# Hotspot Analysis Using Apache Spark, Hadoop and AWS

**Prashant Ravindra Jadhav**
prjadhav@asu.edu
Arizona State University
Tempe, AZ

**Jayesh Kudase**
jkudase@asu.edu
Arizona State University
Tempe, AZ

**Sushmita Muthe**
smuthe@asu.edu
Arizona State University
Tempe, AZ

## ABSTRACT

The Hotspot analysis aims at extracting insightful analysis from the geo-spatial data of NYC Taxi Trip. The problem with dealing such huge dataset is the large processing time to perform plethora of operations. This paper provides an overview of evaluation metrics derived on varying size dataset on a single and multi-node setup using Apache Hadoop , Spark and Amazon Web Services.

## 1 INTRODUCTION

We performed various experiments to derive the data analysis metrics which includes executing the functions like HotcellAnalysis, HotzoneAnalysis, RangeQuery, RangeJoinQuery, DistanceQuery, DistanceJoinQuery on datasets of varying sizes of data . We divided the dataset in three parts as small, medium and large and observed the network in and out flow and memory utilization on AWS ubuntu clusters where our hadoop and spark images are installed. We will discuss detailed analysis in following report for different sizes of data.

## 2 EXPERIMENTAL SETUP AND METHODOLOGY

Our group created one free tier account on Amazon Web Services in order to set up the architecture needed for executing the project. Three Ubuntu/Linux t2 micro instances [3] were created named as Namenode (master node), DataNode001 and DataNode002 (worker nodes) as specified in the table 1. In order to access these instance remotely from a Windows system, we added a security group with type 'All traffic' and source 'Anywhere'. Once these instances were created, we configured them to support a paswordless ssh communication between odes we configure passwordless ssh communication between Datanode001 and Namenode using private and public IP and passwordless ssh communication between Datanode002 and Namenode using private and public IP. We followed the guidelines provided in the project document in order to set up Apache Hadoop and Apache Spark and also referred few external sources available at [1] and [2]

We used puTTy in order to communicate with the instances and WinSCP to securely move local files to these remote instances. In order to submit the job to Apache Spark,

| Node Name | Private IP | Disk Space | RAM |
|---|---|---|---|
| NameNode | 172.31.17.41 | 30GB | 1GB |
| DataNode001 | 172.31.27.7 | 30GB | 1GB |
| DataNode002 | 172.31.28.132 | 30GB | 1GB |

**Table 1: Clusters Configuration**

we needed to create a jar file. This was done by running the 'sbt assembly' command inside the root folder of the project. Once the jar file is created at the master node, we ran commands to execute the functions such as rangequery, rangejoinquery, distancequery, distancejoinquery, HotcellAnalysis and HotzoneAnalysis. An example of such command is as follows:

```
/usr/local/spark-2.3.4-bin-hadoop2.7/bin/spark-submit
/home/ubuntu/CSE512-Project-Phase3-master/target/scala-
2.11/CSE512-Hotspot-Analysis-Template-assembly-0.1.0.jar
test/output hotzoneanalysis src/resources/point-hotzone.csv
src/resources/zone-hotzone.csv hotcellanalysis
src/resources/yellow_trip_data_medium.csv
```

In order to evaluate the CPU utilization, Network Utilization and Memory Utilization, we had set up various experiments to generate these metrics. These metrics are defined as follows.

CPU Utilization: The percentage of compute resources used by an instance at any point. Since the instances used are t2 micro, the processing of the commands mentioned above take a heavy toll on these resources. Thus, the instance may end up using more than 70% of the compute resources to perform efficiently.

Network In: This can be defined as the number of data packets received by an instance across all network interfaces. These are generally calculated as the number of bytes received by the instance.

Network Out: This can be defined as the number of data packets sent out by an instance across all network interfaces. These are generally calculated as the number of bytes sent by the instance.

**SparkMeasure** [9] is a tool that gives Spark workload metrics for performance analysis and troubleshooting of Spark jobs. We make use submissionTime, completionTime and peakExecutionMemory stage metrics collected from SparkMeasure to monitor Execution time and Peak Execution Memory. They are defined as follows:

Execution time: This is the total time taken by a function e.g. HotcellAnalysis, rangejoinquery, etc to execute. This is calculated as the difference between completionTime and submissionTime from the SparkMeasure metrics.

