A Demonstration of GeoSpark: A Cluster Computing Framework for Processing Big Spatial Data

Jia Yu
School of Computing, Informatics,
and Decision Systems Engineering
Arizona State University
Tempe, Arizona 85281
Email: jiayu2@asu.edu

Jinxuan Wu
School of Computing, Informatics,
and Decision Systems Engineering
Arizona State University
Tempe, Arizona 85281
Email: jinxuanw@asu.edu

Mohamed Sarwat
School of Computing, Informatics,
and Decision Systems Engineering
Arizona State University
Tempe, Arizona 85281
Email: msarwat@asu.edu

***Abstract*—This paper demonstrates GEOSPARK a cluster computing framework for developing and processing large-scale spatial data analytics programs. GEOSPARK consists of three main layers: Apache Spark Layer, Spatial RDD Layer and Spatial Query Processing Layer. Apache Spark Layer provides basic Apache Spark functionalities as regular RDD operations. Spatial RDD Layer consists of three novel Spatial Resilient Distributed Datasets (SRDDs) which extend regular Apache Spark RDD to support geometrical and spatial objects with data partitioning and indexing. Spatial Query Processing Layer executes spatial queries (e.g., Spatial Join) on SRDDs. The dynamic status of SRDDs and spatial operations are visualized by GEOSPARK monitoring map interface. We demonstrate GEOSPARK using three spatial analytics applications (spatial aggregation, autocorrelation and co-location) to show how users can easily define their spatial analytics tasks and efficiently process such tasks on large-scale spatial data at interactive performance.**

I. INTRODUCTION

Spatial data includes but is not limited to: weather maps, geological maps, socioeconomic data, vegetation indices, and more. Moreover, novel technology allows hundreds of millions of users to use their mobile devices to access their healthcare information and bank accounts, interact with friends, buy stuff online, search interesting places to visit on-the-go, ask for driving directions, and more. In consequence, everything we do on the mobile internet leaves breadcrumbs of spatial digital traces, e.g., geo-tagged tweets, venue check-ins. Making sense of such spatial data will be beneficial for several applications that may transform science and society – For example: (1) Socio-Economic Analysis: that includes for example climate change analysis, study of deforestation, population migration, and variation in sea levels, (2) Urban Planning: assisting government in city/regional planning, road network design, and transportation / traffic engineering, and (3) Commerce and Advertisement: e.g., point-of-interest (POI) recommendation services. The aforementioned applications need a powerful data management platform to handle the large volume of spatial data such applications deal with. Challenges to building such platform are as follows:

- **Challenge I: System Scalability.** The massive-scale of available spatial data hinders making sense of it using traditional spatial database management systems. Moreover, large-scale spatial data, besides its tremendous storage footprint, may be extremely difficult to manage and maintain. The underlying database system must be able to digest Petabytes of spatial data and effectively analyze it.
- **Challenge II: Fast Analytics.** In spatial data analytics applications, users will not tolerate delays introduced by the underlying spatial database system. Instead, the user needs to see useful information quickly. Hence, the underlying spatial data processing system must figure out effective ways to execute spatial analytics in parallel.

Existing spatial database systems extend relational database systems with new data types, functions, operators, and index structures to handle spatial operations based on the Open Geospatial Consortium. Even though such systems sort of provide full support for spatial data storage and access, they suffer from a scalability issue. Based upon a relational database system, such systems are not scalable enough to handle large-scale analytics over big spatial data. Recent works (e.g., [1], [2]) extend the Hadoop ecosystem to perform spatial analytics at scale. The Hadoop-based approach indeed achieves high scalability. However, these systems though exhibit excellent performance in batch-processing jobs, they show poor performance handling applications that require fast data analysis. Apache Spark [3], on the other hand, is an in-memory cluster computing system. Spark provides a novel data abstraction called resilient distributed datasets (RDDs) [4] that are collections of objects partitioned across a cluster of machines. Each RDD is built using parallelized transformations (filter, join or groupBy) that could be traced back to recover the RDD data. In memory RDDs allow Spark to outperform existing models (MapReduce) by up to two orders of magnitude. Unfortunately, Spark does not provide native support for spatial data and spatial operations. Hence, users need to perform the tedious task of programming their own spatial data processing jobs on top of Spark.

