SPATIAL DATA ANALYSIS USING SPARK

Molife Chaplain
mchapla1@asu.edu
Arizona State University

Naga Sai Teja Vinnakota
nvinnako@asu.edu
Arizona State University

Pranav Bajoria
pbajori1@asu.edu
Arizona State University

ABSTRACT

Data has become a very powerful commodity in the day to day running of institutions and organizations. With this trend and rise in data size, increasing the efficiency of the analysis and recognition of anomalies and statistical patterns has become a very important problem, for which a myriad of solutions are available, being improved upon and being developed. Spatial data analysis in this paper takes the specific form of Geo-spatial analysis of the New York City taxi data with emphasis on significance of distributed computing. This paper analyses the efficiency of Apache Spark, a framework for distributed query processing, in terms of its ability to scale-up and speed-up with respect to varying data size and resource availability.

KEYWORDS

Data Analysis, Spatial Data, Geo-Spatial Data, Apache Spark, Spark SQL, Apache Hadoop, HDFS, YARN, Distributed Computing, Amazon Web Services

1 INTRODUCTION

The term GeoSpatial data refers to information that has a geographic nature, especially with regards to location and movement. This type of data is used in information gathering, product tailoring and resource distribution for efficiency in business, governance and health. The very nature of this data, it's continuous processing, analysis and sheer volume of this data makes its processing very complex. This means that such data cannot be conventionally processed, and hence requires special large scale distributed data processing frameworks capable of efficiently handling the data.

The nature of these distributed frameworks is that they have better fault tolerance and replication. For this project, we use data from the New York City taxi data. For the frameworks we use two powerful frameworks, these are: Apache Hadoop and Apache Spark.

The system used in this project is Amazon T3 based and these machines have Apache and Hadoop installed on them. In this paper, we will analyze the NYC taxi data whilst varying different metrics like number of machines being used to analyze efficiency and parallel combinations to increase productivity. During these parameter changes we will measure and document efficiency using graphs and monitor the changes. We will also provide visual evidence and an analysis of these changes and give details of the system architecture.

2 DATA ANALYSIS USING APACHE SPARK

Spark SQL is an important module of the Apache Spark framework which is used to raise queries on structured data while bringing native support for SQL to Spark [5]. It provides a neat layer of abstraction between the RDDs and conventional relations/tables with DataFrames and also enables faster query processing. To perform data analysis, we used spark SQL programs written in scala to be submitted as jobs to the Spark application.

To perform the final Geo-spatial analysis on the large scale New York Taxi Cabs data, we first wrote generic functions to perform tasks like Range query, Range Join query, Distance query and Distance join query. Then these functions were used to perform the specific data analysis tasks like Hotzone analysis and Hot cell analysis on the entire data.

Spatial Queries

To perform the four aforementioned spatial queries, we constructed 2 helper functions in scala namely "ST_Contains" and "ST_Within". Further details on their implementation are as follows:

ST_Contains. : This function takes 2 parameters as input, a rectangle and a point and returns a boolean value True if the point is within the boundaries of the rectangle and False if not. To achieve this the following steps were taken:

- The input point and the input rectangle are received as strings which are split using ',' as a delimiter and parsed as float values.
- The rectangle coordinates are stored as a list where first 2 values are the x, y coordinates of the top left corner and the next 2 of the bottom right corner.
- Point is stored in a list where 1st value is the x coordinate and the 2nd the y coordinate.
- There are 2 range checks done on each of the Point's coordinates. If both the coordinates are greater than the minimum of the 2 corners of the rectangle's corresponding coordinates and lesser than the maximum of the 2 corners, a true is returned or else False.

ST_Within. : This function takes 2 points and a distance parameter and returns a boolean indicating whether the distance between the 2 points are within the given value. To achieve this the following steps were taken:

- The input points are received as strings which are split using ',' as a delimiter and parsed as float values.

Figure 1: New York City Yellow Cab Trips data. Blue dots represent pickup points present in the data. [1]

- The euclidean distance between the 2 points are calculated
- If the distance is less than or equal to the given range, a true is returned or else False.

These helper functions are called to execute 4 spatial queries explained as follows:

- Range Query: This query takes in a specified rectangle and a set of points stored in a csv file as input and uses the ST_Contains function to filter and return only the points that lie within the spatial boundaries of the given rectangle.
- Range Join Query: 2 CSV files containing a set of Rectangles and a set of Points are passed as input arguments to this query. The query then joins all the points with the rectangles that return true when the ST_Contains function is called with them.
- Distance Query: This query takes in a set of points stored in a csv file as input and a float Distance value and uses the ST_Within function to filter and return only the points that lie within the spatial range of the given distance.
- Distance Join Query: This query takes 2 sets of points stored in csv format along with a floating value specifying the Distance range for the join. The query returns all the pair of points which return true for the ST_Within function.

