Migrating GIS Big Data Computing from Hadoop to Spark: An Exemplary Study Using Twitter

Zhibo Sun*, Hong Zhang*, Zixia Liu*, Chen Xu†, and Liqiang Wang*
*Department of Computer Science, University of Central Florida, USA
†Department of Geography, University of Wyoming, USA
Email: {ericszb1987,kevinzhanghong,zxliuleo,carlxu2008,lwangcs}@gmail.com

***Abstract*—Recent research has demonstrated that social media could provide valuable spatio-temporal data about users activities. However, information extraction and computation from big amount of data pose various challenges. To effectively process massive datasets, several platforms have been developed. Our previous study [20] explored Hadoop-based cloud computing for processing big amount of social media data [9] to study geographic distributions of social media users.**

In this paper, we investigate an emerging system named Spark and present a timely pilot experience on geospatial big data research. In our study, Spark has been utilized to perform some classic geospatial analyses like K-Nearest Neighbors (KNN), geographic mean and median points, and the distribution of the median points. Our design is tested on an Amazon EC2 cluster. An exemplary study using 60GB, 120GB and 180GB Twitter data has demonstrated the performance achievements by migrating computing tasks from Hadoop to Spark. In our experiments, the Spark-based solution can be up to 2.3x faster than the Hadoop-based solution due to its in-memory processing and coarse-grained resource allocation strategy. In the paper, we also discuss optimization strategies on using Spark for different geospatial computing tasks.

***Keywords*-social media, GIS, Spark, Hadoop, KNN, centrographic analysis.;**

I. Introduction

Social media data increasingly attract research interests because they become proxy of peoples activities. By extracting locational and temporal information from social media data, trajectories of social media users daily lives can be plotted. Twitter is one of the most popular social media, which has more than 500 million users (302 million active users) around the world, who are generating hundreds of gigabyte (GB) text data per day [3, 9]. Hence, extracting and analyzing information from such big amount of data pose various challenges. In our previous study [20], we designed a Hadoop and Hbase based system, named Dart, to manage the massive data and process geospatial big data computing, which shows Hadoop-based solution has a significant performance improvement [7]. However the design of the Hadoop-based solution leaves ample room for further performance improvement, such as simplifying the two stages of Map and Reduce and replacing hard drive-based computation with in-memory processing. In this paper, we migrate our aforementioned Hadoop solution to a new platform based on an emerging big data system called Spark [2]. Our experimental results show that the Spark-based solution is up to 2.3x faster than our previous Hadoop-based solution.

Apache Hadoop is a reliable, scalable, and efficient cloud computing framework allowing for distributed processing of large datasets using Map-Reduce programming model [1]. However, it is a kind of disk-based computing framework, which writes all intermediate data to disk between Map tasks and Reduce tasks. In fact, even within a Map task, it also writes huge amount of intermediate data to disk repeatedly [1]. Since many applications in GIS involve lots of processing modules [22], writing too much intermediate data to disk will degrade the overall performance. Furthermore, releasing the container, applying for and initializing new containers also take a long time. Hence, after analyzing these features, we believe that another emerging technology, Spark, could be a better solution.

Apache Spark is a fast, reliable and distributed in-memory large-scale data processing framework. It takes advantage of the Resilient Distributed Dataset (RDD), which allows transparently storing data in memory and persisting it to disk only if it is needed [2, 18]. Hence it can reduce a huge number of disk writes and reads to outperform the Hadoop platform. Because Spark maintains status of assigned resources until a job is completed, Spark reduces time consumption in resource preparation and collection. Moreover, Hadoop is based on the Map-Reduce paradigm. For complex geospatial computations, a multi-pass Map-Reduce Chain, which consists of several Map-Reduce jobs, is commonly required. In contrast, multi-pass in Spark is easier to be deployed and it can reduce the number of writes and reads to disks. In another word, the more passes a computing task needs, the more significant differences can be observed between Spark-based and Hadoop-based systems, especially when the intermediate outputs between the passes are large. Furthermore, Spark is much better at handling iterative or interactive computations [18, 19], which makes Spark ideal for geospatial computations, because these kinds of tasks are common in GIS applications.

In our experiment, we store our data directly in Hadoop Distributed File System (HDFS), which is a Java-based open

2159-6190/16 $31.00 © 2016 IEEE
DOI 10.1109/CLOUD.2016.52

IEEE computer society

source and distributed file system designed to run on commodity hardware with low cost and high bandwidth [1, 21], as our file system. YARN, which has been characterized as a large-scale, distributed operating system, managing and coordinating all resources in the cluster for a big data application, is used to manage our cluster resource.