This paper demonstrates GEOSPARK [1] [5] an in-memory cluster computing system for processing large-scale spatial data. GEOSPARK extends Apache Spark to support spatial data types and operations. In other words, the system extends the resilient distributed datasets (RDDs) concept to support spatial data. This problem is quite challenging due to the fact that (1) spatial data may be quite complex, e.g., rivers' and cities'

[1]GeoSpark Github repository: https://github.com/Sarwat/GeoSpark

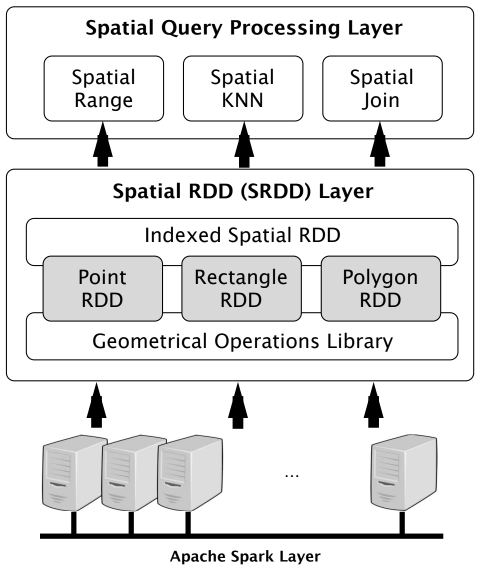


Fig. 1: GeoSpark architecture

geometrical boundaries, (2) spatial (and geometric) operations (e.g., Overlap, MinimumBoundingRectangle, Union) cannot be easily and efficiently expressed using regular RDD transformations and actions. GEOSPARK extends RDDs to form Spatial RDDs (SRDDs) and efficiently partitions SRDD data elements across machines and introduces novel parallelized spatial (geometric operations that follows the Open Geosptial Consortium (OGC) [6] standard) transformations and actions (for SRDD) that provide a more intuitive interface for users to write spatial data analytics programs. Moreover, GEOSPARK extends the SRDD layer to execute spatial queries (e.g., Range query, KNN query, and Join query) on large-scale spatial datasets. The dynamic status of SRDDs and associated queries are visualized by GEOSPARK monitoring interface throughout each entire spatial analytics process. We demonstrate GEOSPARK using three applications: (1) Application 1 uses GEOSPARK to calculate geospatial autocorrelation in a spatial dataset, (2) Application 2 leverages the system to generate a heat map of the San-Francisco trees population, and (3) Application 3 executes a spatial co-location pattern mining with the help of GEOSPARK .

II. GEOSPARK ARCHITECTURE

As depicted in Figure 1, GEOSPARK consists of three main layers: (1) *Apache Spark Layer:* that consists of regular operations that are natively supported by Apache Spark. These native functions are responsible for loading / saving data from / to persistent storage (e.g., stored on local disk or Hadoop file system HDFS). (2) *Spatial Resilient Distributed Dataset (SRDD) Layer* (Section II-A). (3) *Spatial Query Processing Layer* (Section II-B).

A. Spatial RDD (SRDD) Layer

This layer extends Spark with spatial RDDs (SRDDs) that efficiently partition SRDD data elements across machines and introduces novel parallelized spatial transformations and actions (for SRDD) that provide a more intuitive interface for

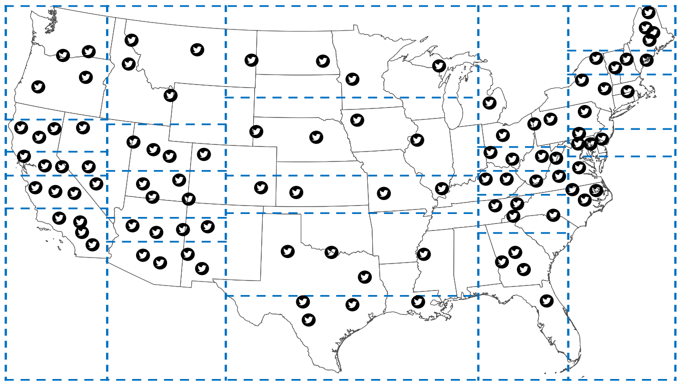


Fig. 2: SRDD partitioning

users to write spatial data analytics programs. The SRDD layer consists of three new RDDs: PointRDD, RectangleRDD and PolygonRDD. One useful Geometrical operations library is also provided for every spatial RDD.