Geo-Spatial Data analysis on NYC Cab Data

Dataset. : The final goal of this paper is to perform geo-spatial data analysis on the New York City cab dataset which contains records of the Yellow Cab taxi trips filtered based on a certain spatial range to reduce noise. All the records are for the locations whose latitude lie between 40.5N and 40.9N and longitude between 73.7W and 74.25W. The pickup locations are marked as blue dots in figure 1.

We need to perform 2 specific types of analysis to extract information of the most active locations in the given data, Hot zone analysis and Hot cell analysis.

Hot Zone Analysis: This type of data analysis is done to get the rectangular zones to be sorted based on the number of pickup locations that they consist. So, the higher the count, the hotter that zone is. To perform this analysis, we can simply make use of the range query described earlier on the set of points and rectangles, perform a join and group them by zones and apply the count function. The function is described below.
Function name: 'runHotZoneAnalysis'
Parameters: SparkSession (spark instance), pointPath(String), rectanglePath (String)
Return Value: Dataframe with rectangles and counts of points lying within
Algorithm:

- Load the point and rectangle data into their respective dataframes by using spark read commands and setting the appropriate delimiter as per the raw format of data to fill the columns appropriately.
- Create temporary views of both the dataframes so that we can perform native SQL queries on them.
- register the ST_Contains function to be used while performing the range join query on the rectangle and points.
- Create a join Dataframe by performing an SQL join query on the rectangle column of the rectangles view and point column of the points view using the registered function as the join condition.
- Group the data frame by "rectangle" column, count the number of rows in each group and order by "rectangle" column and return the dataframe as result.

Hot Cell Analysis: This type of analysis is performed to get a certain 'Z score' (Getis - Ord statistic value [6]) to a space-time cube called a 'cell'. This score takes all important data statistics like mean and standard deviation along with the relative neighborhood cell information into consideration. The hotness here describes the statistical significance of a region at a given time. The equation for calculating the score is given in 2 and 3 where
x_j : number of pickup points in cell j
$w_{i,j}$: spatial weight between cell i and j - 1 if a neighbor, 0 otherwise
n : total number of cells

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

Figure 2: Getis-Ord statistic or Z-score equation.[1]

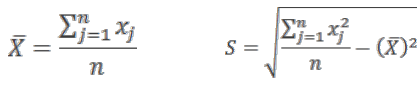


Figure 3: Average (left) and standard deviation (right) of number of pickup points per each cell [1]

The neighborhood to decide the spatial weight is given by all cells that share an edge with the target cell in the space time cube. As given in [1], this cube can be seen as a grid, which is obtained by subdividing the latitude, longitude and the time uniformly. Therefore a cell at the centre would have a total of 26 neighbors.

The steps involved in performing this analysis is given by the following function:

Function name: 'runHotCellAnalysis'

Parameters: SparkSession, pointPath (String)

Return Value: Dataframe with columns latitude, longitude and timestamps ordered decreasingly by their z scores

Algorithm:

- Load the point data into a dataframe by using spark read command and create a temporary view to run Spark SQL queries.
- For converting points to cells, the location column records are sent to a 'CalculateX' and a 'CalculateY' function which then floors the value after using a 'coordinateStep' and converts it to integers and returns the X & Y coordinates of the cell the point belongs to. This is done for the timestamp too to get the Z axis coordinate of the cell in terms of the day it belongs to.
- Now, to get the attribute scores x(i,j) for each cell, we perform a group by on the X,Y and Z coordinates and apply a count function and get a new Dataframe.
- Using this dataframe the mean and standard deviation is computed according to 3.
- Calculate total number of points in all the neighboring cells for each cell by running a spark SQL query.
- use another 'numNeighborsCalculator' function to run a query which appends the number of neighboring cells for each cell.
- Now using the above dataframe which has all the required information for each cell to calculate the Z-score, a 'zScoreCalculator' function is registered and a spark SQL query is run to find the zscores for all the cells and order them in descending order
- return only the x, y, z coordinates of the resulting dataframe as the result.

3 EXPERIMENTAL SETUP

To perform the data analysis task at hand and compare performance based on scaling up and speeding up, the Hadoop ecosystem was chosen. Hadoop is a framework that provides a sophisticated set of utilities like the Hadoop file system (HDFS), to develop projects which are reliable, scalable and that support distributed computing [3]. HDFS allows us to distribute large data across a cluster and can be deployed on low-cost hardware. It provides fault tolerance with the help of replication. While Hadoop also provides a MapReduce framework to perform data analysis, its major drawback is that all the operations should be performed in Map, Reduce phases and the high processing time required while execution as it involves a high number of disk reads and writes, even during the intermediate phases. An efficient solution to this drawback is performing the query execution on the Apache Spark [4] framework which allows in-memory processing resulting in significant increase in performance.

