GeoSpark: A Cluster Computing Framework for Processing Large-Scale Spatial Data

Jia Yu
School of Computing, Informatics, and Decision Systems Engineering, Arizona State University
699 S. Mill Avenue, Tempe, AZ
jiayu2@asu.edu

Jinxuan Wu
School of Computing, Informatics, and Decision Systems Engineering, Arizona State University
699 S. Mill Avenue, Tempe, AZ
jinxuanw@asu.edu

Mohamed Sarwat
School of Computing, Informatics, and Decision Systems Engineering, Arizona State University
699 S. Mill Avenue, Tempe, AZ
msarwat@asu.edu

ABSTRACT

This paper introduces GeoSpark an in-memory cluster computing framework for processing large-scale spatial data. GeoSpark consists of three layers: Apache Spark Layer, Spatial RDD Layer and Spatial Query Processing Layer. Apache Spark Layer provides basic Spark functionalities that include loading / storing data to disk as well as regular RDD operations. Spatial RDD Layer consists of three novel Spatial Resilient Distributed Datasets (SRDDs) which extend regular Apache Spark RDDs to support geometrical and spatial objects. GeoSpark provides a geometrical operations library that accesses Spatial RDDs to perform basic geometrical operations (e.g., Overlap, Intersect). System users can leverage the newly defined SRDDs to effectively develop spatial data processing programs in Spark. The Spatial Query Processing Layer efficiently executes spatial query processing algorithms (e.g., Spatial Range, Join, KNN query) on SRDDs. GeoSpark also allows users to create a spatial index (e.g., R-tree, Quad-tree) that boosts spatial data processing performance in each SRDD partition. Preliminary experiments show that GeoSpark achieves better run time performance than its Hadoop-based counterparts (e.g., SpatialHadoop).

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems—*Distributed databases*; H.2.8 [**DATABASE MANAGEMENT**]: Database Applications—*Spatial databases and GIS*

Keywords

Cluster computing; Large-scale data; Spatial data

1. INTRODUCTION

SIGSPATIAL'15 November 03-06, 2015, Bellevue, WA, USA

http://dx.doi.org/10.1145/2820783.2820860

The volume of available spatial data increased tremendously. Such data includes but not limited to: weather 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. Making sense of such spatial data will be beneficial for several applications that may transform science and society. Challenges to building such platform are as follows: *Challenge I: System Scalability.* The underlying database system must be able to digest Petabytes of spatial data, effectively stores it, and allows applications to efficiently retrieve it when necessary. *Challenge II: Interactive Performance.* The underlying spatial data processing system must figure out effective ways to process user's request in a sub-second response time.

Apache Spark is an in-memory cluster computing system. Spark provides a novel data abstraction called resilient distributed datasets (RDDs) [9] 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). Unfortunately, Spark does not provide support for spatial data and operations. Hence, users need to perform the tedious task of programming their own spatial data processing jobs on top of Spark.

This paper introduces GeoSpark [1] an in-memory cluster computing system for processing large-scale spatial data. GeoSpark extends the core of Apache Spark to support spatial data types, indexes, and operations. In other words, the system extends the resilient distributed datasets (RDDs) concept to support spatial data. The key contributions of this paper are as follows: (1) GeoSpark as a full-fledged cluster computing framework to load, process, and analyze large-scale spatial data in Apache Spark. (2) A set of out-of-the-box Spatial Resilient Distributed Dataset (SRDD) types (e.g., Point RDD and Polygon RDD) that provide in house support for geometrical and distance operations. SRDDS provides an Application Programming Interface (API) for Apache Spark programmers to easily develop their spatial analysis programs. (3) Spatial data indexing strategies that partition the input Spatial RDD using a grid structure and assign grids to machines for parallel execution. GeoSpark

[1]GeoSpark website: `http://geospark.datasyslab.org`

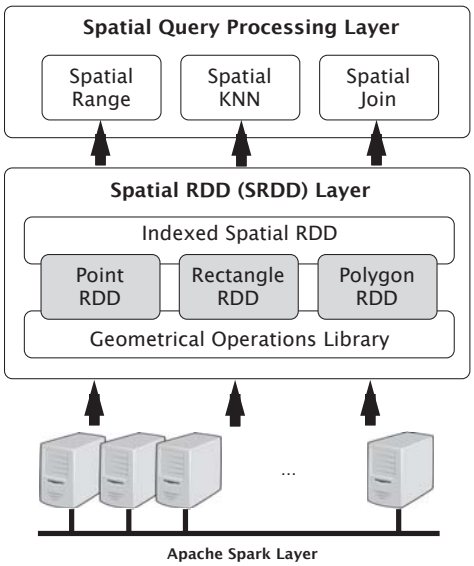

Figure 1: GeoSpark Overview

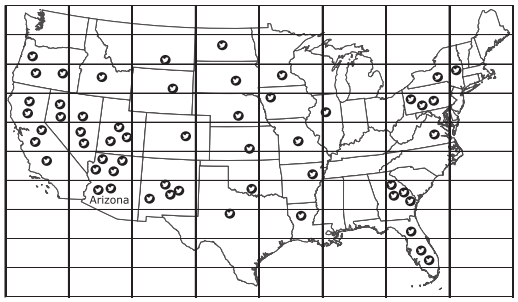

Figure 2: SRDD partitioning

also adaptively decides whether a spatial index needs to be created locally on a Spatial RDD partition to strike a balance between the run time performance and memory/cpu utilization in the cluster. Experiments show that GeoSpark achieves better run time performance than its Hadoop-based counterparts (e.g., SpatialHadoop).