Spatial Objects Support. GEOSPARK supports various spatial data input format (e.g., Comma Separated Value, Tab Separated Value and Well-Known Text). Each type of spatial objects is stored in a SRDD, PointRDD, RectangleRDD or PolygonRDD. GEOSPARK provides a set of geometrical operations which is called Geometrical Operations Library. This library natively supports geometrical operations. For example, `Overlap()`: Finds all of the internal objects which are intersected with others in geometry; `MinimumBoundingRectangle()`: Finds the minimum bounding rectangles for each object in a Spatial RDD or return a large minimum bounding rectangle which contains all of the internal objects in a Spatial RDD; `Union()`: Returns the union polygon of all polygons in this RDD.

SRDD Partitioning. GEOSPARK automatically partitions all loaded Spatial RDDs by creating one global grid file for data partitioning. The main idea for assigning each element in a Spatial RDD to the same 2-Dimensional spatial grid space is as follows: Firstly, split the spatial space into a number of non-equal grid cells which compose a global grid file. This global grid file has load balanced grids according to pre-sampling techniques. Then traverse each element in the SRDD and assign this element to a grid cell if the element overlaps with this grid cell. If one element intersects with two or more grid cells, then duplicate this element and assign different grid IDs to the copies of this element. Figure 2 depicts tweets in the U.S. at a particular moment, tweets and states are assigned to respective grid cells.

SRDD Indexing. Spatial indexes like Quad-Tree and R-Tree are provided in Spatial IndexRDDs which inherit from Spatial RDDs. Users are able to initialize a Spatial IndexRDD. Moreover, GEOSPARK adaptively decides whether a local spatial index should be created for a certain Spatial IndexRDD partition based on a tradeoff between the indexing overhead (memory and time) on one-hand and the query selectivity as well as the number of spatial objects on the other hand.

B. Spatial Query Processing Layer

This layer supports spatial queries (e.g., Range query and Join query) for large-scale spatial datasets. After geometrical objects are stored and processed in the Spatial RDD layer, user may invoke a spatial query provided in Spatial Query Processing Layer. GEOSPARK processes such query and returns the final results to the user. GEOSPARK execution model implements the algorithms proposed by [7] and [8]. To accelerate a spatial query, GEOSPARK leverages the grid partitioned Spatial RDDs, spatial indexing, the fast in-memory computation and DAG scheduler of Apache Spark to parallelize the query execution.

Spatial Range Query. GEOSPARK executes the spatial range query algorithm following the execution model: Load target dataset, partition data, create a spatial index on each SRDD partition if necessary, broadcast the query window to each SRDD partition, check the spatial predicate in each partition, and remove spatial objects duplicates that existed due to the data partitioning phase.

Spatial Join Query. GEOSPARK executes the parallel spatial join query following the execution model. GeoSpark first partitions the data from the two input SRDDs as well as creates local spatial indexes (if required) for the SRDD which is being queried. Then it joins the two datasets by their keys which are grid IDs. For the spatial objects (from the two SRDDs) that have the same grid ID, GeoSpark calculates their spatial relations. If two elements from two SRDDS are overlapped, they are kept in the final results. The algorithm continues to group the results for each rectangle. The grouped results are in the following format: Rectangle, Point, Point, ... Finally, the algorithm removes the duplicated points and returns the result to other operations or saves the final result to disk.

Spatial KNN Query. To process a Spatial KNN query, GEOSPARK uses a heap based top-k algorithm[9], which contains two phases: selection and merge. It takes a partitioned SRDD, a point P and a number k as inputs. To calculate the k nearest objects around point P, in the selection phase, for each SRDD partition GEOSPARK calculates the distances between each object to the given point P, then maintains a local heap by adding or removing elements based on the distances. This heap contains the nearest k objects around the given point P. For IndexedSRDD, the system can utilize the local indexes to reduce the query time. After the selection phase, GEOSPARK merges results from each partition, keeps the nearest k elements that have the shortest distances to P and outputs the result.

III. DEMONSTRATION SCENARIOS

We demonstrate GEOSPARK using three spatial applications which are described below. GEOSPARK provides a monitoring map interface for system users to visualize and monitor the spatial program dynamically. A screenshot of this tool is provided in Figure 3. The interface allows users to execute Scala code interactively through an integrated Scale shell. Meanwhile, a map on-top of the shell visualizes the SRDDs generated by Scala code. Throughout the entire spatial analytics process, all generated SRDDs are listed on the left side pane of the user interface. When the user clicks on any SRDD partition (if it is still alive) on the left pane, she obtains more detailed information from a nested menu such as the data size in this partition, physical machine IP address, CPU and memory utilization. Besides the description of SRDDs, the tool also provides the status of a running spatial program in a progress bar format. By browsing GEOSPARK Monitoring Tool, users can interactively monitor the run time of their entire spatial analytics program.