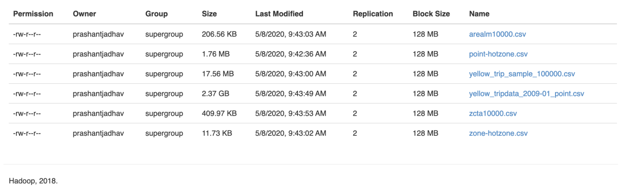
Peak Memory usage: This is defined as the memory used by internal Spark data structures that are generated during shuffling, aggregation and joining activities. We consider the sum of the peak sizes across all such data structures created on executing a particular function say HotzoneAnalysis, rangequery,etc.

We perform various experiments to derive the above metrics which includes executing the functions like HotcellAnalysis, HotzoneAnalysis, rangejoinquery,etc on datasets of varying sizes.The datasets used are available at [5] and [6]. New York Trip dataset is downloaded and used from [7] for month of January and year 2009. We have divided the input data in three different forms- small, medium and large as shown in table 2.

| Type | Size |
|---|---|
| Small | 17.98 MB |
| Medium | 578.43 MB |
| Large | 1.15GB |

**Table 2: Data Distribution**

This divided data is then copied to Hadoop File System as shown in below figure 1 and used it to feed to the spark submit commands to run the jobs on clusters.



| Permission | Owner | Group | Size | Last Modified | Replication | Block Size | Name |
|---|---|---|---|---|---|---|---|
| -rw-r--r-- | prashantjadhav | supergroup | 206.56 KB | 5/8/2020, 9:43:03 AM | 2 | 128 MB | arealm10000.csv |
| -rw-r--r-- | prashantjadhav | supergroup | 1.76 MB | 5/8/2020, 9:42:36 AM | 2 | 128 MB | point-hotzone.csv |
| -rw-r--r-- | prashantjadhav | supergroup | 17.56 MB | 5/8/2020, 9:43:00 AM | 2 | 128 MB | yellow_trip_sample_100000.csv |
| -rw-r--r-- | prashantjadhav | supergroup | 2.37 GB | 5/8/2020, 9:43:49 AM | 2 | 128 MB | yellow_tripdata_2009-01_point.csv |
| -rw-r--r-- | prashantjadhav | supergroup | 409.97 KB | 5/8/2020, 9:43:53 AM | 2 | 128 MB | zcta10000.csv |
| -rw-r--r-- | prashantjadhav | supergroup | 11.73 KB | 5/8/2020, 9:43:02 AM | 2 | 128 MB | zone-hotzone.csv |

Hadoop, 2018.

**Figure 1: Hadoop File System**

Besides this, in order to perform all operations like rangeQuery, rangeJoinQuery, distanceQuery, distanceJoinQuery, hotcellanalysis and hotzoneanalysis and derive all the metrics in a single run, we have merged the code base from Phase 1 and Phase 2 of project and modified Entrance.scala file. This file handles any type of input command to execute a single or multiple operations in one go. All the spark metrics are printed and written to a csv file once the command is executed.

## 3 EXPERIMENTAL EVALUATION

We performed various experiments in order to study and analyze CPU utilization, Network Utilization and Memory Utilization. We identified experiments related to hotcell analysis since its quite evident that these would be most demanding in terms of CPU and Network utilization. As a sample set, we hereby provide our findings for the HotcellAnalysis executed on small, medium and large size dataset. The small dataset is of size 17.98 MB, medium of size 578.43 MB and the large dataset is of size 1.15 GB. All the graphs in this section were taken from the metrics provided by AWS available at [8].

### CPU Utilization - Single Node

In order to analyze the CPU Utilization on a single node cluster, we ran the HotcellAnalysis with small, medium and large data. As it can be seen in fig 2, X-axis being time and Y-axis being CPU Utilization Average (%), the CPU is initially in the idle state as no processes are running on it as of yet. We started executing the HotcellAnalysis on small dataset approximately at 16:13 and this the reason that shortly thereafter, we see a spike in the usage of CPU. But the graph quickly falls down because the total execution happened in 6 seconds which is very short period of time. Shortly afterwards, we executed a wrong script that lead to increase in the CPU utilization, but we aborted it quickly and then ran the script to execute the HotcellAnalysis on medium dataset. This explains the constant rise in the time period between 16:25 to 16.28. Once the execution reached its peak memory execution at 16.30 the graph falls downs quickly. The execution time on the medium dataset was 3 minutes 19 seconds and is very promptly reflected in the graph.