We conduct two studies in our experiments. In the first study, we compute KNN and geographic mean points using 60, 120 and 180 GB raw Twitter datasets. In the second study, geographic median points and their distributions of active users tweets are calculated. These centrographic analyses are popular measurements in geography [14] and have been used for illustrating social media users awareness about geographic places [16].

In our experiments, the Spark-based solution always outperforms the Hadoop-based by 50.17% while performing median point computation. The Spark-based solution could be 2.3x and 1.55x faster than using Hadoop system when performing KNN and Geographic Mean Point analysis, respectively. Pipeline model computation has significant better performance on Spark, while little improvement on Hadoop over the non-pipeline model.

II. Related Work

Apache Hadoop is a big data distributed processing framework implementing the Map-Reduce programming model [1]. Hadoop includes four modules: Hadoop Common, HDFS, YARN and Map-Reduce. It is a reliable, fault tolerant cloud computing framework, so more and more fields start to use it for processing massive data. In GIS, there are also some researchers deploying many geospatial analyses on Hadoop. Chen *et al.* design a MapReduce-based system, called MRGIS, for processing data-intensive applications in GIS [6]. [13] has applied Hadoop and Map-Reduce technology to GIS and provide performance evaluation to demonstrate its efficiency. Eldawy *et al.* [7] introduce a Hadoop-based system, SpatialHadoop, to perform GIS analysis, which adopts two-level index to organize the stored data. [4] shows another Hadoop-based system, Hadoop-GIS, which is a spatial data warehousing system. [23] implements some spatial data query analysis on Hadoop to achieve good performance. [5] uses the Map-Reduce model to solve two kinds of spatial problems and shows results on scalability and parallelism effect. In our previous research [20], we developed a Hadoop and Hbase based system, called Dart, to perform spatio-temporal analysis. In [20] we also designed an optimized grid-based geographic median point algorithm and provided some optimization suggestions when using Dart.

Apache Spark is a newly emerging distributed open-source in-memory computing framework [2]. It has lots of advantages over Hadoop framework, especially on iterative and interactive operations, fault tolerance and recovery capability [2, 18, 19]. Spark is not only a big data computing framework, but also supports stream processing, data fast query, machine learning and graphic processing[2], which suits almost all kinds of geospatial analyses. However, Spark has not been fully investigated for GIS applications. You *et al.* [17] conduct a geospatial analysis on Spark, but just study *join query processing* and compare the performance between Spark and Impala framework. Xie *et al.* [15] try to query data using R-tree and quad tree algorithms on Spark framework, and compare their performance. However, none of these research compare big data computation performance between Spark and Hadoop frameworks, technically explain the differences or provide any optimization suggestions on using Spark for GIS tasks.

III. Methodology

A. Geographic Mean Point

Geographic mean point calculation is to compute average latitude and longitude values to form a geographic position from all locations of one user. In our experiment, we ignore the projection effect of the mean point, since our dataset only covers a small area from New York City to Washington D.C. Equation 1 shows the calculation.

$$
\begin{aligned}
Mean_{lat} &= \sum_{i=1}^{n} Lat_i / n \\
Mean_{lon} &= \sum_{i=1}^{n} Lon_i / n
\end{aligned}
\tag{1}
$$

$Mean_{lat}$ and $Mean_{lon}$ correspond to mean point latitude and longitude values, respectively. Lat_i and Lon_i represent the latitude and longitude of point i, respectively.

Obtaining a mean point is a simple GIS analysis and the computation is not complex. The time-consuming parts in this application are reading all the raw data to the system and grouping these data based on users id.

In Hadoop, we only need to use single-pass to calculate the mean points. In the mapper phase, we read the raw unpreprocessed data and parse them into key value pairs. User id will be the key and location information, which consists of latitude and longitude, will be the value. All records with the same key will be sent to the same reducer. In the reducer phase, we calculate each users mean point. The algorithm in Hadoop is shown in Algorithm 1.

In the Spark-based solution, we import raw data to the cluster memory to create a RDD, and then parse the data. User id is the key, and location is the value. Next, we call *groupByKey()* to classify the data with respect to the user id then use Equation 1 to calculate the mean point of each user. numCore is the total number of cores in all executors. In order to have a better parallel performance and balance workload, we set partition number to numCore*2 when we perform *groupByKey*. Algorithm 2 shows how to implement it on Spark.