A. Application 1: Spatial Autocorrelation

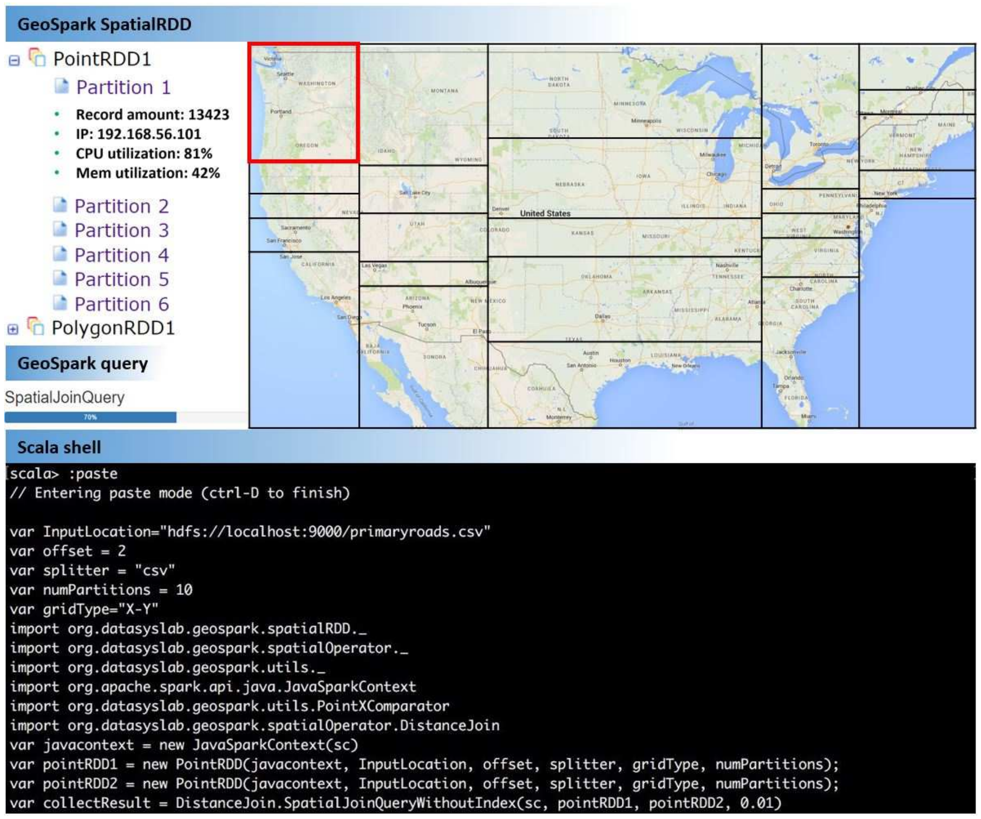
Spatial autocorrelation studies whether neighbor spatial data points might have correlations in some non-spatial attributes. Moran's I and Geary's C are two common coefficients in spatial autocorrelation. Based on them, analysts can determine whether these objects influence each other. These efficients are defined by two specific formulas correspondingly. An important part of these formulas is to find the spatial adjacent matrix. In this matrix, each tuples stands for whether two objects, such as points, rectangles or polygons, are within a specified distance.

An application programmer may leverage GEOSPARK `SpatialJoinQuery()` to calculate the spatial adjacent matrix. Assume one dataset is composed of millions of point objects. The process to find the global adjacent matrix in GEOSPARK is as as follows: (1) Call GEOSPARK PointRDD initialization method to store the dataset in memory. Data partitioning and indexing are also completed by GEOSPARK at this stage. (2) Call GEOSPARK `SpatialJoinQuery()` in PointRDD. The first parameter is the query point set itself and the second one is the specified distance. (3) Use a new instance of Spatial PairRDD to store the result of Step (2). Step (2) will return the whole point set which has a new column specify the neighbors of each tuple within the distance. The expected schema is like this: Point coordinates (longitude, latitude), neighbor 1 coordinates (longitude, latitude), neighbor 2 coordinates (longitude, latitude), ... (4) Call persistence method in GEOSPARK to persist the resulting PointRDD.

B. Application 2: Spatial Aggregation

Assume an environmental scientist – studying the relationship between air quality and trees – would like to explore the trees population in San Francisco. A query may leverage the `SpatialRangeQuery()` provided by GEOSPARK to just return all trees in San Francisco. Alternatively, a heat map (spatial aggregate) that shows the distribution of trees in San Francisco may be also helpful. This spatial aggregate query (i.e., heat map) needs to count all trees at every single region over the map.