Once the above two experiments were done, we quickly executed the HotcellAnalysis on large dataset. As evident from the graph, the processor reached its maximum usage and didn't fall down below 10 %. It continued on processing the data for a very long time. It took us almost 2 hour 30 minutes to complete this task.

### CPU Utilization - HotcellAnalysis on Multi Node

This experiment was done prior to the above experiments to gauge the utilization of CPU on all 3 nodes while working with large dataset. The results that were obtained is as shown in fig 3

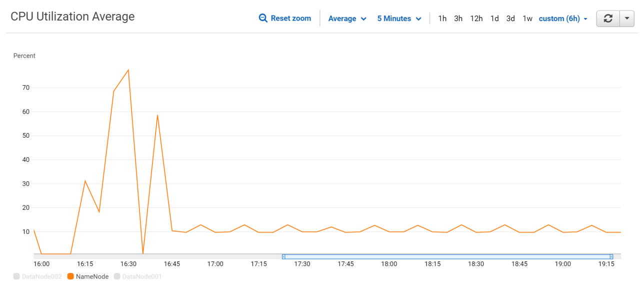


**Figure 2: CPU Utilization during HotcellAnalysis - Single Node**

The worker nodes namely DataNode001 and DataNode002 have almost similar utilization which is true the fact that the master node, in our case NameNode, has distributed the data evenly among the data nodes. We can see that the CPU usage increase for the data node initially and once it distributes the data to be processed among the worker nodes, its CPU usage decreases. At the very same time, it can also be seen that the CPU usage of the worker nodes (DataNode001 and DataNode002) takes a sharp increase initially and once the data is available locally, the CPU usage of these nodes decreases. The calculations happen locally on each processor running on its own local data. Once the worker node are finished with their evaluation, they send the results back to the master node (NameNode). This can be purely extracted from the figure at an approximate time 15:12 wherein the CPU usage of both the worker nodes decreases and that of the master node increases.

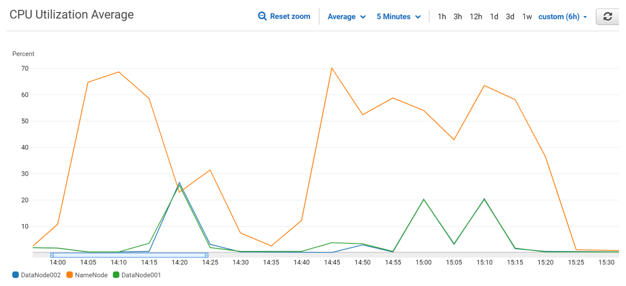


**Figure 3: CPU Utilization during HotcellAnalysis - Multi Node**

The execution took a total time of 1 hour 25 minutes on this muti node (3 nodes) cluster. Looking from an Apache Spark perspective, it was quite intuitive that the time taken by this multi node cluster would be less than that of the execution time on a single node. The same was substantiated by the execution time and CPU utilization as discussed above. The processing workload was distributed among the slave nodes and thus, the master node didn't had to process complete data in this setup. Apache Spark provided a parallel processing of distributed data and thus the run time via this multi node setup was significantly lower than what was observed in the single node setup.

### Network Packets In - Multi Node Setup (HotcellAnalysis)

For the analysis of this section, we will consider the net average network packets that are coming in at any node in the Multi node setup. As seen in the fig 4, once the job to execute the HotcellAnalysis is submitted at the master node, there is a gradual increase in the incoming packets at the worker nodes (DataNode001 and DataNode002). The number or packets incoming at the worker nodes reach a limit and then we can see a decline. Once the execution at the worker nodes is finished locally, these worker nodes are required to send out the results back to the master nodes. This is explained by a sudden spike in the incoming traffic at the master node at time 15:20.

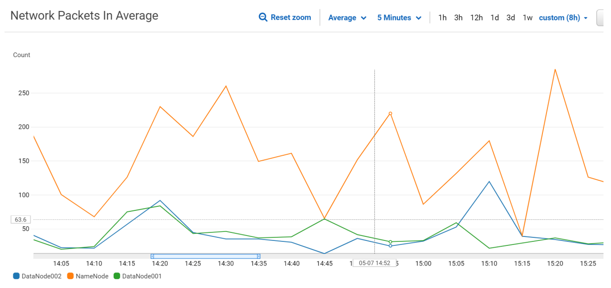


**Figure 4: Network Packets In Average - Multi Node**

The parallel processing in a distribute data environment is perfectly captured by the graph and aids to the explanation about working of Apache Spark.