Hence, Apache Hadoop and Apache Spark were installed on all the systems that were a part of the experiments. This was done on AWS T3 instances by taking a remote SSH of all the systems and establishing a password less SSH communication between the master and workers. The detail specifications of the instances and the software are given in table 1.

Platform	AWS (Amazon Web Services)
Cluster Computing Framework	SPARK 2.3.4
Resource Manager Architecture	YARN
Hadoop	Version. 2.7.7
File System	HDFS (Hadoop File System)
HDFS Replication Factor	2
Java	Version. 1.8
Number of Instances	4 (Master: 1, Worker: 3)
Instance Type	t3.medium
Processor	Intel Xeon Platinum 8000 series
Cores	2
CPU clock speed	Up to 3.1 GHz
RAM	4 GB
Hard Disk	15 GB
Network burst bandwidth	5 Gbps

Table 1: Cluster Specifications

To schedule and run the tasks on all the worker machines, a resource manager is required which runs on the master node. We allocated a separate node for the resource manager as a master node acts as a gateway for all the Web UI services such as Hadoop HDFS, Spark history logging, Yarn, etc, and running tasks on it on top of UI services would affect the overall system performance. Of all the available resource managers we chose Yarn, as it has great compatibility with Hadoop and its file system. The Yarn resource manager is configured according to the cluster instances specifications to meet memory requirements.

The list of experiments performed and the commands used to submit the jobs with the required configuration are as follows:

Parameter: num_executors - To specify the number of workers to run the task on.

Spatial Queries on points and rectangles Data

Since the data for the spatial queries is relatively smaller, we decided to just vary the number of working nodes in each experiment. This can be achieved by replacing "#num_workers" with either 1, 2 or 3 worker nodes in the following shell command to submit the spatial query job from Yarn.

```
"./spark-2.3.4-bin-hadoop2.7/bin/spark-submit
–deploy-mode client
–num-executors=#num_workers
–executor-cores 2
/project/CSE512-Phase1-assembly-0.1.0.jar
hdfs://172.31.5.191:9000/outputresult_1
rangequery
hdfs://172.31.5.191:9000/input/arealm10000.csv
-93.63173,33.0183,-93.359203,33.219456
rangejoinquery
hdfs://172.31.5.191:9000/input/arealm10000.csv
hdfs://172.31.5.191:9000/input/zcta10000.csv
distancequery
hdfs://172.31.5.191:9000/input/arealm10000.csv
-88.331492,32.324142 1
distancejoinquery
hdfs://172.31.5.191:9000/input/arealm10000.csv
hdfs://172.31.5.191:9000/input/arealm10000.csv 0.1"
```

Following are the different configurations we tried:

- 1 Worker node
- 2 Worker nodes
- 3 Worker nodes

Geo-Spatial Data analysis on NYC Cab Data

The given NYC taxi data is divided into month-wise trips records, each over 2 GB in size. Upon uploading these files on to the HDFS, the data is split into multiple blocks of 128 mb each across the 3 data nodes that were marked as workers. The hot cell analysis function takes these files as input, and hence we could do experimentation with splitting the data along with varying the number of executor nodes to study the system's ability to scale-up or speed-up.

These parameters can be varied in the command below by replacing "#data_file" with the different sizes, in our case - half and full, and "#num_workers" with either 1, 2 or 3 worker nodes.

```
"./spark-2.3.4-bin-hadoop2.7/bin/spark-submit
–deploy-mode client
–num-executors=#num_workers
–executor-cores 2
–class cse512.Entrance
/CSE512-Hotspot-Analysis-Template-assembly-0.1.0.jar
hdfs://172.31.5.191:9000/outputresult_5
hotzoneanalysis
hdfs://172.31.5.191:9000/input/point-hotzone.csv
hdfs://172.31.5.191:9000/input/zone-hotzone.csv
hotcellanalysis
hdfs://172.31.5.191:9000/input/#data_file"
```

Following are the different configurations we tried:

- 1 Worker node for 15 days data
- 2 Worker nodes for 15 days data
- 3 Worker nodes for 15 days data
- 1 Worker node for 30 days data
- 2 Worker nodes for 30 days data
- 3 Worker nodes for 30 days data

4 EXPERIMENTAL EVALUATION

To better understand the execution of the Spark framework, and study the performance in terms of its ability to scale-up and speed up, we performed a series of experiments by changing the number of workers and the data size. The metric for evaluation was the execution time which has a couple of important parameters like the processing cost and the communication cost. The execution time was measured for each task using the Yarn web UI on the master's web port 8088. The processing cost and the communication cost for each experiment are presented with graphs provided by AWS's monitoring tool - Amazon CloudWatch [2].

