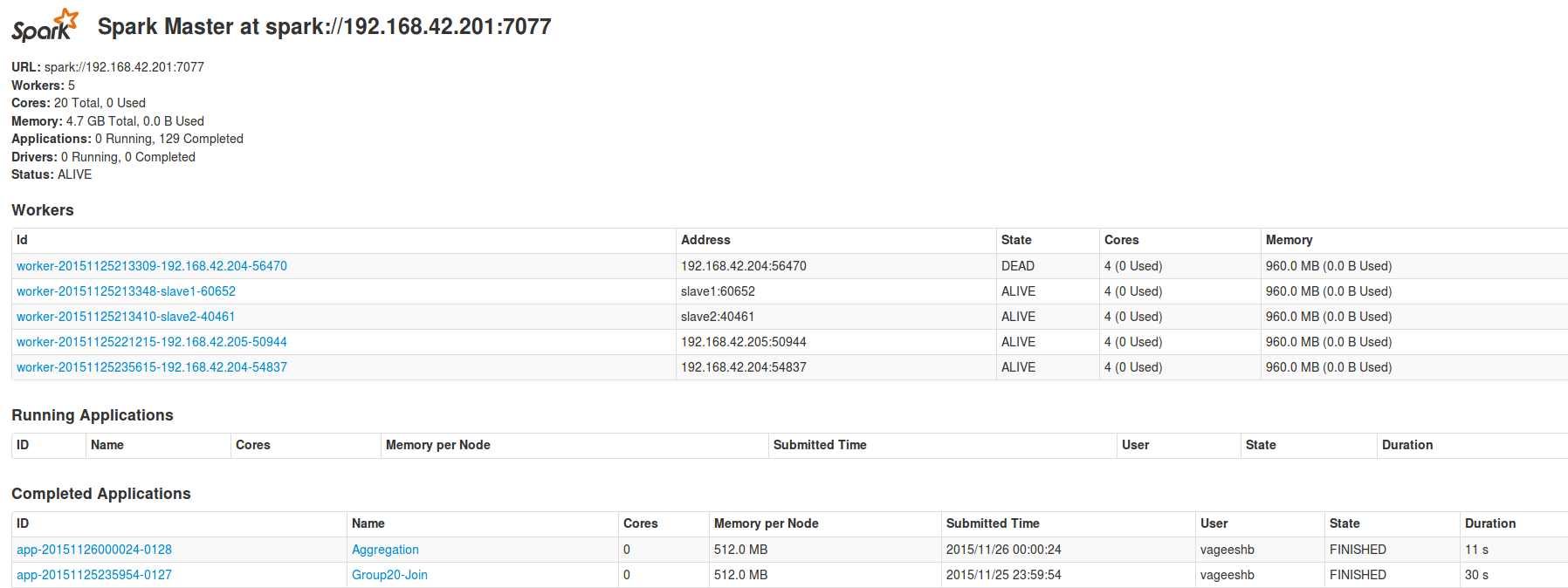


Figure 51. Geospatial Aggregation – Cluster 4 – Spark Status

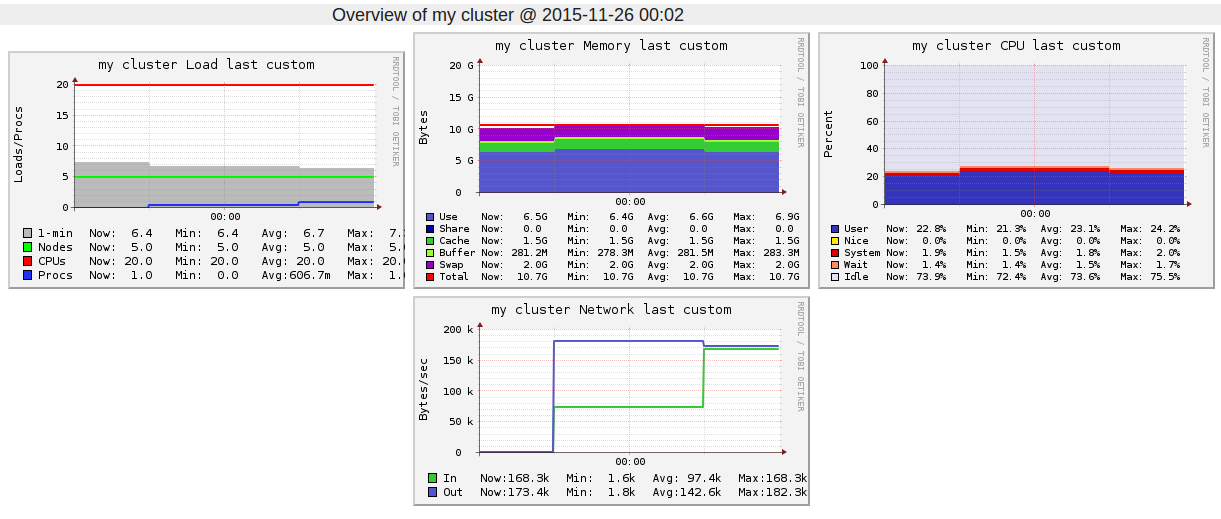


Figure 52. Geospatial aggregation - Cluster 4 – RRDTool