Algorithm 1 Mean on Hadoop

function: Map(k,v)
1: $newPoint(userId, Lat, Lon) = parse(inputdata)$
2: $EMIT(newPoint.userId, newPoint.position)$
function: Reduce(key,values)
3: $(lat, lon, num) = (0.0, 0.0, 0)$
4: **for** each value in values **do**
5: $lat+ = value.lat$
6: $lon+ = value.lon$
7: $num+ = 1$
8: **end for**
9: $(lat, lon) = (lat/num, lon/num)$
10: $results = lat + ";" + lon$
11: $EMIT(key, results)$

Algorithm 2 Mean on Spark

function: main
1: $points = sc.textFile(datasetPath).map(Parse(_))$
2: $results = points.groupByKey(numCore * 2)$
$.map(x => (x._1, calculateMean(x._2))).collect()$

x._1 represents the key, *i.e.* user id, and x._2 means all location information of one user. Collect is used to convert the final result from a RDD to an array and send it to the driver program for displaying.

B. K Nearest Neighbors

K Nearest Neighbors (KNN), a simple but very classic geospatial analysis, is a method for classifying objects based on the closest training examples with respect to some metrics such as distance. In our experiment, we plan to find k points whose distance to a specific point p are closest within a 60 GB Twitter Dataset. To obtain these points, we need to read entire dataset, and calculate the distance between the specific point to all other points. Then we sort all these distance values in an ascending order and finally get the first number of k sorted results in the order.

In our experiment, we assign 10 to k and (40, 70) to be the position of p. We use 60, 120 and 180 GB Twitter datasets as the input data. In the Hadoop system, we use two passes to perform this computation. In the first pass, the mapper is used to read the raw data and parse them then calculate the distance between point p to all other points. Next, distance is used as the key and the other points' latitude and longitude as the value. Since as long as the reducers exist, we have to do sort after map function in an ascending order, using distance as the key will improve the performance. In reducers, all records are sorted, so we just write first 10 records to the HDFS as the input data for the second pass. Since we can not ensure the results in one reducer are the smallest 10 values among all outputs, we need to do a global sorting. In the second pass, mappers are used to read data and because the distance is also the key of the data, after shuffling between the mapper and reducer, reducer could have a global sorted list of records. Then we output the first 10 records to finish the computation. The algorithm in Hadoop is shown in Algorithm 3.

In Spark, since we take advantage of RDD for storing the data, we can finish all computations in one pass and parallelize the computing across all executors. We firstly read the data to memory and parse the data to the correct location format, then use distance calculation function to calculate the distance, return distance as the key and point location as the value. Finally *sortByKey* function is called and the first 10 records are output. Algorithm 4 shows the implementation of KNN on Spark.

Algorithm 3 KNN on Hadoop

function: firstMap(k,v)
1: $newPoint(Lat, Lon) = parse(inputdata)$
2: $distance = calculateDistance(newPoint, specificPoint)$
3: $EMIT(distance, newPoint.position)$
function: firstReduce(key,values)
4: $k = 0$
5: **for** each value in values **do**
6: **while** k¡10 **do**
7: $EMIT(key, value)$
8: $k+ = 1$
9: **end while**
10: **end for**
function: secondMap(k,v)
11: $EMIT(k, v)$
function: secondReduce(key,values)
12: $k = 0$
13: **for** each value in values **do**
14: **while** k¡10 **do**
15: $EMIT(key, value)$
16: $k+ = 1$
17: **end while**
18: **end for**

Algorithm 4 KNN on Spark

function: main
1: $points = sc.textFile(datasetPath).map(Parse(_))$
2: $results = points.map(x =>$
$(calculateDistance(_, specificPoint), (x._1, x._2)))$
$.sortByKey().take(10)$

From Algorithm 4 we can see that coding in Spark is very simple and fast without having any gaps between passes. All data are stored and computed in the memory. The parameter "_" in `calculateDistance` represents all data in each record. x._1 and x._2 mean latitude and longitude values, respectively.

C. Geographic Median Point

A user's geographic median point is a point that minimizes the total distance to all of his/her points in the dataset. Usually, to show user's actual active location, median point position is more accurate than mean point position, since it is less influenced by outlier points. The algorithm of calculating the median point is complex and time consuming, which has lots of iterative operations. In our experiment, we use two algorithms to calculate the median point. In the first algorithm, we only use the midpoint as the initial point. In the second algorithm, we use midpoint as our first initial point then adopt our designed grid-based median point algorithm, which uses grids to find a better initial point[20]. Hence, in this paper, we directly deploy these two algorithms to perform median point computation. Our main idea of these optimized algorithms is to reduce the cost of finding a better initial point during the computation without losing the accuracy.