### Network Packets Out - Multi Node Setup (HotcellAnalysis)

It can be seen from fig 5 that initially the network packets coming out of the master node is very high. This is due to the fact that the master node is distributing the data that needs to be processed among the slave nodes. Thus, at the same time there are very little or no packets going out of the slave nodes.

Understandably, the network packets going out of the slave nodes increases once their local processing is finished and the result needs to be sent back to the master. At the same time, network packets coming out of the master node takes a dip. Thus, this specifically shows the network packets outgoing usage in a distributed parallel processing.

### Peak Execution Memory:

To calculate peak execution memory, we submitted each function individually and recorded the aggregated peakmemory

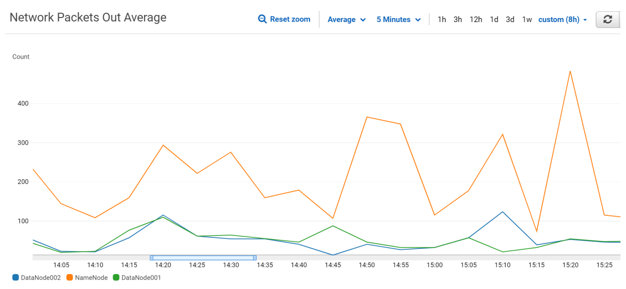


**Figure 5: Network Packets Out Average - Multi Node**

SparkMeasure metric. In the figure 6 we see Peak execution memory for each function executed on a small dataset on single node cluster. We observe that the peak execution memory is least for Spatialquery functions. Hotzone and Hotcell analysis functions show high peak execution memory. This is due to the use of persist action. Persist makes use of default storage level i.e. MEMORY ONLY and waits for the cache manager to store the DataFrame which is why we see a hike in peak execution memory for both Hotzone and Hotcell.

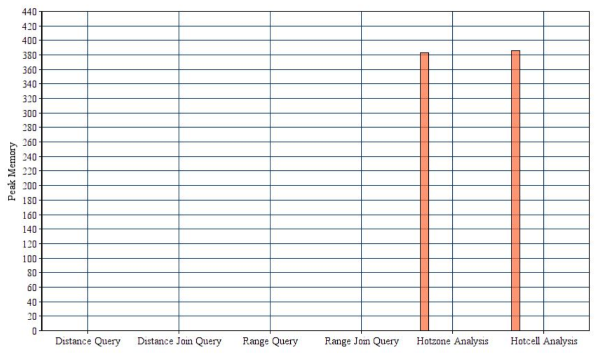


**Figure 6: Peak memory for all functions - Single Node**

### Execution Time:

To calculate execution time, we submitted each function individually once on cluster with single node and then again on a cluster with 3 nodes. We did this for 3 data sets viz. small, medium and large and observed graphs for each.Execution time for each function is calculated as difference of completionTime and submissionTime as collected using the SparkMeasure stage metric. In the figure 7 we see execution time (in seconds) for each function executed on a small dataset on single node cluster as well and multinode cluster.

We observe that the execution time is least for Range and Distance query functions. For other functions, the execution time is high due to the use of Spark SQL functions for join calculation between data frames.