The rest of this paper is organized as follows. Section 2 highlights the related work. GeoSpark architecture is given in Section 3. Preliminary experiments that evaluate GeoSpark are given in Section 4. Finally, Section 5 concludes the paper.

2. BACKGROUND AND RELATED WORK

Spatial Database Systems. Spatial database operations are vital for spatial analysis and spatial data mining. Spatial range queries inquire about certain spatial objects exist in a certain area (e.g., Return all parks in Phoenix). Spatial join queries are queries that combine two datasets or more with a spatial predicate, such as distance relations (e.g., find the parks that have rivers in Phoenix). Spatial k-Nearest Neighbors queries find the k nearest objects to a given spatial object (e.g., show the 10 nearby restaurants). Spatial query processing algorithms usually make use of spatial indexes to reduce the query latency. For instance, R-Tree [3] provides an efficient data partitioning strategy to efficiently index spatial data. Its key idea is that group nearby objects and put them in the next higher level node of the tree. Quad-Tree [8] is also a spatial index that recursively divides a two-dimensional space into four quadrants.

Parallel and Distributed Spatial Data Processing. As the development of distributed data processing system, more and more people in geospatial area direct their attention to deal with massive geospatial data with distributed frameworks. Hadoop-GIS [1] utilizes global partition indexing and customizable on demand local spatial indexing to achieve efficient query processing. SpatialHadoop [2], a comprehensive extension to Hadoop, has native support for spatial data by modifying the underlying code of Hadoop. MD-HBase [6] extends HBase, a non-relational database runs on top of Hadoop, to support multidimensional indexes which allows for efficient retrieval of points using range and kNN queries. Parallel SECONDO [4] combines Hadoop with SECONDO, a database which can handle non-standard data types, like spatial data, usually not supported by standard systems. Although these systems have well-developed functions, all of them are implemented on Hadoop framework. That means they cannot avoid the disadvantages of Hadoop, especially a large number of reads and writes on disks.

3. 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 3.1). (3) *Spatial Query Processing Layer* (Section 3.2).

3.1 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 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

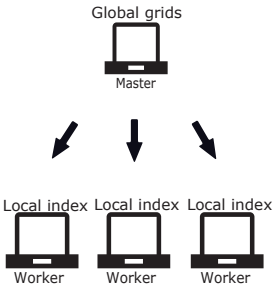

Figure 3: Query execution model

number of equal geographical size grid cells which compose a global grid file. 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.

3.2 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. Figure 3 gives the general execution model followed by GeoSpark . This execution model implements the algorithms proposed by [5] and [10]. 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[7], 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.

4. EXPERIMENTS

This section provides preliminary experimental evaluation that studies the run time performance of the following large-scale spatial data processing systems: (1) `GeoSpark_NoIndex | QuadTree | RTree`: GeoSpark approach without spatial index, with spatial Quad-Tree or R-Tree index. In these approaches, data is partitioned according grids. Required spatial indexes are created on each partition after data partitioned. (2) `Spatial-Hadoop_NoIndex | RTree`: SpatialHadoop approach without spatial index or with spatial R-Tree index.

Experimental Setup. Our cluster setting on Amazon EC2 is as follows: (1) CPU per worker: 8 Intel Xeon Processors operating at 2.5 GHz with Turbo up to 3.3 GHz. (2) Memory per worker: 61 GB in total and 50 GB registered memory in Spark and Hadoop. (3) Storage per worker: Amazon general purpose SSD. We deploy Ganglia, a scalable distributed monitoring system for high performance computing systems such as clusters, on our Amazon EC2 experimental cluster.

Datasets. We use three real spatial datasets extracted from TIGER files in our experiments: Zcta510 1.5 GB dataset, Areawater 6.5 GB dataset and Edges 62 GB dataset. They contain all the cities, all the lakes and all the meaningful boundaries in the US in rectangle format correspondingly. All of the datasets are preprocessed by SpatialHadoop and are open to the public on its website [2].

4.1 Impact of Data Size

This section compares GeoSpark on TIGER Areawater 6.5 GB dataset with TIGER Edges 62 GB dataset as well as SpatialHadoop. They are tested on 16 nodes cluster. Their performance are shown in Figure 5. As depicted in Figure 5, GeoSpark and SpatialHadoop cost more run time on the large dataset than that on the small one. However, GeoSpark achieves much better run time performance than SpatialHadoop in both datasets. This superiority is more obvious on the small dataset. The reason is that GeoSpark can cache more percentage of the intermediate data in memory on the small scale input than that on the large one. That accelerates the processing speed.