We only need a single Map-Reduce pass to perform this analysis in Hadoop. Mapper is used to read all users' raw data and parse the data into key-value pairs. In order to group the data by users, user id is still the key and the location information is the value. Then in the reducer, based on the algorithm selection indicator, we calculate the median point using Algorithm 5.

Our Spark algorithm is shown in Algorithm 6. We read and parse the input raw data then collect the generated RDD. Then we parallelize the collected result to make our processing better parallelized. After that, we use mapPartitions to calculate each user's median point. Finally we output the results.

Algorithm 5 MedianPoint on Hadoop

function: Map(k,v)
1: $newPoint(userId, Lat, Lon) = parse(inputdata)$
2: $EMIT(newPoint.userId, newPoint.position)$
function: Reduce(key,values)
3: $indicator = context.getConfiguration()$
$.get("algorithmIndicator")$
4: $currentPoint = calculateMidPoint(values)$
5: $distance = calculateTotalDistance(currentPoint, values)$
6: **if** $(indicator == gridAlgorithm)$ **then**
7: $currentPoint = updateCurrentPointByGrid$
$(currentPoint, distance, testStep)$
8: **end if**
9: **while** $(testStep > 0.00000002)$ **do**
10: $currentPoint = updateCurrentPoint$
$(currentPoint, distance, testStep)$
11: **end while**
12: $EMIT(key, currentPoint.position)$

Algorithm 6 MedianPoint on Spark

function: main
1: $points = sc.textFile(datasetPath).map(Parse(_))$
2: $tmp = points.groupByKey(numCore * 2)$
3: $result = tmp.mapPartitions$
$(getMedianPoint(_, algorithmIndicator)).collect()$

Algorithm 7 Distribution of MedianPoint on Hadoop

function: firstMap(k,v)
1: $newPoint(userId, Lat, Lon) = parse(inputdata)$
2: $EMIT(newPoint.userId, newPoint.position)$
function: firstReduce(key,values)
3: $currentPoint =$
$calculateMedianPointByUsingAlgorithm5$
4: $EMIT(currentPoint.latitude,$
$currentPoint.longitude)$
function: secondMap(k,v)
5: $newPoint(Lat, Lon) = parse(inputdata)$
6: $id = calculateDistribution(newPoint.position)$
7: $EMIT(id, 1)$
function: secondReduce(key,values)
8: $count = 0$
9: **for** each value in values **do**
10: $count+ = value$
11: **end for**
12: $EMIT(key, count)$

D. Geographic Median Point Distribution

Geographic median point distribution is a very useful and important geospatial analysis in GIS, since it could tell us our interested type of Twitter users' actual spatial distribution, which can be used for further commercial or criminal analysis. We perform this computation in two models: Chain and No-Chain. In the Chain model, users' median points are unavailable, thus calculation of median positions from raw data is needed, and then distribution algorithm is applied to compute the distribution pattern. The No-Chain model assumes that users' median points are available, hence we just need to consequently read these data to the system then compute distribution pattern. Both models are used to study multi-pass situation performance in Hadoop and Spark. For the distribution algorithm, we implement it in the following way: firstly we create a mesh on dataset covered area, with 0.01 degree grid length, and then compute in which grid each median point locates.

In this experiment, we try to analyze active users' location patterns from New York to Washington D.C. In the Chain model, we adopt both of the algorithms demonstrated in Section III-C to perform the computation.

Algorithm 7 introduces the Chain model analysis. In the first pass, mapper is used to read and parse the raw data. Then based on the key, we send all data of the same user

to the same reducer. Reducer then calculates users' median point positions and writes them to HDFS. In the second pass, the written data is read into the system by mappers, mappers then calculate the grid-id that it belongs to and write the key-value pair intermediate data to HDFS. Here, the key is the grid-id and the value is 1. In the reducer phase, we count the number of values in each grid.

In Spark, basically we only add one line of code after the Algorithm 6, which is shown in Algorithm 8.

Algorithm 8 Distribution of MedianPoint on Spark

function: main

1: $points = sc.textFile(datasetPath).map(Parse(_))$
2: $tmp = points.groupByKey(numCore * 2)$
3: $medianPoint = tmp.mapPartitions$
$(getMedianPoint(_, algorithmIndicator))$
4: $result = medianPoint.mapPartitions$
$(checkPointLocation).map(x =>$
$(x, 1)).reduceByKey(_ + _).collect$

Here, we will not show the algorithm of No-Chain model, since it is just from Step 5 to Step 12 in Algorithm 7 and the fourth step in Algorithm 8.