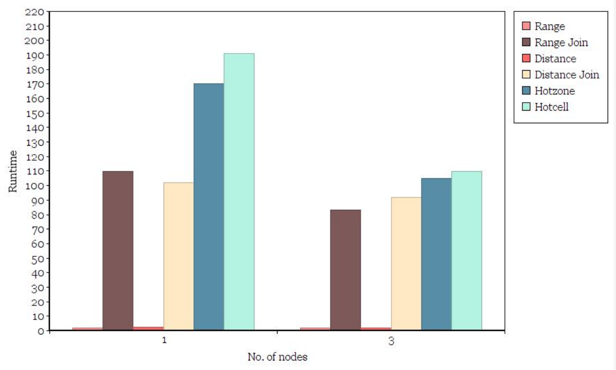


**Figure 7: Execution time for all functions - Single Node and Multi node (Small data-set)**

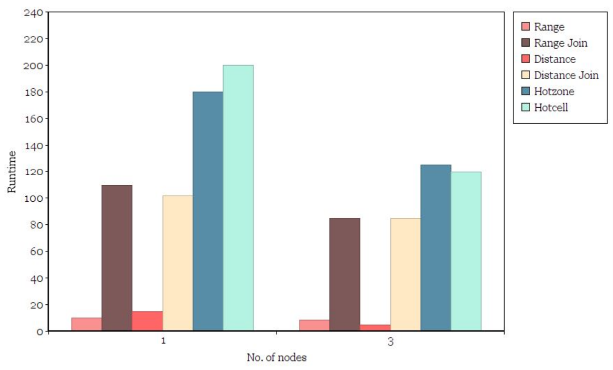


**Figure 8: Execution time for all functions - Single Node and Multi node (medium data-set)**

We also note that even though Hotzone and Hotcell analysis perform more join operations than the spatial join functions, the use of persist function in Hotzone and Hotcell analysis reduces the execution time. Persist is performance optimisation method offered by Spark, and it saves the time involved in recomputing of joins between large data sets when output of one such join is to be used again for near future computations. For Multi node cluster we see that the execution time for all functions on same data set falls compared to the execution time of these functions on single node due to scalable nature of Spark. Thus due to parallelism and availability of more resources for computation in multi node cluster decreases the execution time.

Figure 8 and 9 represents execution time (in minutes) for each functions executed on a medium and large data-sets on single node as well as multi nodes clusters respectively. Range and Distance query functions make use of count, which returns a single value and not a RDD/dataframe and therefore their execution time is least.

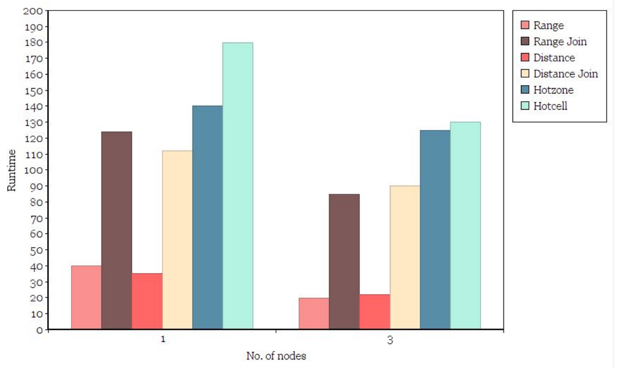


**Figure 9: Execution time for all functions - Single Node and Multi node (large data-set)**

As the data increases, the execution time also increases for each function, but the relative increase in execution time for multinode cluster is not high due to parallelism.

## 4 CONCLUSION

The analysis from above various experiments is a testament to the Apache Spark's low latency and high parallel processing in a distributed environment. Through this project and the experiments, we got a hands-on experience in writing Apache Spark code using built-in APIs for interactive querying the data set. In general we observed that when the workload is not distributed across different nodes, it creates restrictions on resources like CPU utilization and thus gives sub optimal performance. Also, when the number of tasks in a job are too small and there are fewer tasks than the available slots/nodes to run them in, the submitted spark job won't take advantage of all the CPU available. We also observed that persist is a memory intensive action but it helps reduce the execution time. The NYC- Taxi Trip data [7] served as a good data-set to understand the faster execution of queries in a distributed environment using Apache Spark. After performing a careful analysis of the above experiments, we understood the power of Spark to parallel process huge data-sets on multiple nodes.

## REFERENCES

[1] https://www.youtube.com/watch?v=XkDhf1pwPtA&list=PLWsYJ2ygHmWjPsg-6MnQO6WxVWFl8OzK_

[2] https://www.youtube.com/watch?v=tKsR4tnEdHM&list=PLWsYJ2ygHmWjPsg-6MnQO6WxVWFl8OzK_&index=2

[3] https://aws.amazon.com/ec2/

[4] https://klasserom.azurewebsites.net/Lessons/Binder/1960

[5] https://github.com/YuhanSun/CSE512-Project-Phase1-Template/tree/master/src/resources

[6] https://github.com/YuhanSun/CSE512-Project-Phase2-Template/tree/master/src/resources

[7] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[8] https://aws.amazon.com/cloudwatch

[9] https://db-blog.web.cern.ch/blog/luca-canali/2018-08-sparkmeasure-tool-performance-troubleshooting-apache-spark-workloads