4.2 Performance of Spatial Iterative Analysis

Spatial co-location pattern recognition is defined as two or more species are ofter located in a neighborhood rela-

```
double threshold = THRESHOLD;
double baseDistance = 1.0;
double IntervalDistance = 0.5;
int counter=0;
double CoLocationCoefficient=0.0;
// Initialize IndexedPointRDD
IndexedPointRDD target =
    new IndexedPointRDD (SparkContext,
      DatasetLocation);
// Iterative Adjacency Matrix Calculation
while(CoLocationCoefficient>threshold) {
  PairRDD glbAdjMat =
    target.SpatialJoinQuery(target, WITHIN,
    baseDistance + counter * IntervalDistance);
  CoLocationCoefficient=
    CalculateCoLocation(glbAdjMat);
  counter++;
}
return baseDistance + (counter-1) *
    IntervalDistance;
```

Figure 4: *Adjacency Matrix* (Java code) in GeoSpark

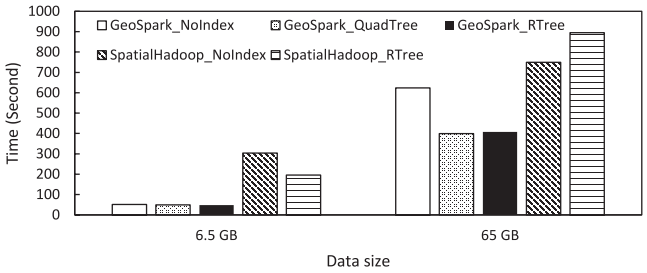

Figure 5: Run Time Performance for Spatial Join Over Different Spatial Datasets

tionship. It usually executes multiple times to form a 2-dimension curve for observation. This calculation needs the adjacent matrix between two type of objects which is the result of a join query. Sample code for finding adjacent matrix is given in Figure 4. We iteratively query GeoSpark SRDDs two times with different distances which can be defined as neighborhood relationships in adjacent matrix. Since SpatialHadoop doesn't natively support iterative jobs, we have to run `SpatialHadoop_RTree` two times for a reasonable comparison. We use the first point column in both of TIGER Zcta 1.5 GB dataset and TIGER Edges 62 GB dataset and join them.

As shown in Figure 6, GeoSpark outperforms SpatialHadoop in spatial co-location. And their performances are also improved when we increase the number of machines per cluster. GeoSpark only costs the quarter time of SpatialHadoop. The main reason behind is that GeoSpark caches these datasets in memory with SRDDs automatically after loads from the storage system. The iterative jobs like spatial co-location can invoke these SRDDs multiple times from memory without any data transformation and data loading. SpatialHadoop has to read and transform the original datasets again and again.

5. CONCLUSION AND FUTURE WORK

This paper introduced GeoSpark an in-memory cluster computing framework for processing large-scale spatial data. GeoSpark provides an API for Apache Spark programmers to easily develop spatial analysis applications. Moreover, GeoSpark provides native support for spatial data

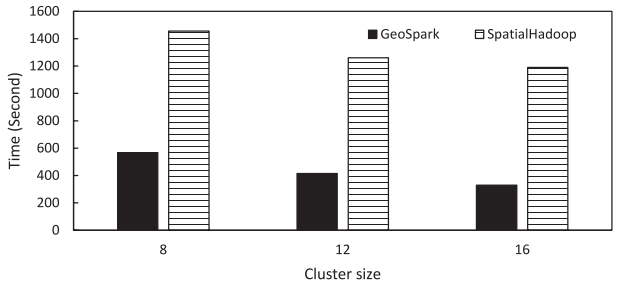

Figure 6: Run Time Performance for Spatial Co-location Pattern Recognition

indexing and query processing algorithms in Apache Spark to efficiently analyze spatial data at scale. Experiments on data sizes and spatial analysis show that GeoSpark achieves better run time performance than its MapReduce-based counterparts (e.g., SpatialHadoop). The proposed ideas are packaged into an open source software artifact. In the future, we envision GeoSpark to be used by Earth and Space Scientists, Geographers, Politicians, Commercial Institutions to analyze spatial data at scale. We also expect the scientific community will contribute to GeoSpark and add new functionalities on top-of it that serve novel spatial data analysis applications.

6. 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. *PVLDB*, 6(11):1009–1020, 2013.

[2] A. Eldawy and M. F. Mokbel. A demonstration of spatialhadoop: An efficient mapreduce framework for spatial data. *PVLDB*, 6(12):1230–1233, 2013.

[3] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[4] J. Lu and R. H. Guting. Parallel Secondo: Boosting Database Engines with Hadoop. In *ICPADS*, pages 738 –743, 2012.

[5] 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*, pages 697–705. IEEE, 2002.

[6] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi. MD-Hbase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In *MDM*, pages 7–16, 2011.

[7] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *ACM SIGMOD record*, volume 24, pages 71–79. ACM, 1995.

[8] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[9] 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 *NSDI*, pages 15–28, 2012.

[10] X. Zhou, D. J. Abel, and D. Truffet. Data partitioning for parallel spatial join processing. *Geoinformatica*, 2(2):175–204, 1998.