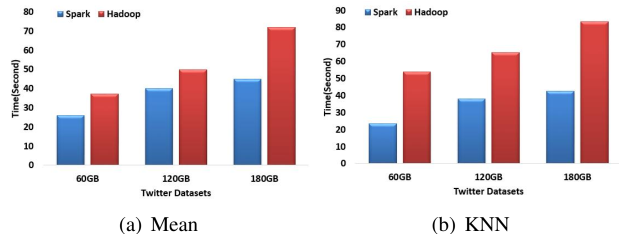
IV. Experiments and Optimizations

Our experiments are conducted on an Amazon EC2 cluster, which consists of 11 m3.xlarge nodes, including one namenode and 10 datanodes. Each m3.xlarge instance has Intel Xeon E5-2670 v2 (Ivy Bridge) Processors, 4 vCPU, 15 GB memory, and 100 GB magnetic storage, running Centos 6. Our experiments are based on Apache Hadoop 2.4, Spark 1.1.1, and Java 6.

A. Study I: Geospatial Big Data Processing

We use 3 different sizes of Twitter datasets, 60, 120 and 180 GB, to perform our two geospatial analyses, KNN and Mean. Our purpose is to compare the performance of Spark and Hadoop on computing spatial operations on real big datasets.

1) Experiment on Mean: Spark has a better performance than Hadoop-based solution on computing the geographic mean point of each user according to Figure 1(a). Our study indicates that Spark can improve performance up to 34.43% and averagely by 21.45%. In fact, we do not need to do sorting while processing mean point analysis, but the Hadoop-based solution involves several sorting operations. In contrast, this step does not exist in the Spark-based solution. Moreover, because the dataset is massive, writing and reading data to disk and shuffling the data to the reducers may take long time in Hadoop. Thus Spark-based has a better performance than Hadoop.



(a) Mean (b) KNN

Figure 1. Performance of Mean and KNN on Spark and Hadoop

2) Experiment on KNN: Figure 1(b) shows the time of performing KNN analysis when k is 10 on Spark and Hadoop system. It shows that our Spark-based solution has a much better performance than the Hadoop-based solution. According to our experiments, KNN on Spark is 2.3x, 1.6x and 1.8x faster than Hadoop on 60, 120, 180 GB datasets, respectively. Since the computation is not complicated, we believe the major reason causes Spark-based is faster than Hadoop-based is the multi-pass, since massive intermediate data has been written to disk 3 times in Hadoop-based solution, while 0 time in Spark-based solution.

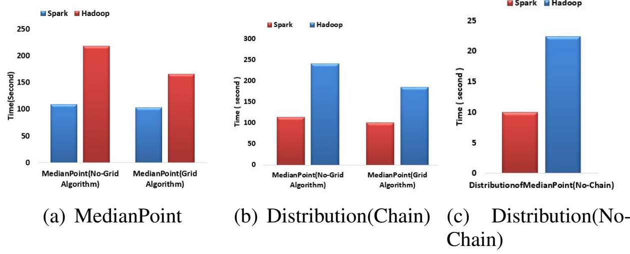
B. Study II: Geospatial Data Analysis

In this section, we compute geographic median points and their distribution of active Twitter users who posted more than 500 tweets in 3 months within 60 GB Twitter dataset covering an area from New York City to Washington D.C. In our experiment dataset, there are 2434 active users, which is around 0.6% of all users. We are interested in only active users, who involve a larger amount of raw data and could be better cases to test our system.

1) Experiment on Geographic Median Point: Figure 2(a) demonstrates that Spark still outperforms the Hadoop-based solution on this complex analysis no matter what type of algorithm we use. Hadoop involves unnecessary sorting process, and has a large number of writes and reads to disks. In contrast, Spark takes full advantage of the cluster resource and performs all tasks in memory. Such kind of jobs contain considerable number of iterative operations, which Spark is good at. Our experiments indicate that the Spark-based solution improves performance by 50.17% compared to the Hadoop-based solution using no-grid algorithm, and by 38.18% using grid-based solution.

2) Experiment on Distribution of Median Point: Figure 2(b) indicates that in the Chain model, no matter what algorithms we use, Spark outperforms Hadoop. Spark can be 2.12x and 1.85x faster for no-grid algorithm and grid algorithm, respectively. If we implement No-Chain model, the Spark-based solution is 2.24x faster than using Hadoop system, as shown in Figure 2(c).