In the heat map case, in terms of spatial queries, the heat map is a spatial join in which the target set is the tree map in San Francisco and the query area set is a set of San Francisco regions. The region number depends on the display resolution, or granularity, in the heat map. One proper GEOSPARK program is as follows: (3) Use a Spatial PairRDD to store the result of Step (2) which is the count for each polygon. The Spatial PairRDD follows the schema like this: (Polygon, count) such that Polygon represents the boundaries of the spatial region. (4) Call persistence method in Spark to persist the result PolygonRDD.



```
scala> :paste
// Entering paste mode (ctrl-D to finish)

var InputLocation="hdfs://localhost:9000/primaryroads.csv"
var offset = 2
var splitter = "csv"
var numPartitions = 10
var gridType="X-Y"
import org.datasyslab.geospark.spatialRDD._
import org.datasyslab.geospark.spatialOperator._
import org.datasyslab.geospark.utils._
import org.apache.spark.api.java.JavaSparkContext
import org.datasyslab.geospark.utils.PointXComparator
import org.datasyslab.geospark.spatialOperator.DistanceJoin
var javacontext = new JavaSparkContext(sc)
var pointRDD1 = new PointRDD(javacontext, InputLocation, offset, splitter, gridType, numPartitions);
var pointRDD2 = new PointRDD(javacontext, InputLocation, offset, splitter, gridType, numPartitions);
var collectResult = DistanceJoin.SpatialJoinQueryWithoutIndex(sc, pointRDD1, pointRDD2, 0.01)
```

Fig. 3: GeoSpark Monitoring Tool

C. Application 3: Spatial Co-location

Spatial co-location is defined as two or more species are often located in a neighborhood relationship. The determination of this co-location pattern may benefit many further scientific researches. Biologists may find symbiotic relationships, mobile carriers can provide proper plans based users' co-location, and advertising agencies are able to place directed advertisements at the center of co-located populations. For instance, one existing co-location pattern is that one kind of tigers always live within a certain distance from one kind of rabbits. Thus we may infer one possible fact that these tigers feed on these rabbits.

Some co-efficients are applied to determine the co-location relationship. Ripley's K function [10] is the most common one in real life. It usually executes numerous times iteratively and finds the ideal distance. The calculation of K function also needs the adjacent matrix between two type of objects. As we mentioned in spatial autocorrelation analysis, seeking adjacent matrix may leverage GeoSpark `SpatialJoinQuery()`. Programmer are able to follow the same procedure depicted in Spatial Autocorrelation.

Furthermore, spatial co-location, different from the previous basic spatial applications, is able to maximize the in memory computation goodness of GeoSpark . Under GeoSpark framework, users only need to spend time on loading data, partitioning data, and constructing indexes in the first iteration and then GeoSpark automatically caches these intermediate data in memory. In the next numerous iterations, users are able to directly keep mining the co-location pattern using the cache in memory instead of loading and pre-processing data from scratch.

References

[1] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. H. Saltz, "Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce," *Proceedings of the VLDB Endowment, PVLDB*, vol. 6, no. 11, pp. 1009–1020, 2013.

[2] A. Eldawy and M. F. Mokbel, "A demonstration of spatialhadoop: An efficient mapreduce framework for spatial data," *Proceedings of the VLDB Endowment, PVLDB*, vol. 6, no. 12, pp. 1230–1233, 2013.

[3] "Spark," https://spark.apache.org.

[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2012, pp. 15–28.

[5] J. Yu, J. Wu, and M. Sarwat, "Geosaprk: A cluster computing framework for processing large scale spatial data," in *Proceedings of ACM SIGSPATIAL GIS*, 2015.

[6] "Open Geospatial Consortium," http://www.opengeospatial.org/.

[7] G. Luo, J. F. Naughton, and C. J. Ellmann, "A non-blocking parallel spatial join algorithm," in *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE, 2002, pp. 697–705.

[8] X. Zhou, D. J. Abel, and D. Truffet, "Data partitioning for parallel spatial join processing," *Geoinformatica*, vol. 2, no. 2, pp. 175–204, 1998.

[9] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in *ACM sigmod record*, vol. 24, no. 2. ACM, 1995, pp. 71–79.

[10] B. D. Ripley, *Spatial statistics*. John Wiley & Sons, 2005, vol. 575.