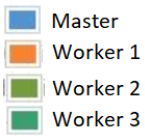


Figure 4: Instances legend

As per the cluster set up shown in table 1, we have 3 live worker nodes and a master node which acts as the driver. There were three major plots that were used in order to perform the evaluation, each plotted against the absolute time. a) CPU utilization representing each instance's percentage of CPU used on the Y axis, b) Network In representing the amount of data, in number of bytes, entering into each instance c) Network Out representing the amount of data flowing out of each instance. in number of bytes. Each instance is represented by a unique color in the plots whose legend can be referred to in figure 4.

Spatial Queries on points and rectangles Data

The results from all the experiments are shown in figure 5. We can observe from table 2 that adding extra worker nodes did not decrease the run-time. This could be a result of the extra communication overhead caused because of transfer of data. We can observe multiple peaks in the network in and

Workers	Data Size	Run Time
1	Full	3 mins 29 sec
2	Full	3 min 37 sec
3	Full	3 mins 31 sec

Table 2: Spatial Data run time analysis

Workers	Data Size	Run Time
1	Half	16 mins 14 sec
2	Half	9 min 14 sec
3	Half	7 mins 0 sec
1	Full	22 mins 0 sec
2	Full	12 mins 15 sec
3	Full	9 mins 2 sec

Table 3: Geo Spatital Data run time analysis

network out graphs of 5b and 5c. Moreover, as expected, we see a single peak in the CPU utilization of 5a and multiple in 5b and 5c.

(a) On 1 Worker Node

(b) On 2 Worker Nodes

(c) On 3 Worker Nodes

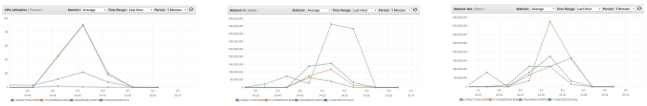
Figure 5: Evaluation metric for spatial queries 3: CPU Utilization(left), Network In (Center), Network Out(Right)

Geo-spatial data analysis on NYC cab data

Since the amount of data to be processed is high, the data gets distributed across almost all available data nodes, when uploaded onto the HDFS. Due to this, whenever the job is executed, the data has to be moved around to enable processing on the respective executors. This is evident by the multiple peaks in the network out and network in plots of all the experiments seen in figures 6 and 7. The peaks depend on the way the driver partitions data and allots tasks to the executors.

From 3, we can observe how increasing the resources results in decrease in run-time that is enabled by parallel processing. Increasing the number of worker nodes also allowed the system to maintain or even expedite the execution time with the increase in data. This shows the system's ability to

(a) On 1 Worker Node



(b) On 2 Worker Nodes

(c) On 3 Worker Nodes

Figure 6: Evaluation metric for Geo-Spatial analysis on half NYC cab data 3: CPU Utilization(left), Network In (Center), Network Out(Right)

(a) On 1 Worker Node

(b) On 2 Worker Nodes

(c) On 3 Worker Nodes

Figure 7: Evaluation metric for Geo-Spatial analysis on full NYC cab data 3: CPU Utilization(left), Network In (Center), Network Out(Right)

scale up. The number of peaks in the CPU utilization plots are in proportion with the number of worker nodes assigned in each experiment, apart from the driver node which is active across all stages. These can be well observed in figures 6 and 7.

5 CONCLUSION

Apache Spark's ability to perform in-memory computation on Resilient Data across a cluster proved to increase the efficiency of the data analysis on big data. However, an important decision to be made while setting up a cluster is the number of worker nodes to be allotted in order to execute a query on the data. One needs to consider the trade

off between the computation cost, communication cost and resource cost. A perfect combination results in higher efficiency in terms of both time and money that is spent while performing data analysis on extremely large sizes of Data.

REFERENCES

[1] [n.d.]. ACM SIGSPATIAL Cup 2016 Problem Definition. http://sigspatial2016.sigspatial.org/giscup2016/problem

[2] Amazon. [n.d.]. *Amazon Cloudwatch*. https://aws.amazon.com/cloudwatch/

[3] Apache. [n.d.]. *Apache Hadoop*. https://hadoop.apache.org/

[4] Apache. [n.d.]. *Apache Spark*. https://spark.apache.org/

[5] databricks. [n.d.]. *Spark SQL*. https://databricks.com/glossary/what-is-spark-sql

[6] J Keith Ord and Arthur Getis. 1995. Local spatial autocorrelation statistics: distributional issues and an application. *Geographical analysis* 27, 4 (1995), 286–306.