Figure 3 shows the influence of computing distribution of median point in pipeline type and non-pipeline type, since we want to know when only the raw data is available,



(a) MedianPoint (b) Distribution(Chain) (c) Distribution(No-Chain)

Figure 2. Performance of MedianPoint and Distribution of Median Point on Spark and Hadoop

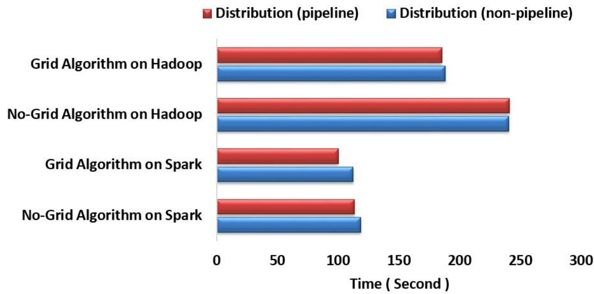
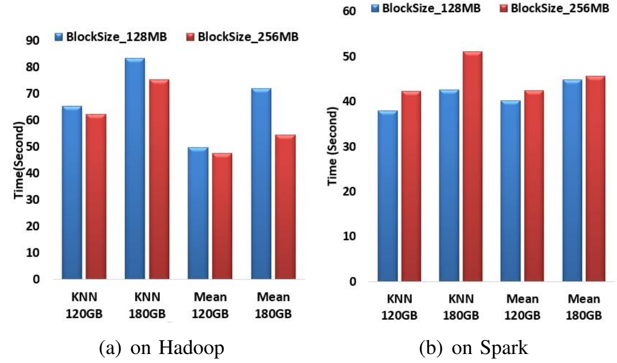


Figure 3. Performance of Distribution of Median Point in Chain vs No-Chain

how much the influence it will be if we use separate jobs with single pass in each (non-pipeline type) instead of using one job with multi-pass (pipeline type) by using the same framework. The time of Distribution (non-pipeline) is the total time that includes the time of calculating the median point distribution from raw data and the time of computing the distribution by using No-Chain model. Figure 3 shows that there is little performance difference in Hadoop between pipeline type and non-pipeline type on processing distribution analysis. Results indicate that if adopting grid algorithm in Hadoop, the pipeline model only improves performance by 1% compared to non-pipeline. The performance of pipeline model using no-grid algorithm in Hadoop is almost the same compared to the non-pipeline model (the improvement is only 0.2%). In contrast, Spark pipeline model always outperforms non-pipeline model, and the improvements are 4.3% and 11% for no-grid algorithm and grid algorithm, respectively. The experiment shows that there is no significant difference between pipeline and non-pipeline computation type in Hadoop, while the pipeline model has an obvious performance improvement over the non-pipeline model in Spark. We believe that the primary reason is all data are stored in RDD. If we use non-pipeline in Spark, we have to read data from disk twice and write once, while using pipeline model, data are stored in memory and we just need to read the raw data from disk once. In contrast, Hadoop is a kind of disk-based system, no matter what type of model we use, it has to write the data to disk. Thus, there is not too much difference between these two



(a) on Hadoop (b) on Spark

Figure 4. Performance of KNN and Mean on Hadoop and Spark with Different Block Size

models. However, we also notice that the pipeline model has a little better performance than the non-pipeline model in Hadoop, because the pipeline model can decrease the initialization time of the pass. For example, in the second pass, we do not need to apply for, launch, and initialize the application master container, but we think the improvement of the pipeline type performance is limited. If the analysis is more complicated with more passes and last longer time, this improvement could be ignorable.

Through our experiments, we notice that our experiment results are not as fast as what Spark official website advocates. We believe the reasons are: (1). Our datasets are still not big enough. Because of small size data sets, Spark computational capacity cannot be fully demonstrated. If the datasets are big and computation is more complicated, we believe the performance will be further improved. (2). We do not have too many iterative operations or re-use large amount of medium data. However, our experiment results show that without catering for preferred operation type, Spark still can have a higher performance than Hadoop-based solution in GIS.

C. Optimizations

1) Optimization Suggestions on HDFS: There are several aspects needing attentions while coding or using Spark. Firstly, the block size in HDFS is critical since it can affect the number of partitions in Spark and mappers in Hadoop. After analyzing and comparing the performance of different block sizes, we notice that 128 MB block size in Spark and 256 MB in Hadoop have the best performance in our experiments. The performance difference caused by block size usually appears in map phase. Because the total size of data is fixed and the number of reducers is the same, the computation time in reducers will not change a lot in Hadoop. In our experiment, mapper is used to read data. When processing big datasets, we need multi-wave to read data. So after reading one block of data, the mapper container will be released and the application master will be

notified in the next heart-beat. Then the application master will request a new mapper container, wait for reply from the resource manager, send request to an assigned node manager, and then launch a new mapper container. After that, we need to initialize the new container, and read the correct block of data. If the data is not localized, it has to be fetched from another node. Hence, bigger block size means less number of steps. So in Hadoop, the bigger block size, 256 MB, could be better. However, larger size will degrade the parallelism. On the contrary, in Spark, all executors will keep running until the whole job is finished, which is a kind of container reuse mechanism. After reading a block of data, it immediately starts to read the next block. Hence reading data does not affect the time too much. But in each executor, after reading data, it immediately starts to process the data. Bigger size of data block in each executor will need longer time to process, if the computation time grows linearly with the input data, the block size will not affect performance obviously, like in Mean Point computation. But if the computation growth is complex like KNN, the affection will be significant. Figure 4(a) shows results of running KNN and Mean analysis on Hadoop with different HDFS block. Figure 4(b) shows the results on Spark.

2) Optimization Suggestions on Data Operations: For Geospatial operations that involve multiple operations on each record and especially when the number of record is large, it is better to use mapPartitions to replace map to perform operations. Because mapPartitions makes each partition call the function once, thus the number of calling is reduced and time for function initialization is shortened. This advantage becomes even more significant especially when the number of records is large, the computation is complex, or lots of objects need to be initialized in each function. In our experiment, using mapPartitions does not improve too much performance compared to using map (only averagely being improved by 4 seconds), because the computation in each method is not very complex and only a few objects need to be initialized. However, based on our experience, mapPartitions sometimes can improve speed more than 20%. While using mapPartitions, it is better to set Spark.akka.frameSize with a large value to avoid errors and improve the performance. In addition, adopting user-defined serializer to replace JavaSerializer can improve the serialization and data transfer speed by setting Spark.serializer. Garbage collection (GC) really takes a long time in many tasks, so if the block size is small, we can assign smaller space for RDD to decrease the number of GC and its time by setting Spark.storage.memoryFraction. Similar to Hadoop, shuffle in Spark also takes a long time. If we have many files needing to be shuffled, setting Spark.shuffle.consolid be true can be a good option, since it can decrease the number of shuffle handlers and files.

3) Optimization Suggestions on Resources: When adopting Spark to perform geospatial analysis, more attentions need to be paid on utilizing cluster resource. Because Spark on Yarn uses a coarse-grained model, no matter whether a task is running in the executor or not, the executor will keep the resource until the job is finished. In many cases, if parallelization is not fully explored, tasks may only run in some of the executors and waste resources. For example, when performing geographic median point analysis, it is good to parallelize the computation by setting partition numbers in shuffle function (i.e. groupByKey), otherwise it may produce less partitions using hashPartitioner, or make workload imbalance.

In the yarn-site.xml, it is better to set yarn.scheduler.minimum-allocation-mb to a small value, such as 100, to fully take advantage of the resource, since memory size of each container is multiple times of the minimum container memory size. For example, if a node has 3 cores and 5 GB memory, and the default minimum container memory size is 1GB, when running 2 executors and setting 2 GB memory in each executor, only one container can be observed running in this node and its memory size is 3 GB. The reason is that there are some executor memory overheads in each container, so actually the container size should be a little larger than 2 GB. However, if the minimum container memory size is 1GB and container memory size must be multiple times of the minimum size, then the container memory size has to be 3 GB and only one container could run in this node under this condition.

V. Discussion

The performance of Spark-based solution outperforms the Hadoop solution, but we also notice that there are several disadvantages of using Spark. The first one is that Spark on YARN uses a coarse-grained model. Of course this model can improve the performance of Spark when the dataset is big and computation is complex, since we do not need to re-apply for new containers. However, if there are more than one job running at the same time, this kind of model will waste lots of resource, which may make other jobs delay a long time or even killed. Secondly, it is hard to debug the application. Spark uses a lazy operation mechanism, which will not start to run the operations until an action operation is called [18], so if the processing is slow, it is hard to detect which transformation operation is the bottleneck. Finally, using Spark framework may put more costs on physical nodes, since it is an in-memory computing platform and needs larger memory compared to Hadoop cluster.

VI. Conclusion and Future Work

In this paper, we implement four classic geospatial analyses, KNN, geographic mean point and median point, and distribution of median point on 60, 120 and 180 GB Twitter datasets on Hadoop and Spark systems. The experimental

results show that the Spark-based solution always outperforms the Hadoop-based on these 4 analysis. Pipeline model computation has significant better performance on Spark, while little improvement on Hadoop over the non-pipeline model. We also discuss potential problems in Hadoop and technical reasons of better performance for Spark. In the end, we provide several optimization suggestions on using Spark.

In the future work, we will extend our systems on Hadoop and Spark to support more GIS computations for large-scale datasets as well as automated GIS scientific workflows [12]. In addition, we plan to adapt our optimization techniques on scientific computations on EC2 [8] and Azure [10, 11] to improve our system's performance and reduce cost on public cloud platforms.

Acknowledgment

This work was supported in part by NSF-CAREER-1622292 and NSFC-61428201.

References

[1] Apache hadoop. https://hadoop.apache.org/.

[2] Apache spark. https://spark.apache.org/.

[3] Twitter on wikipedia. https://en.wikipedia.org/wiki/Twitter.

[4] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop-GIS: A high performance spatial data warehousing system over mapreduce. In *Proceedings of the VLDB Endowment*, 2013.

[5] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on processing spatial data with mapreduce. In *Proceedings of the 21st International Conference on Scientific and Statistical Database Management*, 2009.

[6] Q. Chen, L. Wang, and Z. Shang. Mrgis: A mapreduce-enabled high performance workflow system for gis. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, 2008.

[7] A. Eldawy and M. Mokbel. SpatialHadoop: towards flexible and scalable spatial processing using mapreduce. In *Proceedings of the 2014 SIGMOD*, 2014.

[8] H. Huang, L. Wang, B. C. Tak, L. Wang, and C. Tang. Cap3: A cloud auto-provisioning framework for parallel processing using on-demand and spot instances. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, 2013.

[9] J. Lin and D. Ryaboy. Scaling big data mining infrastructure: The twitter experience. In *ACM SIGKDD Explorations Newsletter*, 2012.

[10] V. Subramanian, H. Ma, L. Wang, E.-J. Lee, and P. Chen. Rapid 3d seismic source inversion using windows azure and amazon ec2. In *Proceedings of the 2011 IEEE World Congress on Services*. IEEE, 2011.

[11] V. Subramanian, L. Wang, E.-J. Lee, and P. Chen. Rapid processing of synthetic seismograms using windows azure cloud. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, 2010.

[12] L. Wang, S. Lu, X. Fei, A. Chebotko, H. V. Bryant, and J. Ram. Atomicity and provenance support for pipelined scientific workflows. *Journal of Future Generation Computer Systems*, 25(5):568–576, 2009.

[13] Y. Wang and S. Wang. Research and implementation on spatial data storage and operation based on hadoop platform. In *2010 Second IITA-GRS International Conference*.

[14] D. Wong and J. Lee. *Statistical Analysis and Modeling of Geographic Information*. John Wiley and Sons, 2005.

[15] X. Xie, Z. Xiong, X. Hu, G. Zhou, and J. Ni. On massive spatial data retrieval based on spark. In *Web-Age Information Management*. Springer, 2014.

[16] C. Xu, D. W. Wong, and C. Yang. Evaluating the geographical awareness of individuals: An exploratory analysis of twitter data. *Cartography and Geographic Information Science*, 2013.

[17] S. You, J. Zhang, and L. Gruenwald. Large-scale spatial join query processing in cloud. In *Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on*. IEEE, 2015.

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[19] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[20] H. Zhang, Z. Sun, Z. Liu, C. Xu, and L. Wang. Dart: A geographic information system on hadoop. In *Proceedings of 2015 IEEE 8th International Conference on Cloud Computing*. IEEE, 2015.

[21] H. Zhang, L. Wang, and H. Huang. Smarth: Enabling multi-pipeline data transfer in hdfs. In *Proceedings of 2014 43rd International Conference on Parallel Processing*. IEEE, 2014.

[22] J. Zhang. Towards personal high-performance geospatial computing (hpc-g): Perspectives and a case study. In *Proceedings of the ACM SIGSPATIAL International Workshop on High Performance and Distributed Geographic Information Systems*, 2010.

[23] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng. Spatial queries evaluation with mapreduce. In *Proceedings of the 2009 Eighth International Conference on Grid and Cooperative Computing*, 2009.